Recursion Theory I

Michael Nedzelsky

December 14, 2021

Abstract

This document presents the formalization of introductory material from recursion theory — definitions and basic properties of primitive recursive functions, Cantor pairing function and computably enumerable sets (including a proof of existence of a one-complete computably enumerable set and a proof of the Rice's theorem).

Contents

1 Cantor pairing function

```
theory CPair
imports Main
begin
```

We introduce a particular coding *c-pair* from ordered pairs of natural numbers to natural numbers. See [1] and the Isabelle documentation for more information.

1.1 Pairing function

```
definition
  sf :: nat ⇒ nat where
  sf-def: sf x = x ∗ (x+1) div 2

definition
  c-pair :: nat ⇒ nat ⇒ nat where
  c-pair x y = sf (x+y) + x

lemma sf-at-0: sf 0 = 0 by (simp add: sf-def)

lemma sf-at-1: sf 1 = 1 by (simp add: sf-def)

lemma sf-at-Suc: sf (x+1) = sf x + x + 1
proof −
  have S1: sf(x+1) = ((x+1)∗(x+2)) div 2 by (simp add: sf-def)
  have S2: (x+1)∗(x+2) = x∗(x+1) + 2∗(x+1) by (auto)
  have S2-1: ⋀ x y. x=y ⟹ x div 2 = y div 2 by auto
  from S2 have S3: (x+1)∗(x+2) div 2 = (x∗(x+1) + 2∗(x+1)) div 2 by (rule S2-1)
  have S4: (0::nat) < 2 by (auto)
  from S4 have S5: (x∗(x+1) + 2∗(x+1)) div 2 = (x+1) + x∗(x+1) div 2 by simp
```

```
  from S1 S3 S5 show ?thesis by (simp add: sf-def)
qed

lemma arg-le-sf: x ≤ sf x
proof −
  have x + x ≤ x∗(x + 1) by simp
  hence (x + x) div 2 ≤ x∗(x+1) div 2 by (rule div-le-mono)
  hence x ≤ x∗(x+1) div 2 by simp
  thus ?thesis by (simp add: sf-def)
qed

lemma sf-mono: x ≤ y ⟹ sf x ≤ sf y
proof −
  assume A1: x ≤ y
  then have x+1 ≤ y+1 by (auto)
  with A1 have x∗(x+1) ≤ y∗(y+1) by (rule mult-le-mono)
  then have x∗(x+1) div 2 ≤ y∗(y+1) div 2 by (rule div-le-mono)
  thus ?thesis by (simp add: sf-def)
qed

lemma sf-strict-mono: x < y ⟹ sf x < sf y
proof −
  assume A1: x < y
  from A1 have S1: x+1 ≤ y by simp
  from S1 sf-mono have S2: sf (x+1) ≤ sf y by (auto)
  from sf-at-Suc have S3: sf x < sf (x+1) by (auto)
  from S2 S3 show ?thesis by (auto)
qed

lemma sf-posI: x > 0 ⟹ sf(x) > 0
proof −
  assume A1: x > 0
  then have sf(0) < sf(x) by (rule sf-strict-mono)
  then show ?thesis by simp
qed

lemma arg-less-sf: x > 1 ⟹ x < sf(x)
proof −
  assume A1: x > 1
  let ?y = x−(1::nat)
  from A1 have S1: x = ?y+1 by simp
  from A1 have ?y > 0 by simp
  then have S2: sf(?y) > 0 by (rule sf-posI)
  have sf(?y+1) = sf(?y) + ?y + 1 by (rule sf-at-Suc)
  with S1 have sf(x) = sf(?y) + x by simp
  with S2 show ?thesis by simp
qed

lemma sf-eq-arg: sf x = x ⟹ x ≤ 1
```

```
proof −
  assume sf(x) = x
  then have ¬ (x < sf(x)) by simp
  then have (¬ (x > 1)) by (auto simp add: arg-less-sf)
  then show ?thesis by simp
qed

lemma sf-le-sfD: sf x ≤ sf y ⟹ x ≤ y
proof −
  assume A1: sf x ≤ sf y
  have S1: y < x ⟹ sf y < sf x by (rule sf-strict-mono)
  have S2: y < x ∨ x ≤ y by (auto)
  from A1 S1 S2 show ?thesis by (auto)
qed

lemma sf-less-sfD: sf x < sf y ⟹ x < y
proof −
  assume A1: sf x < sf y
  have S1: y ≤ x ⟹ sf y ≤ sf x by (rule sf-mono)
  have S2: y ≤ x ∨ x < y by (auto)
  from A1 S1 S2 show ?thesis by (auto)
qed

lemma sf-inj: sf x = sf y ⟹ x = y
proof −
  assume A1: sf x = sf y
  have S1: sf x ≤ sf y ⟹ x ≤ y by (rule sf-le-sfD)
  have S2: sf y ≤ sf x ⟹ y ≤ x by (rule sf-le-sfD)
  from A1 have S3: sf x ≤ sf y ∧ sf y ≤ sf x by (auto)
  from S3 S1 S2 have S4: x ≤ y ∧ y ≤ x by (auto)
  from S4 show ?thesis by (auto)
qed
```

Auxiliary lemmas

```
lemma sf-aux1: x + y < z ⟹ sf(x+y) + x < sf(z)
proof −
  assume A1: x+y < z
  from A1 have S1: x+y+1 ≤ z by (auto)
  from S1 have S2: sf(x+y+1) ≤ sf(z) by (rule sf-mono)
  have S3: sf(x+y+1) = sf(x+y) + (x+y)+1 by (rule sf-at-Suc)
  from S3 S2 have S4: sf(x+y) + (x+y) + 1 ≤ sf(z) by (auto)
  from S4 show ?thesis by (auto)
qed

lemma sf-aux2: sf(z) ≤ sf(x+y) + x ⟹ z ≤ x+y
proof −
  assume A1: sf(z) ≤ sf(x+y) + x
  from A1 have S1: ¬ sf(x+y) +x < sf(z) by (auto)
  from S1 sf-aux1 have S2: ¬ x+y < z by (auto)
```

```
  from S2 show ?thesis by (auto)
qed

lemma sf-aux3: sf(z) + m < sf(z+1) ⟹ m ≤ z
proof −
  assume A1: sf(z) + m < sf(z+1)
  have S1: sf(z+1) = sf(z) + z + 1 by (rule sf-at-Suc)
  from A1 S1 have S2: sf(z) + m < sf(z) + z + 1 by (auto)
  from S2 have S3: m < z + 1 by (auto)
  from S3 show ?thesis by (auto)
qed

lemma sf-aux4: (s::nat) < t ⟹ (sf s) + s < sf t
proof −
  assume A1: (s::nat) < t
  have s∗(s + 1) + 2∗(s+1) ≤ t∗(t+1)
  proof −
    from A1 have S1: (s::nat) + 1 ≤ t by (auto)
    from A1 have (s::nat) + 2 ≤ t+1 by (auto)
    with S1 have ((s::nat)+1)∗(s+2) ≤ t∗(t+1) by (rule mult-le-mono)
    thus ?thesis by (auto)
  qed
  then have S1: (s∗(s+1) + 2∗(s+1)) div 2 ≤  t∗(t+1) div 2 by (rule div-le-mono)
  have (0::nat) < 2 by (auto)
  then have (s∗(s+1) + 2∗(s+1)) div 2 = (s+1) + (s∗(s+1)) div 2 by simp
  with S1 have (s∗(s+1)) div 2 + (s+1) ≤ t∗(t+1) div 2 by (auto)
  then have (s∗(s+1)) div 2 + s < t∗(t+1) div 2 by (auto)
  thus ?thesis by (simp add: sf-def)
qed
```

Basic properties of c_pair function

```
lemma sum-le-c-pair: x + y ≤ c-pair x y
proof −
  have x+y ≤ sf(x+y) by (rule arg-le-sf)
  thus ?thesis by (simp add: c-pair-def)
qed

lemma arg1-le-c-pair: x ≤ c-pair x y
proof −
  have (x::nat) ≤ x + y by (simp)
  moreover have x + y ≤ c-pair x y by (rule sum-le-c-pair)
  ultimately show ?thesis by (simp)
qed

lemma arg2-le-c-pair: y ≤ c-pair x y
proof −
  have (y::nat) ≤ x + y by (simp)
  moreover have x + y ≤ c-pair x y by (rule sum-le-c-pair)
  ultimately show ?thesis by (simp)
```

qed

lemma *c-pair-sum-mono*: (*x1*::*nat*) + *y1* < *x2* + *y2* ⟹ *c-pair x1 y1* < *c-pair x2 y2*
proof −
assume (*x1*::*nat*) + *y1* < *x2* + *y2*
hence *sf* (*x1*+*y1*) + (*x1*+*y1*) < *sf*(*x2*+*y2*) **by** (*rule sf-aux4*)
hence *sf* (*x1*+*y1*) + *x1* < *sf*(*x2*+*y2*) + *x2* **by** (*auto*)
thus *?thesis* **by** (*simp add*: *c-pair-def*)
qed

lemma *c-pair-sum-inj*: *c-pair x1 y1* = *c-pair x2 y2* ⟹ *x1* + *y1* = *x2* + *y2*
proof −
assume *A1*: *c-pair x1 y1* = *c-pair x2 y2*
have *S1*: (*x1*::*nat*) + *y1* < *x2* + *y2* ⟹ *c-pair x1 y1* ≠ *c-pair x2 y2* **by** (*rule less-not-refl3*, *rule c-pair-sum-mono*, *auto*)
have *S2*: (*x2*::*nat*) + *y2* < *x1* + *y1* ⟹ *c-pair x1 y1* ≠ *c-pair x2 y2* **by** (*rule less-not-refl2*, *rule c-pair-sum-mono*, *auto*)
from *S1 S2* **have** (*x1*::*nat*) + *y1* ≠ *x2* + *y2* ⟹ *c-pair x1 y1* ≠ *c-pair x2 y2* **by** (*arith*)
with *A1* **show** *?thesis* **by** (*auto*)
qed

lemma *c-pair-inj*: *c-pair x1 y1* = *c-pair x2 y2* ⟹ *x1* = *x2* ∧ *y1* = *y2*
proof −
assume *A1*: *c-pair x1 y1* = *c-pair x2 y2*
from *A1* **have** *S1*: *x1* + *y1* = *x2* + *y2* **by** (*rule c-pair-sum-inj*)
from *A1* **have** *S2*: *sf* (*x1*+*y1*) + *x1* = *sf* (*x2*+*y2*) + *x2* **by** (*unfold c-pair-def*)
from *S1 S2* **have** *S3*: *x1* = *x2* **by** (*simp*)
from *S1 S3* **have** *S4*: *y1* = *y2* **by** (*simp*)
from *S3 S4* **show** *?thesis* **by** (*auto*)
qed

lemma *c-pair-inj1*: *c-pair x1 y1* = *c-pair x2 y2* ⟹ *x1* = *x2* **by** (*frule c-pair-inj*, *drule conjunct1*)

lemma *c-pair-inj2*: *c-pair x1 y1* = *c-pair x2 y2* ⟹ *y1* = *y2* **by** (*frule c-pair-inj*, *drule conjunct2*)

lemma *c-pair-strict-mono1*: *x1* < *x2* ⟹ *c-pair x1 y* < *c-pair x2 y*
proof −
assume *x1* < *x2*
then have *x1* + *y* < *x2* + *y* **by** *simp*
then show *?thesis* **by** (*rule c-pair-sum-mono*)
qed

lemma *c-pair-mono1*: *x1* ≤ *x2* ⟹ *c-pair x1 y* ≤ *c-pair x2 y*
proof −
assume *A1*: *x1* ≤ *x2*

```
  show ?thesis
  proof cases
    assume x1 < x2
    then have c-pair x1 y < c-pair x2 y by (rule c-pair-strict-mono1)
    then show ?thesis by simp
  next
    assume ¬ x1 < x2
    with A1 have x1 = x2 by simp
    then show ?thesis by simp
  qed
qed
```

```
lemma c-pair-strict-mono2: y1 < y2 ⟹ c-pair x y1 < c-pair x y2
proof −
  assume A1: y1 < y2
  from A1 have S1: x + y1 < x + y2 by simp
  then show ?thesis by (rule c-pair-sum-mono)
qed
```

```
lemma c-pair-mono2: y1 ≤ y2 ⟹ c-pair x y1 ≤ c-pair x y2
proof −
  assume A1: y1 ≤ y2
  show ?thesis
  proof cases
    assume y1 < y2
    then have c-pair x y1 < c-pair x y2 by (rule c-pair-strict-mono2)
    then show ?thesis by simp
  next
    assume ¬ y1 < y2
    with A1 have y1 = y2 by simp
    then show ?thesis by simp
  qed
qed
```

1.2 Inverse mapping

c-fst and *c-snd* are the functions which yield the inverse mapping to *c-pair*.

```
definition
  c-sum :: nat ⇒ nat where
  c-sum u = (LEAST z. u < sf (z+1))
```

```
definition
  c-fst :: nat ⇒ nat where
  c-fst u = u − sf (c-sum u)
```

```
definition
  c-snd :: nat ⇒ nat where
  c-snd u = c-sum u − c-fst u
```

```
lemma arg-less-sf-at-Suc-of-c-sum: u < sf ((c-sum u) + 1)
proof -
  have u+1 ≤ sf(u+1) by (rule arg-le-sf)
  hence u < sf(u+1) by simp
  thus ?thesis by (unfold c-sum-def, rule LeastI)
qed

lemma arg-less-sf-imp-c-sum-less-arg: u < sf(x) ⟹ c-sum u < x
proof -
  assume A1: u < sf(x)
  then show ?thesis
  proof (cases x)
    assume x=0
    with A1 show ?thesis by (simp add: sf-def)
  next
    fix y
    assume A2: x = Suc y
    show ?thesis
    proof -
      from A1 A2 have u < sf(y+1) by simp
      hence (Least (%z. u < sf (z+1))) ≤ y by (rule Least-le)
      hence c-sum u ≤ y by (fold c-sum-def)
      with A2 show ?thesis by simp
    qed
  qed
qed

lemma sf-c-sum-le-arg: u ≥ sf (c-sum u)
proof -
  let ?z = c-sum u
  from arg-less-sf-at-Suc-of-c-sum have S1: u < sf (?z+1) by (auto)
  have S2: ¬ c-sum u < c-sum u by (auto)
  from arg-less-sf-imp-c-sum-less-arg S2 have S3: ¬ u < sf (c-sum u)  by (auto)
  from S3 show ?thesis by (auto)
qed

lemma c-sum-le-arg: c-sum u ≤ u
proof -
  have c-sum u ≤ sf (c-sum u) by (rule arg-le-sf)
  moreover have sf(c-sum u) ≤ u by (rule sf-c-sum-le-arg)
  ultimately show ?thesis by simp
qed

lemma c-sum-of-c-pair [simp]: c-sum (c-pair x y) = x + y
proof -
  let ?u = c-pair x y
  let ?z = c-sum ?u
  have S1: ?u < sf(?z+1) by (rule arg-less-sf-at-Suc-of-c-sum)
  have S2: sf(?z) ≤ ?u by (rule sf-c-sum-le-arg)
```

```
  from S1 have S3: sf(x+y)+x < sf(?z+1) by (simp add: c-pair-def)
  from S2 have S4: sf(?z) ≤ sf(x+y) + x by (simp add: c-pair-def)
  from S3 have S5: sf(x+y) < sf(?z+1) by (auto)
  from S5 have S6: x+y < ?z+1 by (rule sf-less-sfD)
  from S6 have S7: x+y ≤ ?z by (auto)
  from S4 have S8: ?z ≤ x+y by (rule sf-aux2)
  from S7 S8 have S9: ?z = x+y by (auto)
  from S9 show ?thesis by (simp)
qed

lemma c-fst-of-c-pair[simp]: c-fst (c-pair x y) = x
proof −
  let ?u = c-pair x y
  have c-sum ?u = x + y by simp
  hence c-fst ?u = ?u − sf(x+y) by (simp add: c-fst-def)
  moreover have ?u = sf(x+y) + x by (simp add: c-pair-def)
  ultimately show ?thesis by (simp)
qed

lemma c-snd-of-c-pair[simp]: c-snd (c-pair x y) = y
proof −
  let ?u = c-pair x y
  have c-sum ?u = x + y by simp
  moreover have c-fst ?u = x by simp
  ultimately show ?thesis by (simp add: c-snd-def)
qed

lemma c-pair-at-0: c-pair 0 0 = 0 by (simp add: sf-def c-pair-def)

lemma c-fst-at-0: c-fst 0 = 0
proof −
  have c-pair 0 0 = 0 by (rule c-pair-at-0)
  hence c-fst 0 = c-fst (c-pair 0 0) by simp
  thus ?thesis by simp
qed

lemma c-snd-at-0: c-snd 0 = 0
proof −
  have c-pair 0 0 = 0 by (rule c-pair-at-0)
  hence c-snd 0 = c-snd (c-pair 0 0) by simp
  thus ?thesis by simp
qed

lemma sf-c-sum-plus-c-fst: sf(c-sum u) + c-fst u = u
proof −
  have S1: sf(c-sum u) ≤ u by (rule sf-c-sum-le-arg)
  have S2: c-fst u = u − sf(c-sum u) by (simp add: c-fst-def)
  from S1 S2 show ?thesis by (auto)
qed
```

lemma *c-fst-le-c-sum*: *c-fst u* $\leq$ *c-sum u*
proof –
  have *S1*: *sf*(*c-sum u*) + *c-fst u* = *u* **by** (*rule sf-c-sum-plus-c-fst*)
  have *S2*: *u* < *sf*((*c-sum u*) + *1*) **by** (*rule arg-less-sf-at-Suc-of-c-sum*)
  from *S1 S2 sf-aux3* **show** *?thesis* **by** (*auto*)
qed

lemma *c-snd-le-c-sum*: *c-snd u* $\leq$ *c-sum u* **by** (*simp add*: *c-snd-def*)

lemma *c-fst-le-arg*: *c-fst u* $\leq$ *u*
proof –
  have *c-fst u* $\leq$ *c-sum u* **by** (*rule c-fst-le-c-sum*)
  moreover have *c-sum u* $\leq$ *u* **by** (*rule c-sum-le-arg*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *c-snd-le-arg*: *c-snd u* $\leq$ *u*
proof –
  have *c-snd u* $\leq$ *c-sum u* **by** (*rule c-snd-le-c-sum*)
  moreover have *c-sum u* $\leq$ *u* **by** (*rule c-sum-le-arg*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *c-sum-is-sum*: *c-sum u* = *c-fst u* + *c-snd u* **by** (*simp add*: *c-snd-def c-fst-le-c-sum*)

lemma *proj-eq-imp-arg-eq*: $[\![$ *c-fst u* = *c-fst v*; *c-snd u* = *c-snd v*$]\!] \Longrightarrow$ *u* = *v*
proof –
  assume *A1*: *c-fst u* = *c-fst v*
  assume *A2*: *c-snd u* = *c-snd v*
  from *A1 A2 c-sum-is-sum* **have** *S1*: *c-sum u* = *c-sum v* **by** (*auto*)
  have *S2*: *sf*(*c-sum u*) + *c-fst u* = *u* **by** (*rule sf-c-sum-plus-c-fst*)
  from *A1 S1 S2* **have** *S3*: *sf*(*c-sum v*) + *c-fst v* = *u* **by** (*auto*)
  from *S3 sf-c-sum-plus-c-fst* **show** *?thesis* **by** (*auto*)
qed

lemma *c-pair-of-c-fst-c-snd*[*simp*]: *c-pair* (*c-fst u*) (*c-snd u*) = *u*
proof –
  let *?x* = *c-fst u*
  let *?y* = *c-snd u*
  have *S1*: *c-pair ?x ?y* = *sf*(*?x* + *?y*) + *?x* **by** (*simp add*: *c-pair-def*)
  have *S2*: *c-sum u* = *?x* + *?y* **by** (*rule c-sum-is-sum*)
  from *S1 S2* **have** *c-pair ?x ?y* = *sf*(*c-sum u*) + *c-fst u* **by** (*auto*)
  thus *?thesis* **by** (*simp add*: *sf-c-sum-plus-c-fst*)
qed

lemma *c-sum-eq-arg*: *c-sum x* = *x* $\Longrightarrow$ *x* $\leq$ *1*
proof –

```
  assume A1: c-sum x = x
  have S1: sf(c-sum x) + c-fst x = x by (rule sf-c-sum-plus-c-fst)
  from A1 S1 have S2: sf x + c-fst x = x by simp
  have S3: x ≤ sf x by (rule arg-le-sf)
  from S2 S3 have sf(x)=x by simp
  thus ?thesis by (rule sf-eq-arg)
qed

lemma c-sum-eq-arg-2: c-sum x = x ⟹ c-fst x = 0
proof −
  assume A1: c-sum x = x
  have S1: sf(c-sum x) + c-fst x = x by (rule sf-c-sum-plus-c-fst)
  from A1 S1 have S2: sf x + c-fst x = x by simp
  have S3: x ≤ sf x by (rule arg-le-sf)
  from S2 S3 show ?thesis by simp
qed

lemma c-fst-eq-arg: c-fst x = x ⟹ x = 0
proof −
  assume A1: c-fst x = x
  have S1: c-fst x ≤ c-sum x by (rule c-fst-le-c-sum)
  have S2: c-sum x ≤ x by (rule c-sum-le-arg)
  from A1 S1 S2 have c-sum x = x by simp
  then have c-fst x = 0 by (rule c-sum-eq-arg-2)
  with A1 show ?thesis by simp
qed

lemma c-fst-less-arg: x > 0 ⟹ c-fst x < x
proof −
  assume A1: x > 0
  show ?thesis
  proof cases
    assume c-fst x < x
    then show ?thesis by simp
  next
    assume ¬ c-fst x < x
    then have S1: c-fst x ≥ x by simp
    have c-fst x ≤ x by (rule c-fst-le-arg)
    with S1 have c-fst x = x by simp
    then have x = 0 by (rule c-fst-eq-arg)
    with A1 show ?thesis by simp
  qed
qed

lemma c-snd-eq-arg: c-snd x = x ⟹ x ≤ 1
proof −
  assume A1: c-snd x = x
  have S1: c-snd x ≤ c-sum x by (rule c-snd-le-c-sum)
  have S2: c-sum x ≤ x by (rule c-sum-le-arg)
```

```
  from A1 S1 S2 have c-sum x = x by simp
  then show ?thesis by (rule c-sum-eq-arg)
qed

lemma c-snd-less-arg: x > 1 ⟹ c-snd x < x
proof −
  assume A1: x > 1
  show ?thesis
  proof cases
    assume c-snd x < x
    then show ?thesis .
  next
    assume ¬ c-snd x < x
    then have S1: c-snd x ≥ x by auto
    have c-snd x ≤ x by (rule c-snd-le-arg)
    with S1 have c-snd x = x by simp
    then have x ≤ 1 by (rule c-snd-eq-arg)
    with A1 show ?thesis by simp
  qed
qed

end
```

2 Primitive recursive functions

```
theory PRecFun imports CPair
begin
```

This theory contains definition of the primitive recursive functions.

2.1 Basic definitions

```
primrec
  PrimRecOp :: (nat ⇒ nat) ⇒ (nat ⇒ nat ⇒ nat ⇒ nat) ⇒ (nat ⇒ nat ⇒ nat)
where
  PrimRecOp g h 0 x  = g x
| PrimRecOp g h (Suc y) x = h y (PrimRecOp g h y x) x

primrec
  PrimRecOp-last :: (nat ⇒ nat) ⇒ (nat ⇒ nat ⇒ nat ⇒ nat) ⇒ (nat ⇒ nat ⇒
nat)
where
  PrimRecOp-last g h x 0  = g x
| PrimRecOp-last g h x (Suc y)= h x (PrimRecOp-last g h x y) y

primrec
  PrimRecOp1 :: nat ⇒ (nat ⇒ nat ⇒ nat) ⇒ (nat ⇒ nat)
where
  PrimRecOp1 a h 0 = a
```

```
| PrimRecOp1 a h (Suc y) = h y (PrimRecOp1 a h y)

inductive-set
  PrimRec1 :: (nat ⇒ nat) set and
  PrimRec2 :: (nat ⇒ nat ⇒ nat) set and
  PrimRec3 :: (nat ⇒ nat ⇒ nat ⇒ nat) set
where
  zero: (λ x. 0) ∈ PrimRec1
  | suc:   Suc ∈ PrimRec1
  | id1-1: (λ x. x) ∈ PrimRec1
  | id2-1: (λ x y. x) ∈ PrimRec2
  | id2-2: (λ x y. y) ∈ PrimRec2
  | id3-1: (λ x y z. x) ∈ PrimRec3
  | id3-2: (λ x y z. y) ∈ PrimRec3
  | id3-3: (λ x y z. z) ∈ PrimRec3
  | comp1-1: ⟦ f ∈ PrimRec1; g ∈ PrimRec1⟧ ⟹ (λ x. f (g x)) ∈ PrimRec1
  | comp1-2: ⟦ f ∈ PrimRec1; g ∈ PrimRec2⟧ ⟹ (λ x y. f (g x y)) ∈ PrimRec2
  | comp1-3: ⟦ f ∈ PrimRec1; g ∈ PrimRec3⟧ ⟹ (λ x y z. f (g x y z)) ∈ PrimRec3
  | comp2-1: ⟦ f ∈ PrimRec2; g ∈ PrimRec1; h ∈ PrimRec1⟧ ⟹ (λ x. f (g x) (h x)) ∈ PrimRec1
  | comp3-1: ⟦ f ∈ PrimRec3; g ∈ PrimRec1; h ∈ PrimRec1; k ∈ PrimRec1⟧ ⟹ (λ x. f (g x) (h x) (k x)) ∈ PrimRec1
  | comp2-2: ⟦ f ∈ PrimRec2; g ∈ PrimRec2; h ∈ PrimRec2⟧ ⟹ (λ x y. f (g x y) (h x y)) ∈ PrimRec2
  | comp2-3: ⟦ f ∈ PrimRec2; g ∈ PrimRec3; h ∈ PrimRec3⟧ ⟹ (λ x y z. f (g x y z) (h x y z)) ∈ PrimRec3
  | comp3-2: ⟦ f ∈ PrimRec3; g ∈ PrimRec2; h ∈ PrimRec2; k ∈ PrimRec2⟧ ⟹ (λ x y. f (g x y) (h x y) (k x y)) ∈ PrimRec2
  | comp3-3: ⟦ f ∈ PrimRec3; g ∈ PrimRec3; h ∈ PrimRec3; k ∈ PrimRec3⟧ ⟹ (λ x y z. f (g x y z) (h x y z) (k x y z)) ∈ PrimRec3
  | prim-rec: ⟦ g ∈ PrimRec1; h ∈ PrimRec3⟧ ⟹ PrimRecOp g h ∈ PrimRec2

lemmas pr-zero = PrimRec1-PrimRec2-PrimRec3.zero
lemmas pr-suc = PrimRec1-PrimRec2-PrimRec3.suc
lemmas pr-id1-1 = PrimRec1-PrimRec2-PrimRec3.id1-1
lemmas pr-id2-1 = PrimRec1-PrimRec2-PrimRec3.id2-1
lemmas pr-id2-2 = PrimRec1-PrimRec2-PrimRec3.id2-2
lemmas pr-id3-1 = PrimRec1-PrimRec2-PrimRec3.id3-1
lemmas pr-id3-2 = PrimRec1-PrimRec2-PrimRec3.id3-2
lemmas pr-id3-3 = PrimRec1-PrimRec2-PrimRec3.id3-3
lemmas pr-comp1-1 = PrimRec1-PrimRec2-PrimRec3.comp1-1
lemmas pr-comp1-2 = PrimRec1-PrimRec2-PrimRec3.comp1-2
lemmas pr-comp1-3 = PrimRec1-PrimRec2-PrimRec3.comp1-3
lemmas pr-comp2-1 = PrimRec1-PrimRec2-PrimRec3.comp2-1
lemmas pr-comp2-2 = PrimRec1-PrimRec2-PrimRec3.comp2-2
lemmas pr-comp2-3 = PrimRec1-PrimRec2-PrimRec3.comp2-3
lemmas pr-comp3-1 = PrimRec1-PrimRec2-PrimRec3.comp3-1
lemmas pr-comp3-2 = PrimRec1-PrimRec2-PrimRec3.comp3-2
lemmas pr-comp3-3 = PrimRec1-PrimRec2-PrimRec3.comp3-3
```

```
lemmas pr-rec = PrimRec1-PrimRec2-PrimRec3.prim-rec

ML-file ‹Utils.ML›

named-theorems prec

method-setup prec0 = ‹
  Attrib.thms >> (fn ths => fn ctxt => Method.METHOD (fn facts =>
   HEADGOAL (prec0-tac ctxt (facts @ Named-Theorems.get ctxt @{named-theorems
prec}))))
› apply primitive recursive functions


lemmas [prec] = pr-zero pr-suc pr-id1-1 pr-id2-1 pr-id2-2 pr-id3-1 pr-id3-2 pr-id3-3

lemma pr-swap: f ∈ PrimRec2 ⟹ (λ x y. f y x) ∈ PrimRec2 by prec0

theorem pr-rec-scheme: ⟦ g ∈ PrimRec1; h ∈ PrimRec3; ∀ x. f 0 x = g x; ∀ x y.
f (Suc y) x = h y (f y x) x ⟧ ⟹ f ∈ PrimRec2
proof −
  assume g-is-pr: g ∈ PrimRec1
  assume h-is-pr: h ∈ PrimRec3
  assume f-at-0: ∀ x. f 0 x = g x
  assume f-at-Suc: ∀ x y. f (Suc y) x = h y (f y x) x
  from f-at-0 f-at-Suc have ⋀ x y. f y x  = PrimRecOp g h y x by (induct-tac y,
simp-all)
  then have f = PrimRecOp g h by (simp add: ext)
  with g-is-pr h-is-pr show ?thesis by (simp add: pr-rec)
qed

lemma op-plus-is-pr [prec]: (λ x y. x + y) ∈ PrimRec2
proof (rule pr-swap)
 show (λ x y. y+x) ∈ PrimRec2
  proof −
    have S1: PrimRecOp (λ x. x) (λ x y z. Suc y) ∈ PrimRec2
    proof (rule pr-rec)
      show (λ x. x) ∈ PrimRec1 by (rule pr-id1-1)
    next
      show (λ x y z. Suc y) ∈ PrimRec3 by prec0
    qed
    have (λ x y. y+x) = PrimRecOp (λ x. x) (λ x y z. Suc y) (is - = ?f)
    proof −
      have ⋀ x y. (?f y x = y + x) by (induct-tac y, auto)
      thus ?thesis by (simp add: ext)
    qed
    with S1 show ?thesis by simp
  qed
qed
```

```
lemma op-mult-is-pr [prec]: (λ x y. x∗y) ∈ PrimRec2
proof (rule pr-swap)
 show (λ x y. y∗x) ∈ PrimRec2
  proof −
    have S1: PrimRecOp (λ x. 0) (λ x y z. y+z) ∈ PrimRec2
    proof (rule pr-rec)
      show (λ x. 0) ∈ PrimRec1 by (rule pr-zero)
    next
      show (λ x y z. y+z) ∈ PrimRec3 by prec0
    qed
    have (λ x y. y∗x) = PrimRecOp (λ x. 0) (λ x y z. y+z) (is - = ?f)
    proof −
      have ⋀ x y. (?f y x = y ∗ x) by (induct-tac y, auto)
      thus ?thesis by (simp add: ext)
    qed
    with S1 show ?thesis by simp
  qed
qed

lemma const-is-pr: (λ x. (n::nat)) ∈ PrimRec1
proof (induct n)
  show (λ x. 0) ∈ PrimRec1 by (rule pr-zero)
next
  fix n assume (λ x. n) ∈ PrimRec1
  then show (λ x. Suc n) ∈ PrimRec1 by prec0
qed

lemma const-is-pr-2: (λ x y. (n::nat)) ∈ PrimRec2
proof (rule pr-comp1-2 [where ?f=%x.(n::nat) and ?g=%x y. x])
  show (λ x. n) ∈ PrimRec1 by (rule const-is-pr)
next
  show (λ x y. x) ∈ PrimRec2 by (rule pr-id2-1)
qed

lemma const-is-pr-3: (λ x y z. (n::nat)) ∈ PrimRec3
proof (rule pr-comp1-3 [where ?f=%x.(n::nat) and ?g=%x y z. x])
  show (λ x. n) ∈ PrimRec1 by (rule const-is-pr)
next
  show (λ x y z. x) ∈ PrimRec3 by (rule pr-id3-1)
qed

theorem pr-rec-last: ⟦g ∈ PrimRec1; h ∈ PrimRec3⟧ ⟹ PrimRecOp-last g h ∈ PrimRec2
proof −
  assume A1: g ∈ PrimRec1
  assume A2: h ∈ PrimRec3
  let ?h1 = λ x y z. h z y x
   from A2 pr-id3-3 pr-id3-2 pr-id3-1 have h1-is-pr: ?h1 ∈ PrimRec3 by (rule pr-comp3-3)
```

```
  let ?f1 = PrimRecOp g ?h1
  from A1 h1-is-pr have f1-is-pr: ?f1 ∈ PrimRec2 by (rule pr-rec)
  let ?f = λ x y. ?f1 y x
  from f1-is-pr have f-is-pr: ?f ∈ PrimRec2 by (rule pr-swap)
  have ⋀ x y. ?f x y = PrimRecOp-last g h x y by (induct-tac y, simp-all)
  then have ?f = PrimRecOp-last g h by (simp add: ext)
  with f-is-pr show ?thesis by simp
qed

theorem pr-rec1: h ∈ PrimRec2 ⟹ PrimRecOp1 (a::nat) h ∈ PrimRec1
proof −
  assume A1: h ∈ PrimRec2
  let ?g = (λ x. a)
  have g-is-pr: ?g ∈ PrimRec1 by (rule const-is-pr)
  let ?h1 = (λ x y z. h x y)
  from A1 have h1-is-pr: ?h1 ∈ PrimRec3 by prec0
  let ?f1 = PrimRecOp ?g ?h1
  from g-is-pr h1-is-pr have f1-is-pr: ?f1 ∈ PrimRec2 by (rule pr-rec)
  let ?f = (λ x. ?f1 x 0)
 from f1-is-pr pr-id1-1 pr-zero have f-is-pr: ?f ∈ PrimRec1 by (rule pr-comp2-1)
  have ⋀ y. ?f y = PrimRecOp1 a h y by (induct-tac y, auto)
  then have ?f = PrimRecOp1 a h by (simp add: ext)
  with f-is-pr show ?thesis by (auto)
qed

theorem pr-rec1-scheme: ⟦ h ∈ PrimRec2; f 0 = a; ∀ y. f (Suc y) = h y (f y) ⟧ ⟹ f ∈ PrimRec1
proof −
  assume h-is-pr: h ∈ PrimRec2
  assume f-at-0: f 0 = a
  assume f-at-Suc: ∀ y. f (Suc y) = h y (f y)
   from f-at-0 f-at-Suc have ⋀ y. f y  = PrimRecOp1 a h y by (induct-tac y, simp-all)
  then have f = PrimRecOp1 a h by (simp add: ext)
  with h-is-pr show ?thesis by (simp add: pr-rec1)
qed

lemma pred-is-pr: (λ x. x − (1::nat)) ∈ PrimRec1
proof −
  have S1: PrimRecOp1 0 (λ x y. x) ∈ PrimRec1
  proof (rule pr-rec1)
    show (λ x y. x) ∈ PrimRec2 by (rule pr-id2-1)
  qed
  have (λ x. x−(1::nat)) = PrimRecOp1 0 (λ x y. x) (is - = ?f)
  proof −
    have ⋀ x. (?f x = x−(1::nat)) by (induct-tac x, auto)
    thus ?thesis by (simp add: ext)
  qed
  with S1 show ?thesis by simp
```

```
qed

lemma op-sub-is-pr [prec]: (λ x y. x−y) ∈ PrimRec2
proof (rule pr-swap)
 show (λ x y. y − x) ∈ PrimRec2
  proof −
    have S1: PrimRecOp (λ x. x) (λ x y z.  y−(1::nat)) ∈ PrimRec2
    proof (rule pr-rec)
      show (λ x. x) ∈ PrimRec1 by (rule pr-id1-1)
    next
       from pred-is-pr pr-id3-2 show (λ x y z. y−(1::nat)) ∈ PrimRec3 by (rule pr-comp1-3)
    qed
    have (λ x y. y − x) = PrimRecOp (λ x. x) (λ x y z.  y−(1::nat)) (is - = ?f)
    proof −
      have ⋀ x y. (?f y x = x − y) by (induct-tac y, auto)
      thus ?thesis by (simp add: ext)
    qed
    with S1 show ?thesis by simp
  qed
qed

lemmas [prec] =
  const-is-pr [of 0] const-is-pr-2 [of 0] const-is-pr-3 [of 0]
  const-is-pr [of 1] const-is-pr-2 [of 1] const-is-pr-3 [of 1]
  const-is-pr [of 2] const-is-pr-2 [of 2] const-is-pr-3 [of 2]

definition
  sgn1 :: nat ⇒ nat where
  sgn1 x = (case x of 0 ⇒ 0 | Suc y ⇒ 1)

definition
  sgn2 :: nat ⇒ nat where
  sgn2 x ≡ (case x of 0 ⇒ 1 | Suc y ⇒ 0)

definition
  abs-of-diff :: nat ⇒ nat ⇒ nat where
  abs-of-diff = (λ x y. (x − y) + (y − x))

lemma [simp]: sgn1 0 = 0 by (simp add: sgn1-def)
lemma [simp]: sgn1 (Suc y) = 1 by (simp add: sgn1-def)
lemma [simp]: sgn2 0 = 1 by (simp add: sgn2-def)
lemma [simp]: sgn2 (Suc y) = 0 by (simp add: sgn2-def)
lemma [simp]: x ≠ 0 ⟹ sgn1 x = 1 by (simp add: sgn1-def, cases x, auto)
lemma [simp]: x ≠ 0 ⟹ sgn2 x = 0 by (simp add: sgn2-def, cases x, auto)

lemma sgn1-nz-impl-arg-pos: sgn1 x ≠ 0 ⟹ x > 0 by (cases x) auto
lemma sgn1-zero-impl-arg-zero: sgn1 x = 0 ⟹ x = 0 by (cases x) auto
lemma sgn2-nz-impl-arg-zero: sgn2 x ≠ 0 ⟹ x = 0 by (cases x) auto
```

```
lemma sgn2-zero-impl-arg-pos: sgn2 x = 0 ⟹ x > 0 by (cases x) auto

lemma sgn1-nz-eq-arg-pos: (sgn1 x ≠ 0) = (x > 0) by (cases x) auto
lemma sgn1-zero-eq-arg-zero: (sgn1 x = 0) = (x = 0) by (cases x) auto
lemma sgn2-nz-eq-arg-pos: (sgn2 x ≠ 0) = (x = 0) by (cases x) auto
lemma sgn2-zero-eq-arg-zero: (sgn2 x = 0) = (x > 0) by (cases x) auto

lemma sgn1-pos-eq-one: sgn1 x > 0 ⟹ sgn1 x = 1 by (cases x) auto
lemma sgn2-pos-eq-one: sgn2 x > 0 ⟹ sgn2 x = 1 by (cases x) auto

lemma sgn2-eq-1-sub-arg: sgn2 = (λ x. 1 − x)
proof (rule ext)
  fix x show sgn2 x = 1 − x by (cases x) auto
qed

lemma sgn1-eq-1-sub-sgn2: sgn1  = (λ x. 1 − (sgn2 x))
proof
  fix x show sgn1 x = 1 − sgn2 x
  proof −
    have 1− sgn2 x = 1 − (1 − x) by (simp add: sgn2-eq-1-sub-arg)
    then show ?thesis by (simp add: sgn1-def, cases x, auto)
  qed
qed

lemma sgn2-is-pr [prec]: sgn2 ∈ PrimRec1
proof −
  have (λ x. 1 − x) ∈ PrimRec1 by prec0
  thus ?thesis by (simp add: sgn2-eq-1-sub-arg)
qed

lemma sgn1-is-pr [prec]: sgn1 ∈ PrimRec1
proof −
  from sgn2-is-pr have (λ x. 1 − (sgn2 x)) ∈ PrimRec1 by prec0
  thus ?thesis by (simp add: sgn1-eq-1-sub-sgn2)
qed

lemma abs-of-diff-is-pr [prec]: abs-of-diff ∈ PrimRec2 unfolding abs-of-diff-def
by prec0

lemma abs-of-diff-eq: (abs-of-diff x y = 0) = (x = y) by (simp add: abs-of-diff-def,
arith)

lemma sf-is-pr [prec]: sf ∈ PrimRec1
proof −
  have S1: PrimRecOp1 0 (λ x y. y + x + 1) ∈ PrimRec1
  proof (rule pr-rec1)
    show (λ x y. y + x + 1) ∈ PrimRec2 by prec0
  qed
  have (λ x. sf x) = PrimRecOp1 0 (λ x y. y + x + 1) (is - = ?f)
```

```
  proof –
    have ⋀ x. (?f x = sf x)
    proof (induct-tac x)
      show ?f 0 = sf 0 by (simp add: sf-at-0)
    next
      fix x assume ?f x = sf x
      with sf-at-Suc show ?f (Suc x) = sf (Suc x)  by auto
    qed
    thus ?thesis by (simp add: ext)
  qed
  with S1 show ?thesis by simp
qed

lemma c-pair-is-pr [prec]: c-pair ∈ PrimRec2
proof –
  have c-pair = (λ x y. sf (x+y) + x) by (simp add: c-pair-def ext)
  moreover from sf-is-pr have (λ x y. sf (x+y) + x) ∈ PrimRec2 by prec0
  ultimately show ?thesis by (simp)
qed

lemma if-is-pr: ⟦ p ∈ PrimRec1; q1 ∈ PrimRec1; q2 ∈ PrimRec1⟧ ⟹ (λ x. if
(p x = 0) then (q1 x) else (q2 x)) ∈ PrimRec1
proof –
  have if-as-pr: (λ x. if (p x = 0) then (q1 x) else (q2 x)) = (λ x. (sgn2 (p x)) ∗
(q1 x) + (sgn1 (p x)) ∗ (q2 x))
  proof (rule ext)
   fix x show (if (p x = 0) then (q1 x) else (q2 x)) = (sgn2 (p x)) ∗ (q1 x) +
(sgn1 (p x)) ∗ (q2 x) (is ?left = ?right)
    proof cases
      assume A1: p x = 0
      then have S1: ?left = q1 x by simp
      from A1 have S2: ?right = q1 x by simp
      from S1 S2 show ?thesis by simp
    next
      assume A2: p x ≠ 0
      then have S3: p x > 0 by simp
      then show ?thesis by simp
    qed
  qed
  assume p ∈ PrimRec1 and q1 ∈ PrimRec1 and q2 ∈ PrimRec1
  then have (λ x. (sgn2 (p x)) ∗ (q1 x) + (sgn1 (p x)) ∗ (q2 x)) ∈ PrimRec1 by
prec0
  with if-as-pr show ?thesis by simp
qed

lemma if-eq-is-pr [prec]: ⟦ p1 ∈ PrimRec1; p2 ∈ PrimRec1; q1 ∈ PrimRec1; q2
∈ PrimRec1⟧ ⟹ (λ x. if (p1 x = p2 x) then (q1 x) else (q2 x)) ∈ PrimRec1
proof –
  have S1: (λ x. if (p1 x = p2 x) then (q1 x) else (q2 x)) = (λ x. if (abs-of-diff (p1
```

x) (*p2 x*) = *0*) *then* (*q1 x*) *else* (*q2 x*)) (**is** *?L* = *?R*) **by** (*simp add*: *abs-of-diff-eq*)
assume *A1*: *p1* ∈ *PrimRec1* **and** *A2*: *p2* ∈ *PrimRec1*
with *abs-of-diff-is-pr* **have** *S2*: (λ *x*. *abs-of-diff* (*p1 x*) (*p2 x*)) ∈ *PrimRec1* **by** *prec0*
assume *q1* ∈ *PrimRec1* **and** *q2* ∈ *PrimRec1*
with *S2* **have** *?R* ∈ *PrimRec1* **by** (*rule if-is-pr*)
with *S1* **show** *?thesis* **by** *simp*
qed

lemma *if-is-pr2* [*prec*]: ⟦ *p* ∈ *PrimRec2*; *q1* ∈ *PrimRec2*; *q2* ∈ *PrimRec2*⟧ ⟹ (λ *x y*. *if* (*p x y* = *0*) *then* (*q1 x y*) *else* (*q2 x y*)) ∈ *PrimRec2*
proof −
have *if-as-pr*: (λ *x y*. *if* (*p x y* = *0*) *then* (*q1 x y*) *else* (*q2 x y*)) = (λ *x y*. (*sgn2* (*p x y*)) ∗ (*q1 x y*) + (*sgn1* (*p x y*)) ∗ (*q2 x y*))
proof (*rule ext*, *rule ext*)
fix *x* **fix** *y* **show** (*if* (*p x y* = *0*) *then* (*q1 x y*) *else* (*q2 x y*)) = (*sgn2* (*p x y*)) ∗ (*q1 x y*) + (*sgn1* (*p x y*)) ∗ (*q2 x y*) (**is** *?left* = *?right*)
proof *cases*
assume *A1*: *p x y* = *0*
then have *S1*: *?left* = *q1 x y* **by** *simp*
from *A1* **have** *S2*: *?right* = *q1 x y* **by** *simp*
from *S1 S2* **show** *?thesis* **by** *simp*
next
assume *A2*: *p x y* ≠ *0*
then have *S3*: *p x y* > *0* **by** *simp*
then show *?thesis* **by** *simp*
qed
qed
assume *p* ∈ *PrimRec2* **and** *q1* ∈ *PrimRec2* **and** *q2* ∈ *PrimRec2*
then have (λ *x y*. (*sgn2* (*p x y*)) ∗ (*q1 x y*) + (*sgn1* (*p x y*)) ∗ (*q2 x y*)) ∈ *PrimRec2* **by** *prec0*
with *if-as-pr* **show** *?thesis* **by** *simp*
qed

lemma *if-eq-is-pr2*: ⟦ *p1* ∈ *PrimRec2*; *p2* ∈ *PrimRec2*; *q1* ∈ *PrimRec2*; *q2* ∈ *PrimRec2*⟧ ⟹ (λ *x y*. *if* (*p1 x y* = *p2 x y*) *then* (*q1 x y*) *else* (*q2 x y*)) ∈ *PrimRec2*
proof −
have *S1*: (λ *x y*. *if* (*p1 x y* = *p2 x y*) *then* (*q1 x y*) *else* (*q2 x y*)) = (λ *x y*. *if* (*abs-of-diff* (*p1 x y*) (*p2 x y*) = *0*) *then* (*q1 x y*) *else* (*q2 x y*)) (**is** *?L* = *?R*) **by** (*simp add*: *abs-of-diff-eq*)
assume *A1*: *p1* ∈ *PrimRec2* **and** *A2*: *p2* ∈ *PrimRec2*
with *abs-of-diff-is-pr* **have** *S2*: (λ *x y*. *abs-of-diff* (*p1 x y*) (*p2 x y*)) ∈ *PrimRec2* **by** *prec0*
assume *q1* ∈ *PrimRec2* **and** *q2* ∈ *PrimRec2*
with *S2* **have** *?R* ∈ *PrimRec2* **by** (*rule if-is-pr2*)
with *S1* **show** *?thesis* **by** *simp*
qed

lemma *if-is-pr3* [*prec*]: ⟦ *p* ∈ *PrimRec3*; *q1* ∈ *PrimRec3*; *q2* ∈ *PrimRec3*⟧ ⟹ (λ

```
x y z. if (p x y z = 0) then (q1 x y z) else (q2 x y z)) ∈ PrimRec3
proof −
  have if-as-pr: (λ x y z. if (p x y z = 0) then (q1 x y z) else (q2 x y z)) = (λ x y
z. (sgn2 (p x y z)) ∗ (q1 x y z) + (sgn1 (p x y z)) ∗ (q2 x y z))
  proof (rule ext, rule ext, rule ext)
    fix x fix y fix z show (if (p x y z = 0) then (q1 x y z) else (q2 x y z)) = (sgn2
(p x y z)) ∗ (q1 x y z) + (sgn1 (p x y z)) ∗ (q2 x y z) (is ?left = ?right)
     proof cases
       assume A1: p x y z = 0
       then have S1: ?left = q1 x y z by simp
       from A1 have S2: ?right = q1 x y z by simp
       from S1 S2 show ?thesis by simp
     next
       assume A2: p x y z ≠ 0
       then have S3: p x y z > 0 by simp
       then show ?thesis by simp
     qed
  qed
  assume p ∈ PrimRec3 and q1 ∈ PrimRec3 and q2 ∈ PrimRec3
  then have (λ x y z. (sgn2 (p x y z)) ∗ (q1 x y z) + (sgn1 (p x y z)) ∗ (q2 x y
z)) ∈ PrimRec3
     by prec0
  with if-as-pr show ?thesis by simp
qed

lemma if-eq-is-pr3: ⟦ p1 ∈ PrimRec3; p2 ∈ PrimRec3; q1 ∈ PrimRec3; q2 ∈
PrimRec3⟧ =⇒ (λ x y z. if (p1 x y z = p2 x y z) then (q1 x y z) else (q2 x y z))
∈ PrimRec3
proof −
  have S1: (λ x y z. if (p1 x y z = p2 x y z) then (q1 x y z) else (q2 x y z)) = (λ
x y z. if (abs-of-diff (p1 x y z) (p2 x y z) = 0) then (q1 x y z) else (q2 x y z)) (is
?L = ?R) by (simp add: abs-of-diff-eq)
  assume A1: p1 ∈ PrimRec3 and A2: p2 ∈ PrimRec3
   with abs-of-diff-is-pr have S2: (λ x y z. abs-of-diff (p1 x y z) (p2 x y z)) ∈
PrimRec3
     by prec0
  assume q1 ∈ PrimRec3 and q2 ∈ PrimRec3
  with S2 have ?R ∈ PrimRec3 by (rule if-is-pr3)
  with S1 show ?thesis by simp
qed

ML ‹
fun get-if-by-index 1 = @{thm if-eq-is-pr}
  | get-if-by-index 2 = @{thm if-eq-is-pr2}
  | get-if-by-index 3 = @{thm if-eq-is-pr3}
  | get-if-by-index - = raise BadArgument

fun if-comp-tac ctxt = SUBGOAL (fn (t, i) =>
  let
```

```
    val t = extract-trueprop-arg (Logic.strip-imp-concl t)
    val (t1, t2) = extract-set-args t
    val n2 =
      let
        val Const(s, -) = t2
      in
        get-num-by-set s
      end
    val (name, -, n1) = extract-free-arg t1
  in
    if name = @{const-name If} then
      resolve-tac ctxt [get-if-by-index n2] i
    else
      let
        val comp = get-comp-by-indexes (n1, n2)
      in
        Rule-Insts.res-inst-tac ctxt
          [(((f, 0), Position.none), Variable.revert-fixed ctxt name)] [] comp i
      end
  end
  handle BadArgument => no-tac)

fun prec-tac ctxt facts i =
  Method.insert-tac ctxt facts i THEN
  REPEAT (resolve-tac ctxt [@{thm const-is-pr}, @{thm const-is-pr-2}, @{thm
const-is-pr-3}] i ORELSE
    assume-tac ctxt i ORELSE if-comp-tac ctxt i)
›
```

method-setup *prec* = ‹
```
  Attrib.thms >> (fn ths => fn ctxt => Method.METHOD (fn facts =>
   HEADGOAL (prec-tac ctxt (facts @ Named-Theorems.get ctxt @{named-theorems
prec}))))
```
› *apply primitive recursive functions*

2.2 Bounded least operator

definition
b-least :: $(nat \Rightarrow nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat)$ **where**
b-least f x $\equiv$ $(Least\ (\%y.\ y = x \vee (y < x \wedge (f\ x\ y) \neq 0)))$

definition
b-least2 :: $(nat \Rightarrow nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat \Rightarrow nat)$ **where**
b-least2 f x y $\equiv$ $(Least\ (\%z.\ z = y \vee (z < y \wedge (f\ x\ z) \neq 0)))$

lemma *b-least-aux1*: *b-least f x* $= x \vee$ (*b-least f x* $< x \wedge (f\ x$ (*b-least f x*)) $\neq 0$)
proof −
let $?P = \%y.\ y = x \vee (y < x \wedge (f\ x\ y) \neq 0)$
have $?P\ x$ **by** *simp*

then have *?P* (*Least ?P*) **by** (*rule LeastI*)
thus *?thesis* **by** (*simp add*: *b-least-def*)
qed

lemma *b-least-le-arg*: *b-least f x* ≤ *x*
proof −
have *b-least f x* = *x* ∨ (*b-least f x* < *x* ∧ (*f x* (*b-least f x*)) ≠ *0*) **by** (*rule b-least-aux1*)
from *this* **show** *?thesis* **by** (*arith*)
qed

lemma *less-b-least-impl-zero*: *y* < *b-least f x* ⟹ *f x y* = *0*
proof −
assume *A1*: *y* < *b-least f x* (**is** - < *?b*)
have *b-least f x* ≤ *x* **by** (*rule b-least-le-arg*)
with *A1* **have** *S1*: *y* < *x* **by** *simp*
with *A1* **have** *y* < (*Least* (%*y*. *y* = *x* ∨ (*y* < *x* ∧ (*f x y*) ≠ *0*))) **by** (*simp add*: *b-least-def*)
then have ¬ (*y* = *x* ∨ (*y* < *x* ∧ (*f x y*) ≠ *0*)) **by** (*rule not-less-Least*)
with *S1* **show** *?thesis* **by** *simp*
qed

lemma *nz-impl-b-least-le*: (*f x y*) ≠ *0* ⟹ (*b-least f x*) ≤ *y*
proof (*rule ccontr*)
assume *A1*: *f x y* ≠ *0*
assume ¬ *b-least f x* ≤ *y*
then have *y* < *b-least f x* **by** *simp*
with *A1* **show** *False* **by** (*simp add*: *less-b-least-impl-zero*)
qed

lemma *b-least-less-impl-nz*: *b-least f x* < *x* ⟹ *f x* (*b-least f x*) ≠ *0*
proof −
assume *A1*: *b-least f x* < *x*
have *b-least f x* = *x* ∨ (*b-least f x* < *x* ∧ (*f x* (*b-least f x*)) ≠ *0*) **by** (*rule b-least-aux1*)
from *A1 this* **show** *?thesis* **by** *simp*
qed

lemma *b-least-less-impl-eq*: *b-least f x* < *x* ⟹ (*b-least f x*) = (*Least* (%*y*. (*f x y*) ≠ *0*))
proof −
assume *A1*: *b-least f x* < *x* (**is** *?b* < -)
let *?B* = (*Least* (%*y*. (*f x y*) ≠ *0*))
from *A1* **have** *S1*: *f x ?b* ≠ *0* **by** (*rule b-least-less-impl-nz*)
from *S1* **have** *S2*: *?B* ≤ *?b* **by** (*rule Least-le*)
from *S1* **have** *S3*: *f x ?B* ≠ *0* **by** (*rule LeastI*)
from *S3* **have** *S4*: *?b* ≤ *?B* **by** (*rule nz-impl-b-least-le*)
from *S2 S4* **show** *?thesis* **by** *simp*
qed

```
lemma less-b-least-impl-zero2: ⟦y < x; b-least f x = x⟧ ⟹ f x y = 0 by (simp
add: less-b-least-impl-zero)

lemma nz-impl-b-least-less: ⟦y<x; (f x y) ≠ 0⟧ ⟹ (b-least f x) < x
proof –
  assume A1: y < x
  assume f x y ≠ 0
  then have b-least f x ≤ y by (rule nz-impl-b-least-le)
  with A1 show ?thesis by simp
qed

lemma b-least-aux2: ⟦y<x; (f x y) ≠ 0⟧ ⟹ (b-least f x) = (Least (%y. (f x y) ≠
0))
proof –
  assume A1: y < x and A2: f x y ≠ 0
  from A1 A2 have S1: b-least f x < x by (rule nz-impl-b-least-less)
  thus ?thesis by (rule b-least-less-impl-eq)
qed

lemma b-least2-aux1: b-least2 f x y = y ∨ (b-least2 f x y < y ∧ (f x (b-least2 f x
y)) ≠ 0)
proof –
  let ?P = %z. z = y ∨ (z < y ∧ (f x z) ≠ 0)
  have ?P y by simp
  then have ?P (Least ?P) by (rule LeastI)
  thus ?thesis by (simp add: b-least2-def)
qed

lemma b-least2-le-arg: b-least2 f x y ≤ y
proof –
  let ?B = b-least2 f x y
  have ?B = y ∨ (?B < y ∧ (f x ?B) ≠ 0) by (rule b-least2-aux1)
  from this show ?thesis by (arith)
qed

lemma less-b-least2-impl-zero: z < b-least2 f x y ⟹ f x z = 0
proof –
  assume A1: z < b-least2 f x y (is - < ?b)
  have b-least2 f x y ≤ y by (rule b-least2-le-arg)
  with A1 have S1: z < y by simp
  with A1 have  z < (Least (%z. z = y ∨ (z < y ∧ (f x z) ≠ 0))) by (simp add:
b-least2-def)
  then have ¬ (z = y ∨ (z < y ∧ (f x z) ≠ 0)) by (rule not-less-Least)
  with S1 show ?thesis by simp
qed

lemma nz-impl-b-least2-le: (f x z) ≠ 0 ⟹ (b-least2 f x y) ≤ z
proof –
```

```
  assume A1: f x z ≠ 0
  have S1: z < b-least2 f x y ⟹ f x z = 0 by (rule less-b-least2-impl-zero)
  from A1 S1 show ?thesis by arith
qed

lemma b-least2-less-impl-nz: b-least2 f x y < y ⟹ f x (b-least2 f x y) ≠ 0
proof −
  assume A1: b-least2 f x y < y
  have b-least2 f x y = y ∨ (b-least2 f x y < y ∧ (f x (b-least2 f x y)) ≠ 0) by
(rule b-least2-aux1)
  with A1 show ?thesis by simp
qed

lemma b-least2-less-impl-eq: b-least2 f x y < y ⟹ (b-least2 f x y) = (Least (%z.
(f x z) ≠ 0))
proof −
  assume A1: b-least2 f x y < y (is ?b < -)
  let ?B = (Least (%z. (f x z) ≠ 0))
  from A1 have S1: f x ?b ≠ 0 by (rule b-least2-less-impl-nz)
  from S1 have S2: ?B ≤ ?b by (rule Least-le)
  from S1 have S3: f x ?B ≠ 0 by (rule LeastI)
  from S3 have S4: ?b ≤ ?B by (rule nz-impl-b-least2-le)
  from S2 S4 show ?thesis by simp
qed

lemma less-b-least2-impl-zero2: ⟦z<y; b-least2 f x y = y⟧ ⟹ f x z = 0
proof −
  assume z < y and b-least2 f x y = y
  hence z < b-least2 f x y by simp
  thus ?thesis by (rule less-b-least2-impl-zero)
qed

lemma nz-b-least2-impl-less: ⟦z<y; (f x z) ≠ 0⟧ ⟹ (b-least2 f x y) < y
proof (rule ccontr)
  assume A1: z < y
  assume A2: f x z ≠ 0
  assume ¬ (b-least2 f x y) < y then have A3: y ≤ (b-least2 f x y) by simp
  have b-least2 f x y ≤ y by (rule b-least2-le-arg)
  with A3 have b-least2 f x y = y by simp
  with A1 have f x z = 0 by (rule less-b-least2-impl-zero2)
  with A2 show False by simp
qed

lemma b-least2-less-impl-eq2: ⟦z < y; (f x z) ≠ 0⟧ ⟹ (b-least2 f x y) = (Least
(%z. (f x z) ≠ 0))
proof −
  assume A1: z < y and A2: f x z ≠ 0
  from A1 A2 have S1: b-least2 f x y < y by (rule nz-b-least2-impl-less)
  thus ?thesis by (rule b-least2-less-impl-eq)
```

```
qed

lemma b-least2-aux2: b-least2 f x y < y ⟹ b-least2 f x (Suc y) = b-least2 f x y
proof −
  let ?B = b-least2 f x y
  assume A1: ?B < y
  from A1 have S1: f x ?B ≠ 0 by (rule b-least2-less-impl-nz)
  from S1 have S2: b-least2 f x (Suc y) ≤ ?B by (simp add: nz-impl-b-least2-le)
  from A1 S2 have S3: b-least2 f x (Suc y) < Suc y by (simp)
  from S3 have S4: f x (b-least2 f x (Suc y)) ≠ 0 by (rule b-least2-less-impl-nz)
  from S4 have S5: ?B ≤ b-least2 f x (Suc y) by (rule nz-impl-b-least2-le)
  from S2 S5 show ?thesis by simp
qed

lemma b-least2-aux3: ⟦ b-least2 f x y = y; f x y ≠ 0⟧ ⟹ b-least2 f x (Suc y) = y
proof −
  assume A1: b-least2 f x y =y
  assume A2: f x y ≠ 0
  from A2 have S1: b-least2 f x (Suc y) ≤ y by (rule nz-impl-b-least2-le)
  have S2: b-least2 f x (Suc y) < y ⟹ False
  proof −
    assume A2-1: b-least2 f x (Suc y) < y (is ?z < -)
    from A2-1 have S2-1: ?z < Suc y by simp
    from S2-1 have S2-2: f x ?z ≠ 0 by (rule b-least2-less-impl-nz)
    from A2-1 S2-2 have S2-3: b-least2 f x y < y by (rule nz-b-least2-impl-less)
    from S2-3 A1 show ?thesis by simp
  qed
  from S2 have S3: ¬ (b-least2 f x (Suc y) < y) by auto
  from S1 S3 show ?thesis by simp
qed

lemma b-least2-mono: y1 ≤ y2 ⟹ b-least2 f x y1 ≤ b-least2 f x y2
proof (rule ccontr)
  assume A1: y1 ≤ y2
  let ?b1 = b-least2 f x y1 and ?b2 = b-least2 f x y2
  assume ¬ ?b1 ≤ ?b2 then have A2: ?b2 < ?b1 by simp
  have S1: ?b1 ≤ y1 by (rule b-least2-le-arg)
  have S2: ?b2 ≤ y2 by (rule b-least2-le-arg)
  from A1 A2 S1 S2 have S3: ?b2 < y2 by simp
  then have S4: f x ?b2 ≠ 0 by (rule b-least2-less-impl-nz)
  from A2 have S5: f x ?b2 = 0 by (rule less-b-least2-impl-zero)
  from S4 S5 show False by simp
qed

lemma b-least2-aux4: ⟦ b-least2 f x y = y; f x y = 0⟧ ⟹ b-least2 f x (Suc y) =
Suc y
proof −
  assume A1: b-least2 f x y = y
  assume A2: f x y = 0
```

```
  have S1: b-least2 f x (Suc y) ≤ Suc y by (rule b-least2-le-arg)
  have S2: y ≤ b-least2 f x (Suc y)
  proof −
    have y ≤ Suc y by simp
    then have b-least2 f x y ≤ b-least2 f x (Suc y) by (rule b-least2-mono)
    with A1 show ?thesis by simp
  qed
  from S1 S2 have b-least2 f x (Suc y) =y ∨ b-least2 f x (Suc y) = Suc y by
arith
  moreover
  {
    assume A3: b-least2 f x (Suc y) = y
    have f x y ≠ 0
    proof −
      have y < Suc y by simp
      with A3 have b-least2 f x (Suc y) < Suc y by simp
     from this have f x (b-least2 f x (Suc y)) ≠ 0 by (simp add: b-least2-less-impl-nz)
      with A3 show f x y ≠ 0 by simp
    qed
    with A2 have ?thesis by simp
  }
  moreover
  {
    assume b-least2 f x (Suc y) = Suc y
    then have ?thesis by simp
  }
  ultimately show ?thesis by blast
qed

lemma b-least2-at-zero: b-least2 f x 0 = 0
proof −
  have S1: b-least2 f x 0 ≤ 0 by (rule b-least2-le-arg)
  from S1 show ?thesis by auto
qed

theorem pr-b-least2: f ∈ PrimRec2 ⟹ b-least2 f ∈ PrimRec2
proof −
  define loc-Op1 where loc-Op1 = (λ (f::nat ⇒ nat ⇒ nat) x y z. (sgn1 (z −
y)) ∗ y + (sgn2 (z − y))∗((sgn1 (f x z))∗z + (sgn2 (f x z))∗(Suc z)))
  define loc-Op2 where loc-Op2 = (λ f. PrimRecOp-last (λ x. 0) (loc-Op1 f))
  have loc-op2-lm-1: ⋀ f x y. loc-Op2 f x y < y ⟹ loc-Op2 f x (Suc y) = loc-Op2
f x y
  proof −
    fix f x y
    let ?b = loc-Op2 f x y
    have S1: loc-Op2 f x (Suc y) = (loc-Op1 f) x ?b y by (simp add: loc-Op2-def)
    assume ?b < y
    then have y − ?b > 0 by simp
    then have loc-Op1 f x ?b y = ?b by (simp add: loc-Op1-def)
```

```
    with S1 show loc-Op2 f x y < y ⟹ loc-Op2 f x (Suc y) = loc-Op2 f x y by simp
  qed
  have loc-op2-lm-2: ⋀ f x y. ⟦¬(loc-Op2 f x y < y); f x y ≠ 0⟧ ⟹ loc-Op2 f x (Suc y) = y
  proof −
    fix f x y
    let ?b = loc-Op2 f x y and ?h = loc-Op1 f
    have S1: loc-Op2 f x (Suc y) = ?h x ?b y by (simp add: loc-Op2-def)
    assume ¬(?b < y)
    then have S2: y − ?b = 0 by simp
    assume f x y ≠ 0
    with S2 have ?h x ?b y = y by (simp add: loc-Op1-def)
    with S1 show loc-Op2 f x (Suc y) = y by simp
  qed
  have loc-op2-lm-3: ⋀ f x y. ⟦¬(loc-Op2 f x y < y); f x y = 0⟧ ⟹ loc-Op2 f x (Suc y) = Suc y
  proof −
    fix f x y
    let ?b = loc-Op2 f x y and ?h = loc-Op1 f
    have S1: loc-Op2 f x (Suc y) = ?h x ?b y by (simp add: loc-Op2-def)
    assume ¬(?b < y)
    then have S2: y − ?b = 0 by simp
    assume f x y = 0
    with S2 have ?h x ?b y = Suc y by (simp add: loc-Op1-def)
    with S1 show loc-Op2 f x (Suc y) = Suc y by simp
  qed
  have Op2-eq-b-least2-at-point: ⋀ f x y. loc-Op2 f x y = b-least2 f x y
  proof − fix f x show ⋀ y. loc-Op2 f x y = b-least2 f x y
  proof (induct-tac y)
   show loc-Op2 f x 0 = b-least2 f x 0 by (simp add: loc-Op2-def b-least2-at-zero)
  next
    fix y
    assume A1: loc-Op2 f x y = b-least2 f x y
    then show loc-Op2 f x (Suc y) = b-least2 f x (Suc y)
    proof cases
      assume A2: loc-Op2 f x y < y
      then have S1: loc-Op2 f x (Suc y) = loc-Op2 f x y by (rule loc-op2-lm-1)
      from A1 A2 have b-least2 f x y < y by simp
      then have S2: b-least2 f x (Suc y) = b-least2 f x y by (rule b-least2-aux2)
      from A1 S1 S2 show ?thesis by simp
    next
      assume A3: ¬ loc-Op2 f x y < y
      have A3': b-least2 f x y = y
      proof −
        have b-least2 f x y ≤ y by (rule b-least2-le-arg)
        from A1 A3 this show ?thesis by simp
      qed
      then show ?thesis
```

```
      proof cases
        assume A4: f x y ≠ 0
        with A3 have S3: loc-Op2 f x (Suc y) = y by (rule loc-op2-lm-2)
        from A3' A4 have S4: b-least2 f x (Suc y) = y by (rule b-least2-aux3)
        from S3 S4 show ?thesis by simp
      next
        assume ¬ f x y ≠  0
        then have A5: f x y = 0 by simp
        with A3 have S5: loc-Op2 f x (Suc y) = Suc y by (rule loc-op2-lm-3)
       from A3' A5 have S6: b-least2 f x (Suc y) = Suc y by (rule b-least2-aux4)
        from S5 S6 show ?thesis by simp
      qed
    qed
  qed
  qed
  have Op2-eq-b-least2: loc-Op2 = b-least2 by (simp add: Op2-eq-b-least2-at-point
ext)
  assume A1: f ∈ PrimRec2
  have pr-loc-Op2: loc-Op2 f ∈ PrimRec2
  proof −
    from A1 have S1: loc-Op1 f ∈ PrimRec3 by (simp add: loc-Op1-def, prec)
    from pr-zero S1 have S2: PrimRecOp-last (λ x. 0) (loc-Op1 f) ∈ PrimRec2
by (rule pr-rec-last)
    from this show ?thesis by (simp add: loc-Op2-def)
  qed
  from Op2-eq-b-least2 this show b-least2 f ∈ PrimRec2 by simp
qed
```

```
lemma b-least-def1: b-least f = (λ x. b-least2 f x x) by (simp add: b-least2-def
b-least-def ext)
```

```
theorem pr-b-least: f ∈ PrimRec2 ⟹ b-least f ∈ PrimRec1
proof −
  assume f ∈ PrimRec2
  then have b-least2 f ∈ PrimRec2 by (rule pr-b-least2)
  from this pr-id1-1 pr-id1-1 have (λ x. b-least2 f x x) ∈ PrimRec1 by (rule
pr-comp2-1)
  then show ?thesis by (simp add: b-least-def1)
qed
```

2.3 Examples

```
theorem c-sum-as-b-least: c-sum = (λ u. b-least2 (λ u z. (sgn1 (sf(z+1) − u)))
u (Suc u))
proof (rule ext)
  fix u show c-sum u = b-least2 (λ u z. (sgn1 (sf(z+1) − u))) u (Suc u)
  proof −
    have lm-1: (λ x y. (sgn1 (sf(y+1) − x) ≠ 0)) = (λ x y. (x < sf(y+1)))
    proof (rule ext, rule ext)
```

```
      fix x y show (sgn1 (sf(y+1) − x) ≠ 0) = (x < sf(y+1))
      proof −
      have (sgn1 (sf(y+1) − x) ≠ 0) = (sf(y+1) − x > 0) by (rule sgn1-nz-eq-arg-pos)
        thus (sgn1 (sf(y+1) − x) ≠ 0) = (x < sf(y+1)) by auto
      qed
    qed
    let ?f = λ u z. (sgn1 (sf(z+1) − u))
    have S1: ?f u u ≠ 0
    proof −
      have S1-1: u+1 ≤ sf(u+1) by (rule arg-le-sf)
      have S1-2: u < u+1 by simp
      from S1-1 S1-2 have S1-3: u < sf(u+1) by simp
      from S1-3 have S1-4: sf(u+1) − u > 0 by simp
      from S1-4 have S1-5: sgn1 (sf(u+1)−u) = 1 by simp
      from S1-5 show ?thesis by simp
    qed
    have S3: u < Suc u by simp
    from S3 S1 have S4: b-least2 ?f u (Suc u) = (Least (%z. (?f u z) ≠ 0)) by (rule b-least2-less-impl-eq2)
    let ?P = λ u z. ?f u z ≠ 0
    let ?Q = λ u z. u < sf(z+1)
    from lm-1 have S6: ?P = ?Q by simp
    from S6 have S7: (%z. ?P u z) = (%z. ?Q u z) by (rule fun-cong)
    from S7 have S8: (Least (%z. ?P u z)) = (Least (%z. ?Q u z)) by auto
    from S4 S8 have S9: b-least2 ?f u (Suc u) = (Least (%z. u < sf(z+1))) by (rule trans)
    thus ?thesis by (simp add: c-sum-def)
  qed
qed

theorem c-sum-is-pr: c-sum ∈ PrimRec1
proof −
  let ?f = λ u z. (sgn1 (sf(z+1) − u))
  have S1: (λ u z. sgn1 ((sf(z+1) − u))) ∈ PrimRec2 by prec
  define g where g = b-least2 ?f
  from g-def S1 have g ∈ PrimRec2 by (simp add: pr-b-least2)
  then have S2: (λ u. g u (Suc u)) ∈ PrimRec1 by prec
  from g-def have c-sum = (λ u. g u (Suc u)) by (simp add: c-sum-as-b-least ext)
  with S2 show ?thesis by simp
qed

theorem c-fst-is-pr [prec]: c-fst ∈ PrimRec1
proof −
  have S1: (λ u. c-fst u) = (λ u. (u − sf (c-sum u))) by (simp add: c-fst-def ext)
  from c-sum-is-pr have (λ u. (u − sf (c-sum u))) ∈ PrimRec1 by prec
  with S1 show ?thesis by simp
qed

theorem c-snd-is-pr [prec]: c-snd ∈ PrimRec1
```

proof −
  have *S1*: *c-snd* = (λ *u*. (*c-sum u*) − (*c-fst u*)) **by** (*simp add*: *c-snd-def ext*)
  from *c-sum-is-pr c-fst-is-pr* **have** *S2*: (λ *u*. (*c-sum u*) − (*c-fst u*)) ∈ *PrimRec1* **by** *prec*
  from *S1 this* **show** *?thesis* **by** *simp*
qed

theorem *pr-1-to-2*: *f* ∈ *PrimRec1* ⟹ (λ *x y*. *f* (*c-pair x y*)) ∈ *PrimRec2* **by** *prec*

theorem *pr-2-to-1*: *f* ∈ *PrimRec2* ⟹ (λ *z*. *f* (*c-fst z*) (*c-snd z*)) ∈ *PrimRec1* **by** *prec*

definition *pr-conv-1-to-2* = (λ *f x y*. *f* (*c-pair x y*))
definition *pr-conv-1-to-3* = (λ *f x y z*. *f* (*c-pair* (*c-pair x y*) *z*))
definition *pr-conv-2-to-1* = (λ *f x*. *f* (*c-fst x*) (*c-snd x*))
definition *pr-conv-3-to-1* = (λ *f x*. *f* (*c-fst* (*c-fst x*)) (*c-snd* (*c-fst x*)) (*c-snd x*))
definition *pr-conv-3-to-2* = (λ *f*. *pr-conv-1-to-2* (*pr-conv-3-to-1 f*))
definition *pr-conv-2-to-3* = (λ *f*. *pr-conv-1-to-3* (*pr-conv-2-to-1 f*))

lemma [*simp*]: *pr-conv-1-to-2* (*pr-conv-2-to-1 f*) = *f* **by**(*simp add*: *pr-conv-1-to-2-def pr-conv-2-to-1-def*)
lemma [*simp*]: *pr-conv-2-to-1* (*pr-conv-1-to-2 f*) = *f* **by**(*simp add*: *pr-conv-1-to-2-def pr-conv-2-to-1-def*)
lemma [*simp*]: *pr-conv-1-to-3* (*pr-conv-3-to-1 f*) = *f* **by**(*simp add*: *pr-conv-1-to-3-def pr-conv-3-to-1-def*)
lemma [*simp*]: *pr-conv-3-to-1* (*pr-conv-1-to-3 f*) = *f* **by**(*simp add*: *pr-conv-1-to-3-def pr-conv-3-to-1-def*)
lemma [*simp*]: *pr-conv-3-to-2* (*pr-conv-2-to-3 f*) = *f* **by**(*simp add*: *pr-conv-3-to-2-def pr-conv-2-to-3-def*)
lemma [*simp*]: *pr-conv-2-to-3* (*pr-conv-3-to-2 f*) = *f* **by**(*simp add*: *pr-conv-3-to-2-def pr-conv-2-to-3-def*)

lemma *pr-conv-1-to-2-lm*: *f* ∈ *PrimRec1* ⟹ *pr-conv-1-to-2 f* ∈ *PrimRec2* **by** (*simp add*: *pr-conv-1-to-2-def*, *prec*)
lemma *pr-conv-1-to-3-lm*: *f* ∈ *PrimRec1* ⟹ *pr-conv-1-to-3 f* ∈ *PrimRec3* **by** (*simp add*: *pr-conv-1-to-3-def*, *prec*)
lemma *pr-conv-2-to-1-lm*: *f* ∈ *PrimRec2* ⟹ *pr-conv-2-to-1 f* ∈ *PrimRec1* **by** (*simp add*: *pr-conv-2-to-1-def*, *prec*)
lemma *pr-conv-3-to-1-lm*: *f* ∈ *PrimRec3* ⟹ *pr-conv-3-to-1 f* ∈ *PrimRec1* **by** (*simp add*: *pr-conv-3-to-1-def*, *prec*)
lemma *pr-conv-3-to-2-lm*: *f* ∈ *PrimRec3* ⟹ *pr-conv-3-to-2 f* ∈ *PrimRec2*
proof −
  assume *f* ∈ *PrimRec3*
  then have *pr-conv-3-to-1 f* ∈ *PrimRec1* **by** (*rule pr-conv-3-to-1-lm*)
  thus *?thesis* **by** (*simp add*: *pr-conv-3-to-2-def pr-conv-1-to-2-lm*)
qed
lemma *pr-conv-2-to-3-lm*: *f* ∈ *PrimRec2* ⟹ *pr-conv-2-to-3 f* ∈ *PrimRec3*
proof −
  assume *f* ∈ *PrimRec2*

```
  then have pr-conv-2-to-1 f ∈ PrimRec1 by (rule pr-conv-2-to-1-lm)
  thus ?thesis by (simp add: pr-conv-2-to-3-def pr-conv-1-to-3-lm)
qed

theorem b-least2-scheme: ⟦ f ∈ PrimRec2; g ∈ PrimRec1; ∀ x. h x < g x; ∀ x. f
x (h x) ≠ 0; ∀ z x. z < h x ⟶ f x z = 0 ⟧ ⟹
                              h ∈ PrimRec1
proof −
  assume f-is-pr: f ∈ PrimRec2
  assume g-is-pr: g ∈ PrimRec1
  assume h-lt-g: ∀ x. h x < g x
  assume f-at-h-nz: ∀ x. f x (h x) ≠ 0
  assume h-is-min: ∀ z x. z < h x ⟶ f x z = 0
  have h-def: h = (λ x. b-least2 f x (g x))
  proof
    fix x show h x = b-least2 f x (g x)
    proof −
    from f-at-h-nz have S1: b-least2 f x (g x) ≤ h x by (simp add: nz-impl-b-least2-le)
      from h-lt-g have h x < g x by auto
      with S1 have b-least2 f x (g x) < g x by simp
      then have S2: f x (b-least2 f x (g x)) ≠ 0 by (rule b-least2-less-impl-nz)
      have S3: h x ≤ b-least2 f x (g x)
      proof (rule ccontr)
        assume ¬ h x ≤ b-least2 f x (g x) then have b-least2 f x (g x) < h x by
auto
        with h-is-min have f x (b-least2 f x (g x)) = 0 by simp
        with S2 show False by auto
      qed
      from S1 S3 show ?thesis by auto
    qed
  qed
  define f1 where f1 = b-least2 f
  from f-is-pr f1-def have f1-is-pr: f1 ∈ PrimRec2 by (simp add: pr-b-least2)
  with g-is-pr have (λ x. f1 x (g x)) ∈ PrimRec1 by prec
  with h-def f1-def show h ∈ PrimRec1 by auto
qed

theorem b-least2-scheme2: ⟦ f ∈ PrimRec3; g ∈ PrimRec2; ∀ x y. h x y < g x y;
∀ x y. f x y (h x y) ≠ 0;
                              ∀ z x y. z < h x y ⟶ f x y z = 0 ⟧ ⟹
                              h ∈ PrimRec2
proof −
  assume f-is-pr: f ∈ PrimRec3
  assume g-is-pr: g ∈ PrimRec2
  assume h-lt-g: ∀ x y. h x y < g x y
  assume f-at-h-nz: ∀ x y. f x y (h x y) ≠ 0
  assume h-is-min: ∀ z x y. z < h x y ⟶ f x y z = 0
  define f1 where f1 = pr-conv-3-to-2 f
  define g1 where g1 = pr-conv-2-to-1 g
```

```
  define h1 where h1 = pr-conv-2-to-1 h
  from f-is-pr f1-def have f1-is-pr: f1 ∈ PrimRec2 by (simp add: pr-conv-3-to-2-lm)
  from g-is-pr g1-def have g1-is-pr: g1 ∈ PrimRec1 by (simp add: pr-conv-2-to-1-lm)
  from h-lt-g h1-def g1-def have h1-lt-g1: ∀ x. h1 x < g1 x by (simp add: pr-conv-2-to-1-def)
  from f-at-h-nz f1-def h1-def have f1-at-h1-nz: ∀ x. f1 x (h1 x) ≠ 0 by (simp add: pr-conv-2-to-1-def pr-conv-3-to-2-def pr-conv-3-to-1-def pr-conv-1-to-2-def)
  from h-is-min f1-def h1-def have h1-is-min: ∀ z x. z < h1 x ⟶ f1 x z = 0 by (simp add: pr-conv-2-to-1-def pr-conv-3-to-2-def pr-conv-3-to-1-def pr-conv-1-to-2-def)
  from f1-is-pr g1-is-pr h1-lt-g1 f1-at-h1-nz h1-is-min have h1-is-pr: h1 ∈ PrimRec1 by (rule b-least2-scheme)
  from h1-def have h = pr-conv-1-to-2 h1 by simp
  with h1-is-pr show h ∈ PrimRec2 by (simp add: pr-conv-1-to-2-lm)
qed

theorem div-is-pr: (λ a b. a div b) ∈ PrimRec2
proof −
  define f where f a b z = (sgn1 b) ∗ (sgn1 (b∗(z+1)−a)) + (sgn2 b)∗(sgn2 z) for a b z
  have f-is-pr: f ∈ PrimRec3 unfolding f-def by prec
  define h where h a b = a div b for a b :: nat
  define g where g a b = a + 1 for a b :: nat
  have g-is-pr: g ∈ PrimRec2 unfolding g-def by prec
  have h-lt-g: ∀ a b. h a b < g a b
  proof (rule allI, rule allI)
    fix a b
    from h-def have h a b ≤ a by simp
    also from g-def have a < g a b by simp
    ultimately show h a b < g a b by simp
  qed
  have f-at-h-nz: ∀ a b. f a b (h a b) ≠ 0
  proof (rule allI, rule allI)
    fix a b show f a b (h a b) ≠ 0
    proof cases
      assume A: b = 0
      with h-def have h a b = 0 by simp
      with f-def A show ?thesis by simp
    next
      assume A: b ≠ 0
      then have S1: b > 0 by auto
      from A f-def have S2: f a b (h a b) = sgn1 (b ∗ (h a b + 1) − a) by simp
      then have ?thesis = (sgn1(b ∗ (h a b + 1) − a) ≠ 0) by auto
      also have ... = (b ∗ (h a b + 1) − a > 0) by (rule sgn1-nz-eq-arg-pos)
      also have ... = (a < b ∗ (h a b + 1)) by auto
      also have ... = (a < b ∗ (h a b) + b) by auto
      also from h-def have ... = (a < b ∗ (a div b) + b) by simp
      finally have S3: ?thesis = (a < b ∗ (a div b) + b) by auto
      have S4: a < b ∗ (a div b) + b
      proof −
```

```
        from S1 have S4-1: a mod b < b by (rule mod-less-divisor)
        also have S4-2: b ∗ (a div b) + a mod b = a by (rule mult-div-mod-eq)
        from S4-1 have S4-3: b ∗ (a div b) + a mod b < b ∗ (a div b) + b by arith
        from S4-2 S4-3 show ?thesis by auto
      qed
      from S3 S4 show ?thesis by auto
    qed
  qed
  have h-is-min: ∀ z a b. z < h a b ⟶ f a b z = 0
  proof (rule allI, rule allI, rule allI, rule impI)
    fix a b z assume A: z < h a b show f a b z = 0
    proof −
      from A h-def have S1: z < a div b by simp
      then have S2: a div b > 0 by simp
      have S3: b ≠ 0
      proof (rule ccontr)
        assume ¬ b ≠ 0 then have b = 0 by auto
        then have a div b = 0 by auto
        with S2 show False by auto
      qed
      from S3 have b-pos: 0 < b by auto
      from S1 have S4: z+1 ≤ a div b by auto
      from b-pos have (b ∗ (z+1) ≤ b ∗ (a div b)) = (z+1 ≤ a div b) by (rule nat-mult-le-cancel1)
      with S4 have S5: b∗(z+1) ≤ b∗(a div b) by simp
      moreover have b∗(a div b) ≤ a
      proof −
        have b∗(a div b) + (a mod b) = a by (rule mult-div-mod-eq)
        moreover have 0 ≤ a mod b by auto
        ultimately show ?thesis by arith
      qed
      ultimately have S6: b∗(z+1) ≤ a
        by (simp add: minus-mod-eq-mult-div [symmetric])
      then have b∗(z+1) − a = 0 by auto
      with S3 f-def show ?thesis by simp
    qed
  qed
  from f-is-pr g-is-pr h-lt-g f-at-h-nz h-is-min have h-is-pr: h ∈ PrimRec2 by (rule b-least2-scheme2)
  with h-def [abs-def] show ?thesis by simp
qed

theorem mod-is-pr: (λ a b. a mod b) ∈ PrimRec2
proof −
  have (λ (a::nat) (b::nat). a mod b) = (λ a b. a − (a div b) ∗ b)
  proof (rule ext, rule ext)
    fix a b show (a::nat) mod b = a − (a div b) ∗ b by (rule minus-div-mult-eq-mod [symmetric])
  qed
```

```
  also from div-is-pr have (λ a b. a − (a div b) ∗ b) ∈ PrimRec2 by prec
  ultimately show ?thesis by auto
qed

theorem pr-rec-last-scheme: ⟦ g ∈ PrimRec1; h ∈ PrimRec3; ∀ x. f x 0 = g x; ∀
x y. f x (Suc y) = h x (f x y) y ⟧ ⟹ f ∈ PrimRec2
proof −
  assume g-is-pr: g ∈ PrimRec1
  assume h-is-pr: h ∈ PrimRec3
  assume f-at-0: ∀ x. f x 0 = g x
  assume f-at-Suc: ∀ x y. f x (Suc y) = h x (f x y) y
  from f-at-0 f-at-Suc have ⋀ x y. f x y = PrimRecOp-last g h x y by (induct-tac
y, simp-all)
  then have f = PrimRecOp-last g h by (simp add: ext)
  with g-is-pr h-is-pr show ?thesis by (simp add: pr-rec-last)
qed

theorem power-is-pr: (λ (x::nat) (n::nat). x ^ n) ∈ PrimRec2
proof −
  define g :: nat ⇒ nat where g x = 1 for x
  define h where h a b c = a ∗ b for a b c :: nat
  have g-is-pr: g ∈ PrimRec1 unfolding g-def by prec
  have h-is-pr: h ∈ PrimRec3 unfolding h-def by prec
  let ?f = λ (x::nat) (n::nat). x ^ n
  have f-at-0: ∀ x. ?f x 0 = g x
  proof
    fix x show x ^ 0 = g x by (simp add: g-def)
  qed
  have f-at-Suc: ∀ x y. ?f x (Suc y) = h x (?f x y) y
  proof (rule allI, rule allI)
    fix x y show ?f x (Suc y) = h x (?f x y) y by (simp add: h-def)
  qed
  from g-is-pr h-is-pr f-at-0 f-at-Suc show ?thesis by (rule pr-rec-last-scheme)
qed

end
```

3 Primitive recursive coding of lists of natural numbers

```
theory PRecList
imports PRecFun
begin
```

We introduce a particular coding *list-to-nat* from lists of natural numbers to natural numbers.

```
definition
  c-len :: nat ⇒ nat where
```

```
  c-len = (λ (u::nat). (sgn1 u) ∗ (c-fst(u−(1::nat))+1))

lemma c-len-1: c-len u = (case u of 0 ⇒ 0 | Suc v ⇒ c-fst(v)+1) by (unfold
c-len-def, cases u, auto)

lemma c-len-is-pr: c-len ∈ PrimRec1 unfolding c-len-def by prec

lemma [simp]: c-len 0 = 0 by (simp add: c-len-def)

lemma c-len-2: u ≠ 0 ⟹ c-len u = c-fst(u−(1::nat))+1 by (simp add: c-len-def)

lemma c-len-3: u>0 ⟹ c-len u > 0 by (simp add: c-len-2)

lemma c-len-4: c-len u = 0 ⟹ u = 0
proof cases
  assume A1: u = 0
  thus ?thesis by simp
next
  assume A1: c-len u = 0 and A2: u ≠ 0
  from A2 have c-len u > 0 by (simp add: c-len-3)
  from A1 this show u=0 by simp
qed

lemma c-len-5: c-len u > 0 ⟹ u > 0
proof cases
  assume A1: c-len u > 0 and A2: u=0
  from A2 have c-len u = 0 by simp
  from A1 this show ?thesis by simp
next
  assume A1: u ≠ 0
  from A1 show u>0 by simp
qed

fun c-fold :: nat list ⇒ nat where
    c-fold [] = 0
  | c-fold [x] = x
  | c-fold (x#ls) = c-pair x (c-fold ls)

lemma c-fold-0: ls ≠ [] ⟹ c-fold (x#ls) = c-pair x (c-fold ls)
proof −
  assume A1: ls ≠ []
  then have S1: ls = (hd ls)#(tl ls) by simp
  then have S2: x#ls = x#(hd ls)#(tl ls) by simp
  have S3: c-fold (x#(hd ls)#(tl ls)) = c-pair x (c-fold ((hd ls)#(tl ls))) by simp
  from S1 S2 S3 show ?thesis by simp
qed

primrec
  c-unfold :: nat ⇒ nat ⇒ nat list
```

```
where
  c-unfold 0 u = []
| c-unfold (Suc k) u = (if k = 0 then [u] else ((c-fst u) # (c-unfold k (c-snd u))))

lemma c-fold-1: c-unfold 1 (c-fold [x]) = [x] by simp

lemma c-fold-2: c-fold (c-unfold 1 u) = u by simp

lemma c-unfold-1: c-unfold 1 u = [u] by simp

lemma c-unfold-2: c-unfold (Suc 1) u = (c-fst u) # (c-unfold 1 (c-snd u)) by
simp

lemma c-unfold-3: c-unfold (Suc 1) u = [c-fst u, c-snd u] by simp

lemma c-unfold-4: k > 0 ⟹ c-unfold (Suc k) u = (c-fst u) # (c-unfold k (c-snd
u)) by simp

lemma c-unfold-4-1: k > 0 ⟹ c-unfold (Suc k) u ≠ [] by (simp add: c-unfold-4)

lemma two: (2::nat) = Suc 1 by simp

lemma c-unfold-5: c-unfold 2 u = [c-fst u, c-snd u] by (simp add: two)

lemma c-unfold-6: k>0 ⟹ c-unfold k u ≠ []
proof −
  assume A1: k>0
  let ?k1 = k−(1::nat)
  from A1 have S1: k = Suc ?k1 by simp
  have S2: ?k1 = 0 ⟹ ?thesis
  proof −
    assume A2-1: ?k1=0
    from A1 A2-1 have S2-1: k=1 by simp
    from S2-1 show ?thesis by (simp add: c-unfold-1)
  qed
  have S3: ?k1 > 0 ⟹ ?thesis
  proof −
    assume A3-1: ?k1 > 0
    from A3-1 have S3-1: c-unfold (Suc ?k1) u ≠ [] by (rule c-unfold-4-1)
    from S1 S3-1 show ?thesis by simp
  qed
  from S2 S3 show ?thesis by arith
qed

lemma th-lm-1: k=1 ⟹ (∀ u. c-fold (c-unfold k u) = u) by (simp add: c-fold-2)

lemma th-lm-2: ⟦k>0; (∀ u. c-fold (c-unfold k u) = u)⟧ ⟹ (∀ u. c-fold (c-unfold
(Suc k) u) = u)
proof
```

assume *A1*: *k>0*
assume *A2*: ∀ *u*. *c-fold* (*c-unfold k u*) = *u*
fix *u*
from *A1* **have** *S1*: *c-unfold* (*Suc k*) *u* = (*c-fst u*) # (*c-unfold k* (*c-snd u*)) **by** (*rule c-unfold-4*)
let *?ls* = *c-unfold k* (*c-snd u*)
from *A1* **have** *S2*: *?ls* ≠ [] **by** (*rule c-unfold-6*)
from *S2* **have** *S3*: *c-fold* ((*c-fst u*) # *?ls*) = *c-pair* (*c-fst u*) (*c-fold ?ls*) **by** (*rule c-fold-0*)
from *A2* **have** *S4*: *c-fold ?ls* = *c-snd u* **by** *simp*
from *S3 S4* **have** *S5*: *c-fold* ((*c-fst u*) # *?ls*) = *c-pair* (*c-fst u*) (*c-snd u*) **by** *simp*
from *S5* **have** *S6*: *c-fold* ((*c-fst u*) # *?ls*) = *u* **by** *simp*
from *S1 S6* **have** *S7*: *c-fold* (*c-unfold* (*Suc k*) *u*) = *u* **by** *simp*
thus *c-fold* (*c-unfold* (*Suc k*) *u*) = *u* **.**
qed

lemma *th-lm-3*: (∀ *u*. *c-fold* (*c-unfold* (*Suc k*) *u*) = *u*)⟹ (∀ *u*. *c-fold* (*c-unfold* (*Suc* (*Suc k*)) *u*) = *u*)
proof −
assume *A1*: ∀ *u*. *c-fold* (*c-unfold* (*Suc k*) *u*) = *u*
let *?k1* = *Suc k*
have *S1*: *?k1* > *0* **by** *simp*
from *S1 A1* **have** *S2*: ∀ *u*. *c-fold* (*c-unfold* (*Suc ?k1*) *u*) = *u* **by** (*rule th-lm-2*)
thus *?thesis* **by** *simp*
qed

theorem *th-1*: ∀ *u*. *c-fold* (*c-unfold* (*Suc k*) *u*) = *u*
apply(*induct k*)
apply(*simp add*: *c-fold-2*)
apply(*rule th-lm-3*)
apply(*assumption*)
done

theorem *th-2*: *k* > *0* ⟹ (∀ *u*. *c-fold* (*c-unfold k u*) = *u*)
proof −
assume *A1*: *k>0*
let *?k1* = *k*−(*1*::*nat*)
from *A1* **have** *S1*: *Suc ?k1* = *k* **by** *simp*
have *S2*: ∀ *u*. *c-fold* (*c-unfold* (*Suc ?k1*) *u*) = *u* **by** (*rule th-1*)
from *S1 S2* **show** *?thesis* **by** *simp*
qed

lemma *c-fold-3*: *c-unfold 2* (*c-fold* [*x*, *y*]) = [*x*, *y*] **by** (*simp add*: *two*)

theorem *c-unfold-len*: *ALL u*. *length* (*c-unfold k u*) = *k*
apply(*induct k*)
apply(*simp*)
apply(*subgoal-tac n*=(*0*::*nat*) ∨ *n*>*0*)

```
apply(drule disjE)
prefer 3
apply(simp-all)
apply(auto)
done

lemma th-3-lm-0: ⟦c-unfold (length ls) (c-fold ls) = ls; ls = a # ls1; ls1 = aa # list⟧ ⟹ c-unfold (length (x # ls)) (c-fold (x # ls)) = x # ls
proof –
  assume A1: c-unfold (length ls) (c-fold ls) = ls
  assume A2: ls = a # ls1
  assume A3: ls1 = aa # list
  from A2 have S1: ls ≠ [] by simp
  from S1 have S2: c-fold (x#ls) = c-pair x (c-fold ls) by (rule c-fold-0)
  have S3: length (x#ls) = Suc (length ls) by simp
  from S3 have S4: c-unfold (length (x # ls)) (c-fold (x # ls)) = c-unfold (Suc (length ls)) (c-fold (x # ls)) by simp
  from A2 have S5: length ls > 0 by simp
  from S5 have S6: c-unfold (Suc (length ls)) (c-fold (x # ls)) = c-fst (c-fold (x # ls))#(c-unfold (length ls) (c-snd (c-fold (x#ls)))) by (rule c-unfold-4)
  from S2 have S7: c-fst (c-fold (x#ls)) = x by simp
  from S2 have S8: c-snd (c-fold (x#ls)) = c-fold ls by simp
  from S6 S7 S8 have S9: c-unfold (Suc (length ls)) (c-fold (x # ls)) = x # (c-unfold (length ls) (c-fold ls)) by simp
  from A1 have S10: x # (c-unfold (length ls) (c-fold ls)) = x # ls by simp
  from S9 S10 have S11: c-unfold (Suc (length ls)) (c-fold (x # ls)) = (x # ls) by simp
  thus ?thesis by simp
qed

lemma th-3-lm-1: ⟦c-unfold (length ls) (c-fold ls) = ls; ls = a # ls1⟧ ⟹ c-unfold (length (x # ls)) (c-fold (x # ls)) = x # ls
apply(cases ls1)
apply(simp add: c-fold-1)
apply(simp)
done

lemma th-3-lm-2: c-unfold (length ls) (c-fold ls) = ls ⟹ c-unfold (length (x # ls)) (c-fold (x # ls)) = x # ls
apply(cases ls)
apply(simp add: c-fold-1)
apply(rule th-3-lm-1)
apply(assumption+)
done

theorem th-3: c-unfold (length ls) (c-fold ls) = ls
apply(induct ls)
apply(simp)
apply(rule th-3-lm-2)
```

```
apply(assumption)
done

definition
  list-to-nat :: nat list ⇒ nat where
  list-to-nat = (λ ls. if ls=[] then 0 else (c-pair ((length ls) − 1) (c-fold ls))+1)

definition
  nat-to-list :: nat ⇒ nat list where
  nat-to-list = (λ u. if u=0 then [] else (c-unfold (c-len u) (c-snd (u−(1::nat)))))

lemma nat-to-list-of-pos: u>0 ⟹ nat-to-list u = c-unfold (c-len u) (c-snd (u−(1::nat)))
by (simp add: nat-to-list-def)

theorem list-to-nat-th [simp]: list-to-nat (nat-to-list u) = u
proof −
  have S1: u=0 ⟹ ?thesis by (simp add: list-to-nat-def nat-to-list-def)
  have S2: u>0 ⟹ ?thesis
  proof −
    assume A1: u>0
    define ls where ls = nat-to-list u
     from ls-def A1 have S2-1: ls = c-unfold (c-len u) (c-snd (u−(1::nat))) by
(simp add: nat-to-list-def)
    let ?k = c-len u
    from A1 have S2-2: ?k > 0 by (rule c-len-3)
    from S2-1 have S2-3: length ls = ?k by (simp add: c-unfold-len)
    from S2-2 S2-3 have S2-4: length ls > 0 by simp
    from S2-4 have S2-5: ls ≠ [] by simp
   from S2-5 have S2-6: list-to-nat ls = c-pair ((length ls)−(1::nat)) (c-fold ls)+1
by (simp add: list-to-nat-def)
    have S2-7: c-fold ls = c-snd(u−(1::nat))
    proof −
     from S2-1 have S2-7-1: c-fold ls = c-fold (c-unfold (c-len u) (c-snd (u−(1::nat))))
by simp
      from S2-2 S2-7-1 show ?thesis by (simp add: th-2)
    qed
    have S2-8: (length ls)−(1::nat) = c-fst (u−(1::nat))
    proof −
      from S2-3 have S2-8-1: length ls = c-len u by simp
       from A1 S2-8-1 have S2-8-2: length ls = c-fst(u−(1::nat)) + 1 by (simp
add: c-len-2)
      from S2-8-2 show ?thesis by simp
    qed
     from S2-7 S2-8 have S2-9: c-pair ((length ls)−(1::nat)) (c-fold ls) = c-pair
(c-fst (u−(1::nat))) (c-snd (u−(1::nat))) by simp
    from S2-9 have S2-10: c-pair ((length ls)−(1::nat)) (c-fold ls) = u − (1::nat)
by simp
    from S2-6 S2-10 have S2-11: list-to-nat ls = (u − (1::nat))+1 by simp
    from A1 have S2-12: (u − (1::nat))+1 = u by simp
```

```
    from ls-def S2-11 S2-12 show ?thesis by simp
  qed
  from S1 S2 show ?thesis by arith
qed

theorem nat-to-list-th [simp]: nat-to-list (list-to-nat ls) = ls
proof −
  have S1: ls=[] ⟹ ?thesis by (simp add: nat-to-list-def list-to-nat-def)
  have S2: ls ≠ [] ⟹ ?thesis
  proof −
    assume A1: ls ≠ []
    define u where u = list-to-nat ls
    from u-def A1 have S2-1: u = (c-pair ((length ls)−(1::nat)) (c-fold ls))+1 by
(simp add: list-to-nat-def)
    let ?k = length ls
    from A1 have S2-2: ?k > 0 by simp
    from S2-1 have S2-3: u>0 by simp
    from S2-3 have S2-4: nat-to-list u = c-unfold (c-len u) (c-snd (u−(1::nat)))
by (simp add: nat-to-list-def)
    have S2-5: c-len u = length ls
    proof −
      from S2-1 have S2-5-1: u−(1::nat) = c-pair ((length ls)−(1::nat)) (c-fold
ls) by simp
      from S2-5-1 have S2-5-2: c-fst (u−(1::nat)) = (length ls)−(1::nat) by simp
      from S2-2 S2-5-2 have c-fst (u−(1::nat))+1 = length ls by simp
      from S2-3 this show ?thesis by (simp add: c-len-2)
    qed
    have S2-6: c-snd (u−(1::nat)) = c-fold ls
    proof −
      from S2-1 have S2-6-1: u−(1::nat) = c-pair ((length ls)−(1::nat)) (c-fold
ls) by simp
      from S2-6-1 show ?thesis by simp
    qed
    from S2-4 S2-5 S2-6 have S2-7:nat-to-list u = c-unfold (length ls) (c-fold ls)
by simp
    from S2-7 have nat-to-list u = ls by (simp add: th-3)
    from u-def this show ?thesis by simp
  qed
  have S3: ls = [] ∨ ls ≠ [] by simp
  from S1 S2 S3 show ?thesis by auto
qed

lemma [simp]: list-to-nat [] = 0 by (simp add: list-to-nat-def)

lemma [simp]: nat-to-list 0 = [] by (simp add: nat-to-list-def)

theorem c-len-th-1: c-len (list-to-nat ls) = length ls
proof (cases)
  assume ls=[]
```

```
  from this show ?thesis by simp
next
  assume S1: ls ≠ []
  then have S2: list-to-nat ls = c-pair ((length ls)−(1::nat)) (c-fold ls)+1 by
(simp add: list-to-nat-def)
  let ?u = list-to-nat ls
  from S2 have u-not-zero: ?u > 0 by simp
  from S2 have S3: ?u−(1::nat) = c-pair ((length ls)−(1::nat)) (c-fold ls) by
simp
  then have S4: c-fst(?u−(1::nat)) = (length ls)−(1::nat) by simp
  from S1 this have S5: c-fst(?u−(1::nat))+1=length ls by simp
  from u-not-zero S5 have S6: c-len (?u) = length ls by (simp add: c-len-2)
  from S1 S6 show ?thesis by simp
qed

theorem length (nat-to-list u) = c-len u
proof −
  let ?ls = nat-to-list u
  have S1: u = list-to-nat ?ls by (rule list-to-nat-th [THEN sym])
  from c-len-th-1 have S2: length ?ls = c-len (list-to-nat ?ls) by (rule sym)
  from S1 S2 show ?thesis by (rule ssubst)
qed

definition
  c-hd :: nat ⇒ nat where
  c-hd = (λ u. if u=0 then 0 else hd (nat-to-list u))

definition
  c-tl :: nat ⇒ nat where
  c-tl = (λ u. list-to-nat (tl (nat-to-list u)))

definition
  c-cons :: nat ⇒ nat ⇒ nat where
  c-cons = (λ x u. list-to-nat (x # (nat-to-list u)))


lemma [simp]: c-hd 0 = 0 by (simp add: c-hd-def)

lemma c-hd-aux0: c-len u = 1 ⟹ nat-to-list u = [c-snd (u−(1::nat))] by (simp
add: nat-to-list-def c-len-5)

lemma c-hd-aux1: c-len u = 1 ⟹ c-hd u = c-snd (u−(1::nat))
proof −
  assume A1: c-len u = 1
  then have S1: nat-to-list u = [c-snd (u−(1::nat))] by (simp add: nat-to-list-def
c-len-5)
  from A1 have u > 0 by (simp add: c-len-5)
  with S1 show ?thesis by (simp add: c-hd-def)
qed
```

lemma *c-hd-aux2*: *c-len u* > *1* ⟹ *c-hd u* = *c-fst* (*c-snd* (*u*−(*1*::*nat*)))
proof −
assume *A1*: *c-len u* > *1*
let *?k* = (*c-len u*) − *1*
from *A1* **have** *S1*: *c-len u* = *Suc ?k* **by** *simp*
from *A1* **have** *S2*: *c-len u* > *0* **by** *simp*
from *S2* **have** *S3*: *u* > *0* **by** (*rule c-len-5*)
from *S3* **have** *S4*: *c-hd u* = *hd* (*nat-to-list u*) **by** (*simp add*: *c-hd-def*)
from *S3* **have** *S5*: *nat-to-list u* = *c-unfold* (*c-len u*) (*c-snd* (*u*−(*1*::*nat*))) **by** (*rule nat-to-list-of-pos*)
from *S1 S5* **have** *S6*: *nat-to-list u* = *c-unfold* (*Suc ?k*) (*c-snd* (*u*−(*1*::*nat*))) **by** *simp*
from *A1* **have** *S7*: *?k* > *0* **by** *simp*
from *S7* **have** *S8*: *c-unfold* (*Suc ?k*) (*c-snd* (*u*−(*1*::*nat*))) = (*c-fst* (*c-snd* (*u*−(*1*::*nat*)))) # (*c-unfold ?k* (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) **by** (*rule c-unfold-4*)
from *S6 S8* **have** *S9*: *nat-to-list u* = (*c-fst* (*c-snd* (*u*−(*1*::*nat*)))) # (*c-unfold ?k* (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) **by** *simp*
from *S9* **have** *S10*: *hd* (*nat-to-list u*) = *c-fst* (*c-snd* (*u*−(*1*::*nat*))) **by** *simp*
from *S4 S10* **show** *?thesis* **by** *simp*
qed

lemma *c-hd-aux3*: *u* > *0* ⟹ *c-hd u* = (*if* (*c-len u*) = *1* *then c-snd* (*u*−(*1*::*nat*)) *else c-fst* (*c-snd* (*u*−(*1*::*nat*))))
proof −
assume *A1*: *u* > *0*
from *A1* **have** *c-len u* > *0* **by** (*rule c-len-3*)
then have *S1*: *c-len u* = *1* ∨ *c-len u* > *1* **by** *arith*
let *?tmp* = *if* (*c-len u*) = *1* *then c-snd* (*u*−(*1*::*nat*)) *else c-fst* (*c-snd* (*u*−(*1*::*nat*)))
have *S2*: *c-len u* = *1* ⟹ *?thesis*
proof −
assume *A2-1*: *c-len u* = *1*
then have *S2-1*: *c-hd u* = *c-snd* (*u*−(*1*::*nat*)) **by** (*rule c-hd-aux1*)
from *A2-1* **have** *S2-2*: *?tmp* = *c-snd*(*u*−(*1*::*nat*)) **by** *simp*
from *S2-1 this* **show** *?thesis* **by** *simp*
qed
have *S3*: *c-len u* > *1* ⟹ *?thesis*
proof −
assume *A3-1*: *c-len u* > *1*
from *A3-1* **have** *S3-1*: *c-hd u* = *c-fst* (*c-snd* (*u*−(*1*::*nat*))) **by** (*rule c-hd-aux2*)
from *A3-1* **have** *S3-2*: *?tmp* = *c-fst* (*c-snd* (*u*−(*1*::*nat*))) **by** *simp*
from *S3-1 this* **show** *?thesis* **by** *simp*
qed
from *S1 S2 S3* **show** *?thesis* **by** *auto*
qed

lemma *c-hd-aux4*: *c-hd u* = (*if u*=*0* *then 0* *else* (*if* (*c-len u*) = *1* *then c-snd* (*u*−(*1*::*nat*)) *else c-fst* (*c-snd* (*u*−(*1*::*nat*)))))
proof *cases*

```
  assume u=0 then show ?thesis by simp
next
  assume u ≠ 0 then have A1: u > 0 by simp
  then show ?thesis by (simp add: c-hd-aux3)
qed

lemma c-hd-is-pr: c-hd ∈ PrimRec1
proof −
  have c-hd = (%u. (if u=0 then 0 else (if (c-len u) = 1 then c-snd (u−(1::nat))
else c-fst (c-snd (u−(1::nat)))))) (is - = ?R) by (simp add: c-hd-aux4 ext)
  moreover have ?R ∈ PrimRec1
  proof (rule if-is-pr)
    show (λ x. x) ∈ PrimRec1 by (rule pr-id1-1)
    next show (λ x. 0) ∈ PrimRec1 by (rule pr-zero)
    next show (λx. if c-len x = 1 then c-snd (x − 1) else c-fst (c-snd (x − 1)))
∈ PrimRec1
    proof (rule if-eq-is-pr)
      show c-len ∈ PrimRec1 by (rule c-len-is-pr)
      next show (λ x. 1) ∈ PrimRec1 by (rule const-is-pr)
      next show (λx. c-snd (x − 1)) ∈ PrimRec1 by prec
      next show (λx. c-fst (c-snd (x − 1))) ∈ PrimRec1 by prec
    qed
  qed
  ultimately show ?thesis by simp
qed

lemma [simp]: c-tl 0 = 0 by (simp add: c-tl-def)

lemma c-tl-eq-tl: c-tl (list-to-nat ls) = list-to-nat (tl ls) by (simp add: c-tl-def)

lemma tl-eq-c-tl: tl (nat-to-list x) = nat-to-list (c-tl x) by (simp add: c-tl-def)

lemma c-tl-aux1: c-len u = 1 ⟹ c-tl u = 0 by (unfold c-tl-def, simp add:
c-hd-aux0)

lemma c-tl-aux2: c-len u > 1 ⟹ c-tl u = (c-pair (c-len u − (2::nat)) (c-snd
(c-snd (u−(1::nat))))) + 1
proof −
  assume A1: c-len u > 1
  let ?k = (c-len u) − 1
  from A1 have S1: c-len u = Suc ?k by simp
  from A1 have S2: c-len u > 0 by simp
  from S2 have S3: u > 0 by (rule c-len-5)
  from S3 have S4: nat-to-list u = c-unfold (c-len u) (c-snd (u−(1::nat))) by
(rule nat-to-list-of-pos)
  from A1 have S5: ?k > 0 by simp
  from S5 have S6: c-unfold (Suc ?k) (c-snd (u−(1::nat))) = (c-fst (c-snd (u−(1::nat))))
# (c-unfold ?k (c-snd (c-snd (u−(1::nat))))) by (rule c-unfold-4)
  from S6 have S7: tl (c-unfold (Suc ?k) (c-snd (u−(1::nat)))) = c-unfold ?k
```

(*c-snd* (*c-snd* (*u*−(*1*::*nat*)))) **by** *simp*
from *S2 S4 S7* **have** *S8*: *tl* (*nat-to-list u*) = *c-unfold ?k* (*c-snd* (*c-snd* (*u*−(*1*::*nat*)))) **by** *simp*
define *ls* **where** *ls* = *tl* (*nat-to-list u*)
from *ls-def S8* **have** *S9*: *length ls* = *?k* **by** (*simp add*: *c-unfold-len*)
from *ls-def* **have** *S10*: *c-tl u* = *list-to-nat ls* **by** (*simp add*: *c-tl-def*)
from *S5 S9* **have** *S11*: *length ls* > *0* **by** *simp*
from *S11* **have** *S12*: *ls* ≠ [] **by** *simp*
from *S12* **have** *S13*: *list-to-nat ls* = (*c-pair* ((*length ls*) − *1*) (*c-fold ls*))+*1* **by** (*simp add*: *list-to-nat-def*)
from *S10 S13* **have** *S14*: *c-tl u* = (*c-pair* ((*length ls*) − *1*) (*c-fold ls*))+*1* **by** *simp*
from *S9* **have** *S15*: (*length ls*)−(*1*::*nat*) = *?k*−(*1*::*nat*) **by** *simp*
from *A1* **have** *S16*: *?k*−(*1*::*nat*) = *c-len u* − (*2*::*nat*) **by** *arith*
from *S15 S16* **have** *S17*: (*length ls*)−(*1*::*nat*) = *c-len u* − (*2*::*nat*) **by** *simp*
from *ls-def S8* **have** *S18*: *ls* = *c-unfold ?k* (*c-snd* (*c-snd* (*u*−(*1*::*nat*)))) **by** *simp*
from *S5* **have** *S19*: *c-fold* (*c-unfold ?k* (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) = *c-snd* (*c-snd* (*u*−(*1*::*nat*))) **by** (*simp add*: *th-2*)
from *S18 S19* **have** *S20*: *c-fold ls* = *c-snd* (*c-snd* (*u*−(*1*::*nat*))) **by** *simp*
from *S14 S17 S20* **show** *?thesis* **by** *simp*
qed

lemma *c-tl-aux3*: *c-tl u* = (*sgn1* ((*c-len u*) − *1*))∗((*c-pair* (*c-len u* − (*2*::*nat*)) (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) + *1*) (**is** - = *?R*)
proof −
have *S1*: *u*=*0* ⟹ *?thesis* **by** *simp*
have *S2*: *u*>*0* ⟹ *?thesis*
proof −
assume *A1*: *u*>*0*
have *S2-1*: *c-len u* = *1* ⟹ *?thesis* **by** (*simp add*: *c-tl-aux1*)
have *S2-2*: *c-len u* ≠ *1* ⟹ *?thesis*
proof −
assume *A2-2-1*: *c-len u* ≠ *1*
from *A1* **have** *S2-2-1*: *c-len u* > *0* **by** (*rule c-len-3*)
from *A2-2-1 S2-2-1* **have** *S2-2-2*: *c-len u* > *1* **by** *arith*
from *this* **have** *S2-2-3*: *c-len u* − *1* > *0* **by** *simp*
from *this* **have** *S2-2-4*: *sgn1* (*c-len u* − *1*)=*1* **by** *simp*
from *S2-2-4* **have** *S2-2-5*: *?R* = (*c-pair* (*c-len u* − (*2*::*nat*)) (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) + *1* **by** *simp*
from *S2-2-2* **have** *S2-2-6*: *c-tl u* = (*c-pair* (*c-len u* − (*2*::*nat*)) (*c-snd* (*c-snd* (*u*−(*1*::*nat*))))) + *1* **by** (*rule c-tl-aux2*)
from *S2-2-5 S2-2-6* **show** *?thesis* **by** *simp*
qed
from *S2-1 S2-2* **show** *?thesis* **by** *blast*
qed
from *S1 S2* **show** *?thesis* **by** *arith*
qed

lemma *c-tl-less*: *u* > *0* ⟹ *c-tl u* < *u*

```
proof −
  assume A1: u > 0
  then have S1: c-len u > 0 by (rule c-len-3)
  then show ?thesis
  proof cases
    assume c-len u = 1
    from this A1 show ?thesis by (simp add: c-tl-aux1)
  next
    assume ¬ c-len u = 1 with S1 have A2: c-len u > 1 by simp
   then have S2: c-tl u = (c-pair (c-len u − (2::nat)) (c-snd (c-snd (u−(1::nat)))))
+ 1 by (rule c-tl-aux2)
    from A1 have S3: c-len u = c-fst(u−(1::nat))+1 by (simp add: c-len-def)
    from A2 S3 have S4: c-len u − (2::nat) < c-fst(u−(1::nat)) by simp
     then have S5: (c-pair (c-len u − (2::nat)) (c-snd (c-snd (u−(1::nat))))) <
(c-pair (c-fst(u−(1::nat))) (c-snd (c-snd (u−(1::nat))))) by (rule c-pair-strict-mono1)
   have S6: c-snd (c-snd (u−(1::nat))) ≤ c-snd (u−(1::nat)) by (rule c-snd-le-arg)
     then have S7: (c-pair (c-fst(u−(1::nat))) (c-snd (c-snd (u−(1::nat))))) ≤
(c-pair (c-fst(u−(1::nat))) (c-snd (u−(1::nat)))) by (rule c-pair-mono2)
     then have S8: (c-pair (c-fst(u−(1::nat))) (c-snd (c-snd (u−(1::nat))))) ≤
u−(1::nat) by simp
    with S5 have (c-pair (c-len u − (2::nat)) (c-snd (c-snd (u−(1::nat))))) < u
− (1::nat) by simp
    with S2 have c-tl u < (u−(1::nat))+1 by simp
    with A1 show ?thesis by simp
  qed
qed

lemma c-tl-le: c-tl u ≤ u
proof (cases u)
  assume u=0
  then show ?thesis by simp
next
  fix v assume A1: u = Suc v
  then have S1: u > 0 by simp
  then have S2: c-tl u < u by (rule c-tl-less)
  with A1 show c-tl u ≤ u by simp
qed

theorem c-tl-is-pr: c-tl ∈ PrimRec1
proof −
  have c-tl = (λ u. (sgn1 ((c-len u) − 1))∗((c-pair (c-len u − (2::nat)) (c-snd
(c-snd (u−(1::nat))))) + 1)) (is - = ?R) by (simp add: c-tl-aux3 ext)
  moreover from c-len-is-pr c-pair-is-pr have ?R ∈ PrimRec1 by prec
  ultimately show ?thesis by simp
qed

lemma c-cons-aux1: c-cons x 0 = (c-pair 0 x) + 1
apply(unfold c-cons-def)
apply(simp)
```

```
apply(unfold list-to-nat-def)
apply(simp)
done

lemma c-cons-aux2: u > 0 ⟹ c-cons x u = (c-pair (c-len u) (c-pair x (c-snd
(u−(1::nat))))) + 1
proof −
  assume A1: u > 0
  from A1 have S1: c-len u > 0 by (rule c-len-3)
  from A1 have S2: nat-to-list u = c-unfold (c-len u) (c-snd (u−(1::nat))) by
(rule nat-to-list-of-pos)
  define ls where ls = nat-to-list u
  from ls-def S2 have S3: ls = c-unfold (c-len u) (c-snd (u−(1::nat))) by simp
  from S3 have S4: length ls = c-len u by (simp add: c-unfold-len)
  from S4 S1 have S5: length ls > 0 by simp
  from S5 have S6: ls ≠ [] by simp
  from ls-def have S7: c-cons x u = list-to-nat (x # ls) by (simp add: c-cons-def)
  have S8: list-to-nat (x # ls) = (c-pair ((length (x#ls))−(1::nat)) (c-fold (x#ls)))+1
by (simp add: list-to-nat-def)
  have S9: (length (x#ls))−(1::nat) = length ls by simp
  from S9 S4 S8 have S10: list-to-nat (x # ls) = (c-pair (c-len u) (c-fold
(x#ls)))+1 by simp
  have S11: c-fold (x#ls) = c-pair x (c-snd (u−(1::nat)))
  proof −
    from S6 have S11-1: c-fold (x#ls) = c-pair x (c-fold ls) by (rule c-fold-0)
    from S3 have S11-2: c-fold ls = c-fold (c-unfold (c-len u) (c-snd (u−(1::nat))))
by simp
    from S1 S11-2 have S11-3: c-fold ls = c-snd (u−(1::nat)) by (simp add: th-2)
    from S11-1 S11-3 show ?thesis by simp
  qed
  from S7 S10 S11 show ?thesis by simp
qed

lemma c-cons-aux3: c-cons = (λ x u. (sgn2 u)∗((c-pair 0 x)+1) + (sgn1 u)∗((c-pair
(c-len u) (c-pair x (c-snd (u−(1::nat))))) + 1))
proof (rule ext, rule ext)
  fix x u show c-cons x u = (sgn2 u)∗((c-pair 0 x)+1) + (sgn1 u)∗((c-pair (c-len
u) (c-pair x (c-snd (u−(1::nat))))) + 1) (is - = ?R)
  proof cases
    assume A1: u=0
    then have ?R = (c-pair 0 x)+1 by simp
    moreover from A1 have c-cons x u = (c-pair 0 x)+1 by (simp add: c-cons-aux1)
    ultimately show ?thesis by simp
  next
    assume A1: u≠0
    then have S1: ?R = (c-pair (c-len u) (c-pair x (c-snd (u−(1::nat))))) + 1 by
simp
    from A1 have S2: c-cons x u = (c-pair (c-len u) (c-pair x (c-snd (u−(1::nat)))))
+ 1 by (simp add: c-cons-aux2)
```

```
    from S1 S2 have c-cons x u = ?R by simp
    then show ?thesis .
  qed
qed

lemma c-cons-pos: c-cons x u > 0
proof cases
  assume u=0
  then show c-cons x u > 0 by (simp add: c-cons-aux1)
next
  assume ¬ u=0 then have u>0 by simp
  then show c-cons x u > 0 by (simp add: c-cons-aux2)
qed

theorem c-cons-is-pr: c-cons ∈ PrimRec2
proof −
  have c-cons = (λ x u. (sgn2 u)∗((c-pair 0 x)+1) + (sgn1 u)∗((c-pair (c-len u)
(c-pair x (c-snd (u−(1::nat))))) + 1)) (is - = ?R) by (simp add: c-cons-aux3)
  moreover from c-pair-is-pr c-len-is-pr have ?R ∈ PrimRec2 by prec
  ultimately show ?thesis by simp
qed

definition
  c-drop :: nat ⇒ nat ⇒ nat where
  c-drop = PrimRecOp (λ x. x) (λ x y z. c-tl y)

lemma c-drop-at-0 [simp]: c-drop 0 x = x by (simp add: c-drop-def)

lemma c-drop-at-Suc: c-drop (Suc y) x = c-tl (c-drop y x) by (simp add: c-drop-def)

theorem c-drop-is-pr: c-drop ∈ PrimRec2
proof −
  have (λ x. x) ∈ PrimRec1 by (rule pr-id1-1)
  moreover from c-tl-is-pr have (λ x y z. c-tl y) ∈ PrimRec3 by prec
  ultimately show ?thesis by (simp add: c-drop-def pr-rec)
qed

lemma c-tl-c-drop: c-tl (c-drop y x) = c-drop y (c-tl x)
apply(induct y)
apply(simp)
apply(simp add: c-drop-at-Suc)
done

lemma c-drop-at-Suc1: c-drop (Suc y) x = c-drop y (c-tl x)
apply(simp add: c-drop-at-Suc c-tl-c-drop)
done

lemma c-drop-df: ∀ ls. drop n ls = nat-to-list (c-drop n (list-to-nat ls))
proof (induct n)
```

show ∀ *ls. drop 0 ls = nat-to-list* (*c-drop 0* (*list-to-nat ls*)) **by** (*simp add*: *c-drop-def*)
next
fix *n* **assume** *A1*: ∀ *ls. drop n ls = nat-to-list* (*c-drop n* (*list-to-nat ls*))
then show ∀ *ls. drop* (*Suc n*) *ls = nat-to-list* (*c-drop* (*Suc n*) (*list-to-nat ls*))
proof −
{
fix *ls*::*nat list*
have *S1*: *drop* (*Suc n*) *ls = drop n* (*tl ls*) **by** (*rule drop-Suc*)
from *A1* **have** *S2*: *drop n* (*tl ls*) = *nat-to-list* (*c-drop n* (*list-to-nat* (*tl ls*))) **by** *simp*
also have ... = *nat-to-list* (*c-drop n* (*c-tl* (*list-to-nat ls*))) **by** (*simp add*: *c-tl-eq-tl*)
also have ... = *nat-to-list* (*c-drop* (*Suc n*) (*list-to-nat ls*)) **by** (*simp add*: *c-drop-at-Suc1*)
finally have *drop n* (*tl ls*) = *nat-to-list* (*c-drop* (*Suc n*) (*list-to-nat ls*)) **by** *simp*
with *S1* **have** *drop* (*Suc n*) *ls = nat-to-list* (*c-drop* (*Suc n*) (*list-to-nat ls*)) **by** *simp*
}
then show *?thesis* **by** *blast*
qed
qed

definition
c-nth :: *nat* ⇒ *nat* ⇒ *nat* **where**
c-nth = (*λ x n. c-hd* (*c-drop n x*))

lemma *c-nth-is-pr*: *c-nth* ∈ *PrimRec2*
proof (*unfold c-nth-def*)
from *c-hd-is-pr c-drop-is-pr* **show** (*λx n. c-hd* (*c-drop n x*)) ∈ *PrimRec2* **by** *prec*
qed

lemma *c-nth-at-0*: *c-nth x 0 = c-hd x* **by** (*simp add*: *c-nth-def*)

lemma *c-hd-c-cons* [*simp*]: *c-hd* (*c-cons x y*) = *x*
proof −
have *c-cons x y* > *0* **by** (*rule c-cons-pos*)
then show *?thesis* **by** (*simp add*: *c-hd-def c-cons-def*)
qed

lemma *c-tl-c-cons* [*simp*]: *c-tl* (*c-cons x y*) = *y* **by** (*simp add*: *c-tl-def c-cons-def*)

definition
c-f-list :: (*nat* ⇒ *nat* ⇒ *nat*) ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
c-f-list = (*λ f.*
let g = (*%x. c-cons* (*f 0 x*) *0*); *h* = (*%a b c. c-cons* (*f* (*Suc a*) *c*) *b*) *in PrimRecOp g h*)

lemma *c-f-list-at-0*: *c-f-list f 0 x = c-cons* (*f 0 x*) *0* **by** (*simp add*: *c-f-list-def*

Let-def)

lemma *c-f-list-at-Suc*: *c-f-list f* (*Suc y*) *x* = *c-cons* (*f* (*Suc y*) *x*) (*c-f-list f y x*) **by** ((*simp add*: *c-f-list-def Let-def*))

lemma *c-f-list-is-pr*: *f* ∈ *PrimRec2* ⟹ *c-f-list f* ∈ *PrimRec2*
proof −
  assume *A1*: *f* ∈ *PrimRec2*
  let *?g* = (%*x*. *c-cons* (*f 0 x*) *0*)
  from *A1 c-cons-is-pr* **have** *S1*: *?g* ∈ *PrimRec1* **by** *prec*
  let *?h* = (%*a b c*. *c-cons* (*f* (*Suc a*) *c*) *b*)
  from *A1 c-cons-is-pr* **have** *S2*: *?h* ∈ *PrimRec3* **by** *prec*
  from *S1 S2* **show** *?thesis* **by** (*simp add*: *pr-rec c-f-list-def Let-def*)
qed

lemma *c-f-list-to-f-0*: *f y x* = *c-hd* (*c-f-list f y x*)
apply(*induct y*)
apply(*simp add*: *c-f-list-at-0*)
apply(*simp add*: *c-f-list-at-Suc*)
done

lemma *c-f-list-to-f*: *f* = (λ *y x*. *c-hd* (*c-f-list f y x*))
apply(*rule ext*, *rule ext*)
apply(*rule c-f-list-to-f-0*)
done

lemma *c-f-list-f-is-pr*: *c-f-list f* ∈ *PrimRec2* ⟹ *f* ∈ *PrimRec2*
proof −
  assume *A1*: *c-f-list f* ∈ *PrimRec2*
  have *S1*: *f* = (λ *y x*. *c-hd* (*c-f-list f y x*)) **by** (*rule c-f-list-to-f*)
  from *A1 c-hd-is-pr* **have** *S2*: (λ *y x*. *c-hd* (*c-f-list f y x*)) ∈ *PrimRec2* **by** *prec*
  with *S1* **show** *?thesis* **by** *simp*
qed

lemma *c-f-list-lm-1*: *c-nth* (*c-cons x y*) (*Suc z*) = *c-nth y z* **by** (*simp add*: *c-nth-def c-drop-at-Suc1*)

lemma *c-f-list-lm-2*: *z* < *Suc n* ⟹ *c-nth* (*c-f-list f* (*Suc n*) *x*) (*Suc n* − *z*) = *c-nth* (*c-f-list f n x*) (*n* − *z*)
proof −
  assume *z* < *Suc n*
  then have *Suc n* − *z* = *Suc* (*n*−*z*) **by** *arith*
  then have *c-nth* (*c-f-list f* (*Suc n*) *x*) (*Suc n* − *z*) = *c-nth* (*c-f-list f* (*Suc n*) *x*) (*Suc* (*n* − *z*)) **by** *simp*
  also have ... = *c-nth* (*c-cons* (*f* (*Suc n*) *x*) (*c-f-list f n x*)) (*Suc* (*n* − *z*)) **by** (*simp add*: *c-f-list-at-Suc*)
  also have ... = *c-nth* (*c-f-list f n x*) (*n* − *z*) **by** (*simp add*: *c-f-list-lm-1*)
  finally show *?thesis* **by** *simp*
qed

```
lemma c-f-list-nth: z ≤ y ⟶ c-nth (c-f-list f y x) (y−z) = f z x
proof (induct y)
  show z ≤ 0 ⟶ c-nth (c-f-list f 0 x) (0 − z) = f z x
  proof
    assume z ≤ 0 then have A1: z=0 by simp
    then have c-nth (c-f-list f 0 x) (0 − z) = c-nth (c-f-list f 0 x) 0 by simp
    also have ... = c-hd (c-f-list f 0 x) by (simp add: c-nth-at-0)
    also have ... = c-hd (c-cons (f 0 x) 0) by (simp add: c-f-list-at-0)
    also have ... = f 0 x by simp
    finally show c-nth (c-f-list f 0 x) (0 − z) = f z x by (simp add: A1)
  qed
next
  fix n assume A2:  z ≤ n ⟶ c-nth (c-f-list f n x) (n − z) = f z x show z ≤ Suc n ⟶ c-nth (c-f-list f (Suc n) x) (Suc n − z) = f z x
  proof
    assume A3: z ≤ Suc n
    show  z ≤ Suc n ⟹ c-nth (c-f-list f (Suc n) x) (Suc n − z) = f z x
    proof cases
      assume AA1: z ≤ n
      then have AA2: z < Suc n by simp
      from A2 this have S1: c-nth (c-f-list f n x) (n − z) = f z x by auto
      from AA2 have c-nth (c-f-list f (Suc n) x) (Suc n − z) = c-nth (c-f-list f n x) (n − z) by (rule c-f-list-lm-2)
      with S1 show c-nth (c-f-list f (Suc n) x) (Suc n − z) = f z x by simp
    next
      assume ¬ z ≤ n
      from A3 this have S1: z = Suc n by simp
      then have S2: Suc n − z = 0 by simp
      then have c-nth (c-f-list f (Suc n) x) (Suc n − z) = c-nth (c-f-list f (Suc n) x) 0 by simp
      also have ... = c-hd (c-f-list f (Suc n) x) by (simp add: c-nth-at-0)
       also have ... = c-hd (c-cons (f (Suc n) x) (c-f-list f n x)) by (simp add: c-f-list-at-Suc)
      also have ... = f (Suc n) x by simp
      finally show c-nth (c-f-list f (Suc n) x) (Suc n − z) = f z x by (simp add: S1)
    qed
  qed
qed

theorem th-pr-rec: ⟦ g ∈ PrimRec1; h ∈ PrimRec3; (∀ x. (f 0 x) = (g x)); (∀ x y. (f (Suc y) x) = h y (f y x) x) ⟧ ⟹ f ∈ PrimRec2
proof −
  assume g-is-pr: g ∈ PrimRec1
  assume h-is-pr: h ∈ PrimRec3
  assume f-0: ∀ x. f 0 x = g x
  assume f-1: ∀ x y. (f (Suc y) x) = h y (f y x) x
  let ?f = PrimRecOp g h
```

```
  from g-is-pr h-is-pr have S1: ?f ∈ PrimRec2 by (rule pr-rec)
  have f-2:∀ x. ?f 0 x = g x by simp
  have f-3: ∀ x y. (?f (Suc y) x) = h y (?f y x) x by simp
  have S2: f = ?f
  proof −
    have ⋀ x y. f y x = ?f y x
    apply(induct-tac y)
    apply(insert f-0 f-1)
    apply(auto)
    done
    then show f = ?f by (simp add: ext)
  qed
  from S1 S2 show ?thesis by simp
qed

theorem th-rec: ⟦ g ∈ PrimRec1; α ∈ PrimRec2; h ∈ PrimRec3; (∀ x y. α y x ≤
y); (∀ x. (f 0 x) = (g x)); (∀ x y. (f (Suc y) x) = h y (f (α y x) x) x) ⟧ ⟹ f ∈
PrimRec2
proof −
  assume g-is-pr: g ∈ PrimRec1
  assume a-is-pr: α ∈ PrimRec2
  assume h-is-pr: h ∈ PrimRec3
  assume a-le: (∀ x y. α y x ≤ y)
  assume f-0: ∀ x. f 0 x = g x
  assume f-1: ∀ x y. (f (Suc y) x) = h y (f (α y x) x) x
  let ?g′ = λ x. c-cons (g x) 0
  let ?h′ = λ a b c. c-cons (h a (c-nth b (a − (α a c))) c) b
  let ?r = c-f-list f
  from g-is-pr c-cons-is-pr have g′-is-pr: ?g′ ∈ PrimRec1 by prec
  from h-is-pr c-cons-is-pr c-nth-is-pr a-is-pr have h′-is-pr: ?h′ ∈ PrimRec3 by
prec
  have S1: ∀ x. ?r 0 x = ?g′ x
  proof
    fix x have ?r 0 x = c-cons (f 0 x) 0 by (rule c-f-list-at-0)
    with f-0 have ?r 0 x = c-cons (g x) 0 by simp
    then show ?r 0 x = ?g′ x by simp
  qed
  have S2: ∀ x y. ?r (Suc y) x = ?h′ y (?r y x) x
  proof (rule allI, rule allI)
    fix x y show ?r (Suc y) x = ?h′ y (?r y x) x
    proof −
     have S2-1: ?r (Suc y) x = c-cons (f (Suc y) x) (?r y x) by (rule c-f-list-at-Suc)
      with f-1 have S2-2: f (Suc y) x = h y (f (α y x) x) x by simp
      from a-le have S2-3: α y x ≤ y by simp
       then have S2-4: f (α y x) x = c-nth (?r y x) (y−(α y x)) by (simp add:
c-f-list-nth)
      from S2-1 S2-2 S2-4 show ?thesis by simp
    qed
  qed
```

```
  from g′-is-pr h′-is-pr S1 S2 have S3: ?r ∈ PrimRec2 by (rule th-pr-rec)
  then show f ∈ PrimRec2 by (rule c-f-list-f-is-pr)
qed

declare c-tl-less [termination-simp]

fun c-assoc-have-key :: nat ⇒ nat ⇒ nat where
  c-assoc-have-key-df [simp del]: c-assoc-have-key y x = (if y = 0 then 1 else
    (if c-fst (c-hd y) = x then 0 else c-assoc-have-key (c-tl y) x))

lemma c-assoc-have-key-lm-1: y ≠ 0 ⟹ c-assoc-have-key y x = (if c-fst (c-hd y)
= x then 0 else c-assoc-have-key (c-tl y) x) by (simp add: c-assoc-have-key-df)

theorem c-assoc-have-key-is-pr: c-assoc-have-key ∈ PrimRec2
proof −
  let ?h = λ a b c. if c-fst (c-hd (Suc a)) = c then 0 else b
  let ?a = λ y x. c-tl (Suc y)
  let ?g = λ x. (1::nat)
  have g-is-pr: ?g ∈ PrimRec1 by (rule const-is-pr)
  from c-tl-is-pr have a-is-pr: ?a ∈ PrimRec2 by prec
  have h-is-pr: ?h ∈ PrimRec3
  proof (rule if-eq-is-pr3)
    from c-fst-is-pr c-hd-is-pr show (λx y z. c-fst (c-hd (Suc x))) ∈ PrimRec3 by
prec
  next
    show (λx y z. z) ∈ PrimRec3 by (rule pr-id3-3)
  next
    show (λx y z. 0) ∈ PrimRec3 by prec
  next
    show (λx y z. y) ∈ PrimRec3 by (rule pr-id3-2)
  qed
  have a-le: ∀ x y. ?a y x ≤ y
  proof (rule allI, rule allI)
    fix x y show ?a y x ≤ y
    proof −
      have Suc y > 0 by simp
      then have ?a y x < Suc y by (rule c-tl-less)
      then show ?thesis by simp
    qed
  qed
  have f-0: ∀ x. c-assoc-have-key 0 x = ?g x by (simp add: c-assoc-have-key-df)
  have f-1: ∀ x y. c-assoc-have-key (Suc y) x = ?h y (c-assoc-have-key (?a y x)
x) x by (simp add: c-assoc-have-key-df)
  from g-is-pr a-is-pr h-is-pr a-le f-0 f-1 show ?thesis by (rule th-rec)
qed

fun c-assoc-value :: nat ⇒ nat ⇒ nat where
  c-assoc-value-df [simp del]: c-assoc-value y x = (if y = 0 then 0 else
    (if c-fst (c-hd y) = x then c-snd (c-hd y) else c-assoc-value (c-tl y) x))
```

```
lemma c-assoc-value-lm-1: y ≠ 0 ⟹ c-assoc-value y x = (if c-fst (c-hd y) = x then c-snd (c-hd y) else c-assoc-value (c-tl y) x) by (simp add: c-assoc-value-df)
```

```
theorem c-assoc-value-is-pr: c-assoc-value ∈ PrimRec2
proof −
  let ?h = λ a b c. if c-fst (c-hd (Suc a)) = c then c-snd (c-hd (Suc a)) else b
  let ?a = λ y x. c-tl (Suc y)
  let ?g = λ x. (0::nat)
  have g-is-pr: ?g ∈ PrimRec1 by (rule const-is-pr)
  from c-tl-is-pr have a-is-pr: ?a ∈ PrimRec2 by prec
  have h-is-pr: ?h ∈ PrimRec3
  proof (rule if-eq-is-pr3)
    from c-fst-is-pr c-hd-is-pr show (λx y z. c-fst (c-hd (Suc x))) ∈ PrimRec3 by prec
  next
    show (λx y z. z) ∈ PrimRec3 by (rule pr-id3-3)
  next
    from c-snd-is-pr c-hd-is-pr show (λx y z. c-snd (c-hd (Suc x))) ∈ PrimRec3 by prec
  next
    show (λx y z. y) ∈ PrimRec3 by (rule pr-id3-2)
  qed
  have a-le: ∀ x y. ?a y x ≤ y
  proof (rule allI, rule allI)
    fix x y show ?a y x ≤ y
    proof −
      have Suc y > 0 by simp
      then have ?a y x < Suc y by (rule c-tl-less)
      then show ?thesis by simp
    qed
  qed
  have f-0: ∀ x. c-assoc-value 0 x = ?g x by (simp add: c-assoc-value-df)
  have f-1: ∀ x y. c-assoc-value (Suc y) x = ?h y (c-assoc-value (?a y x) x) x by (simp add: c-assoc-value-df)
  from g-is-pr a-is-pr h-is-pr a-le f-0 f-1 show ?thesis by (rule th-rec)
qed
```

```
lemma c-assoc-lm-1: c-assoc-have-key (c-cons (c-pair x y) z) x = 0
apply(simp add: c-assoc-have-key-df)
apply(simp add: c-cons-pos)
done
```

```
lemma c-assoc-lm-2: c-assoc-value (c-cons (c-pair x y) z) x = y
apply(simp add: c-assoc-value-df)
apply(rule impI)
apply(insert c-cons-pos [where x=(c-pair x y) and u=z])
apply(auto)
done
```

```
lemma c-assoc-lm-3: x1 ≠ x ⟹ c-assoc-have-key (c-cons (c-pair x y) z) x1 = c-assoc-have-key z x1
proof -
  assume A1: x1 ≠ x
  let ?ls = (c-cons (c-pair x y) z)
  have S1: ?ls ≠ 0 by (simp add: c-cons-pos)
  then have S2: c-assoc-have-key ?ls x1 = (if c-fst (c-hd ?ls) = x1 then 0 else c-assoc-have-key (c-tl ?ls) x1) (is - = ?R)  by (rule c-assoc-have-key-lm-1)
  have S3: c-fst (c-hd ?ls) = x by simp
  with A1 have S4: ¬ (c-fst (c-hd ?ls) = x1) by simp
  from S4 have S5: ?R = c-assoc-have-key (c-tl ?ls) x1 by (rule if-not-P)
  from S2 S5 show ?thesis by simp
qed

lemma c-assoc-lm-4: x1 ≠ x ⟹ c-assoc-value (c-cons (c-pair x y) z) x1 = c-assoc-value z x1
proof -
  assume A1: x1 ≠ x
  let ?ls = (c-cons (c-pair x y) z)
  have S1: ?ls ≠ 0 by (simp add: c-cons-pos)
  then have S2: c-assoc-value ?ls x1 = (if c-fst (c-hd ?ls) = x1 then c-snd (c-hd ?ls) else c-assoc-value (c-tl ?ls) x1) (is - = ?R)  by (rule c-assoc-value-lm-1)
  have S3: c-fst (c-hd ?ls) = x by simp
  with A1 have S4: ¬ (c-fst (c-hd ?ls) = x1) by simp
  from S4 have S5: ?R = c-assoc-value (c-tl ?ls) x1 by (rule if-not-P)
  from S2 S5 show ?thesis by simp
qed

end
```

4 Primitive recursive functions of one variable

```
theory PRecFun2
imports PRecFun
begin
```

4.1 Alternative definition of primitive recursive functions of one variable

```
definition
  UnaryRecOp :: (nat ⇒ nat) ⇒ (nat ⇒ nat) ⇒ (nat ⇒ nat) where
  UnaryRecOp = (λ g h. pr-conv-2-to-1 (PrimRecOp g (pr-conv-1-to-3 h)))

lemma unary-rec-into-pr: ⟦ g ∈ PrimRec1; h ∈ PrimRec1 ⟧ ⟹ UnaryRecOp g h ∈ PrimRec1 by (simp add: UnaryRecOp-def pr-conv-1-to-3-lm pr-conv-2-to-1-lm pr-rec)

definition
```

```
  c-f-pair :: (nat ⇒ nat) ⇒ (nat ⇒ nat) ⇒ (nat ⇒ nat) where
  c-f-pair = (λ f g x. c-pair (f x) (g x))

lemma c-f-pair-to-pr: ⟦ f ∈ PrimRec1; g ∈ PrimRec1 ⟧ ⟹ c-f-pair f g ∈ PrimRec1
  unfolding c-f-pair-def by prec

inductive-set PrimRec1' :: (nat ⇒ nat) set
where
  zero: (λ x. 0) ∈ PrimRec1'
  | suc:  Suc ∈ PrimRec1'
  | fst:  c-fst ∈ PrimRec1'
  | snd:  c-snd ∈ PrimRec1'
  | comp: ⟦ f ∈ PrimRec1'; g ∈ PrimRec1' ⟧ ⟹ (λ x. f (g x)) ∈ PrimRec1'
  | pair: ⟦ f ∈ PrimRec1'; g ∈ PrimRec1' ⟧ ⟹ c-f-pair f g ∈ PrimRec1'
  | un-rec: ⟦ f ∈ PrimRec1'; g ∈ PrimRec1' ⟧ ⟹ UnaryRecOp f g ∈ PrimRec1'

lemma primrec'-into-primrec: f ∈ PrimRec1' ⟹ f ∈ PrimRec1
proof (induct f rule: PrimRec1'.induct)
  case zero show ?case by (rule pr-zero)
next
  case suc show ?case by (rule pr-suc)
next
  case fst show ?case by (rule c-fst-is-pr)
next
  case snd show ?case by (rule c-snd-is-pr)
next
  case comp from comp show ?case by (simp add: pr-comp1-1)
next
  case pair from pair show ?case by (simp add: c-f-pair-to-pr)
next
  case un-rec from un-rec show ?case by (simp add: unary-rec-into-pr)
qed

lemma pr-id1-1': (λ x. x) ∈ PrimRec1'
proof -
 have c-f-pair c-fst c-snd ∈ PrimRec1' by (simp add: PrimRec1'.fst PrimRec1'.snd
PrimRec1'.pair)
  moreover have c-f-pair c-fst c-snd = (λ x. x) by (simp add: c-f-pair-def)
  ultimately show ?thesis by simp
qed

lemma pr-id2-1': pr-conv-2-to-1 (λ x y. x) ∈ PrimRec1' by (simp add: pr-conv-2-to-1-def
PrimRec1'.fst)

lemma pr-id2-2': pr-conv-2-to-1 (λ x y. y) ∈ PrimRec1' by (simp add: pr-conv-2-to-1-def
PrimRec1'.snd)

lemma pr-id3-1': pr-conv-3-to-1 (λ x y z. x) ∈ PrimRec1'
proof -
```

have *pr-conv-3-to-1* (λ *x y z. x*) = (λ*x. c-fst* (*c-fst x*)) **by** (*simp add*: *pr-conv-3-to-1-def*)
moreover from *PrimRec1'.fst PrimRec1'.fst* **have** (λ*x. c-fst* (*c-fst x*)) ∈ *PrimRec1'* **by** (*rule PrimRec1'.comp*)
ultimately show *?thesis* **by** *simp*
qed

lemma *pr-id3-2'*: *pr-conv-3-to-1* (λ *x y z. y*) ∈ *PrimRec1'*
proof −
have *pr-conv-3-to-1* (λ *x y z. y*) = (λ*x. c-snd* (*c-fst x*)) **by** (*simp add*: *pr-conv-3-to-1-def*)
moreover from *PrimRec1'.snd PrimRec1'.fst* **have** (λ*x. c-snd* (*c-fst x*)) ∈ *PrimRec1'* **by** (*rule PrimRec1'.comp*)
ultimately show *?thesis* **by** *simp*
qed

lemma *pr-id3-3'*: *pr-conv-3-to-1* (λ *x y z. z*) ∈ *PrimRec1'*
proof −
have *pr-conv-3-to-1* (λ *x y z. z*) = (λ*x. c-snd x*) **by** (*simp add*: *pr-conv-3-to-1-def*)
thus *?thesis* **by** (*simp add*: *PrimRec1'.snd*)
qed

lemma *pr-comp2-1'*: ⟦ *pr-conv-2-to-1 f* ∈ *PrimRec1'*; *g* ∈ *PrimRec1'*; *h* ∈ *PrimRec1'* ⟧ ⟹ (λ *x. f* (*g x*) (*h x*)) ∈ *PrimRec1'*
proof −
assume *A1*: *pr-conv-2-to-1 f* ∈ *PrimRec1'*
assume *A2*: *g* ∈ *PrimRec1'*
assume *A3*: *h* ∈ *PrimRec1'*
let *?f1* = *pr-conv-2-to-1 f*
have *S1*: (%*x. ?f1* ((*c-f-pair g h*) *x*)) = (λ *x. f* (*g x*) (*h x*)) **by** (*simp add*: *c-f-pair-def pr-conv-2-to-1-def*)
from *A2 A3* **have** *S2*: *c-f-pair g h* ∈ *PrimRec1'* **by** (*rule PrimRec1'.pair*)
from *A1 S2* **have** *S3*: (%*x. ?f1* ((*c-f-pair g h*) *x*)) ∈ *PrimRec1'* **by** (*rule PrimRec1'.comp*)
with *S1* **show** *?thesis* **by** *simp*
qed

lemma *pr-comp3-1'*: ⟦ *pr-conv-3-to-1 f* ∈ *PrimRec1'*; *g* ∈ *PrimRec1'*; *h* ∈ *PrimRec1'*; *k* ∈ *PrimRec1'* ⟧ ⟹ (λ *x. f* (*g x*) (*h x*) (*k x*)) ∈ *PrimRec1'*
proof −
assume *A1*: *pr-conv-3-to-1 f* ∈ *PrimRec1'*
assume *A2*: *g* ∈ *PrimRec1'*
assume *A3*: *h* ∈ *PrimRec1'*
assume *A4*: *k* ∈ *PrimRec1'*
from *A2 A3* **have** *c-f-pair g h* ∈ *PrimRec1'* **by** (*rule PrimRec1'.pair*)
from *this A4* **have** *c-f-pair* (*c-f-pair g h*) *k* ∈ *PrimRec1'* **by** (*rule PrimRec1'.pair*)
from *A1 this* **have** (%*x.* (*pr-conv-3-to-1 f*) ((*c-f-pair* (*c-f-pair g h*) *k*) *x*)) ∈ *PrimRec1'* **by** (*rule PrimRec1'.comp*)
then show *?thesis* **by** (*simp add*: *c-f-pair-def pr-conv-3-to-1-def*)
qed

```
lemma pr-comp1-2': ⟦f ∈ PrimRec1'; pr-conv-2-to-1 g ∈ PrimRec1'⟧ ⟹ pr-conv-2-to-1
(λ x y. f (g x y)) ∈ PrimRec1'
proof −
  assume f ∈ PrimRec1'
  and pr-conv-2-to-1 g ∈ PrimRec1' (is ?g1 ∈ PrimRec1')
  then have (λ x. f (?g1 x)) ∈ PrimRec1' by (rule PrimRec1'.comp)
  then show ?thesis by (simp add: pr-conv-2-to-1-def)
qed
```

```
lemma pr-comp1-3': ⟦f ∈ PrimRec1'; pr-conv-3-to-1 g ∈ PrimRec1'⟧ ⟹ pr-conv-3-to-1
(λ x y z. f (g x y z)) ∈ PrimRec1'
proof −
  assume f ∈ PrimRec1'
  and pr-conv-3-to-1 g ∈ PrimRec1' (is ?g1 ∈ PrimRec1')
  then have (λ x. f (?g1 x)) ∈ PrimRec1' by (rule PrimRec1'.comp)
  then show ?thesis by (simp add: pr-conv-3-to-1-def)
qed
```

```
lemma pr-comp2-2': ⟦ pr-conv-2-to-1 f ∈ PrimRec1'; pr-conv-2-to-1 g ∈ Prim-
Rec1'; pr-conv-2-to-1 h ∈ PrimRec1' ⟧ ⟹ pr-conv-2-to-1 (λ x y. f (g x y) (h x
y)) ∈ PrimRec1'
proof −
  assume pr-conv-2-to-1 f ∈ PrimRec1'
  and pr-conv-2-to-1 g ∈ PrimRec1' (is ?g1 ∈ PrimRec1')
  and pr-conv-2-to-1 h ∈ PrimRec1' (is ?h1 ∈ PrimRec1')
  then have (λ x. f (?g1 x) (?h1 x)) ∈ PrimRec1' by (rule pr-comp2-1')
  then show ?thesis by (simp add: pr-conv-2-to-1-def)
qed
```

```
lemma pr-comp2-3': ⟦ pr-conv-2-to-1 f ∈ PrimRec1'; pr-conv-3-to-1 g ∈ Prim-
Rec1'; pr-conv-3-to-1 h ∈ PrimRec1' ⟧ ⟹ pr-conv-3-to-1 (λ x y z. f (g x y z) (h
x y z)) ∈ PrimRec1'
proof −
  assume pr-conv-2-to-1 f ∈ PrimRec1'
  and pr-conv-3-to-1 g ∈ PrimRec1' (is ?g1 ∈ PrimRec1')
  and pr-conv-3-to-1 h ∈ PrimRec1' (is ?h1 ∈ PrimRec1')
  then have (λ x. f (?g1 x) (?h1 x)) ∈ PrimRec1' by (rule pr-comp2-1')
  then show ?thesis by (simp add: pr-conv-3-to-1-def)
qed
```

```
lemma pr-comp3-2': ⟦ pr-conv-3-to-1 f ∈ PrimRec1'; pr-conv-2-to-1 g ∈ Prim-
Rec1'; pr-conv-2-to-1 h ∈ PrimRec1'; pr-conv-2-to-1 k ∈ PrimRec1' ⟧ ⟹ pr-conv-2-to-1
(λ x y. f (g x y) (h x y) (k x y)) ∈ PrimRec1'
proof −
  assume pr-conv-3-to-1 f ∈ PrimRec1'
  and pr-conv-2-to-1 g ∈ PrimRec1' (is ?g1 ∈ PrimRec1')
  and pr-conv-2-to-1 h ∈ PrimRec1' (is ?h1 ∈ PrimRec1')
  and pr-conv-2-to-1 k ∈ PrimRec1' (is ?k1 ∈ PrimRec1')
  then have (λ x. f (?g1 x) (?h1 x) (?k1 x)) ∈ PrimRec1' by (rule pr-comp3-1')
```

then show *?thesis* **by** (*simp add*: *pr-conv-2-to-1-def*)
qed

lemma *pr-comp3-3′*: ⟦ *pr-conv-3-to-1 f* ∈ *PrimRec1′*; *pr-conv-3-to-1 g* ∈ *PrimRec1′*; *pr-conv-3-to-1 h* ∈ *PrimRec1′*; *pr-conv-3-to-1 k* ∈ *PrimRec1′* ⟧ ⟹ *pr-conv-3-to-1* (λ *x y z*. *f* (*g x y z*) (*h x y z*) (*k x y z*)) ∈ *PrimRec1′*
proof −
assume *pr-conv-3-to-1 f* ∈ *PrimRec1′*
and *pr-conv-3-to-1 g* ∈ *PrimRec1′* (**is** *?g1* ∈ *PrimRec1′*)
and *pr-conv-3-to-1 h* ∈ *PrimRec1′* (**is** *?h1* ∈ *PrimRec1′*)
and *pr-conv-3-to-1 k* ∈ *PrimRec1′* (**is** *?k1* ∈ *PrimRec1′*)
then have (λ *x*. *f* (*?g1 x*) (*?h1 x*) (*?k1 x*)) ∈ *PrimRec1′* **by** (*rule pr-comp3-1′*)
then show *?thesis* **by** (*simp add*: *pr-conv-3-to-1-def*)
qed

lemma *lm′*: (*f1* ∈ *PrimRec1* ⟶ *f1* ∈ *PrimRec1′*) ∧ (*g1* ∈ *PrimRec2* ⟶ *pr-conv-2-to-1 g1* ∈ *PrimRec1′*) ∧ (*h1* ∈ *PrimRec3* ⟶ *pr-conv-3-to-1 h1* ∈ *PrimRec1′*)
proof (*induct rule*: *PrimRec1-PrimRec2-PrimRec3.induct*)
case *zero* **show** *?case* **by** (*rule PrimRec1′.zero*)
next case *suc* **show** *?case* **by** (*rule PrimRec1′.suc*)
next case *id1-1* **show** *?case* **by** (*rule pr-id1-1′*)
next case *id2-1* **show** *?case* **by** (*rule pr-id2-1′*)
next case *id2-2* **show** *?case* **by** (*rule pr-id2-2′*)
next case *id3-1* **show** *?case* **by** (*rule pr-id3-1′*)
next case *id3-2* **show** *?case* **by** (*rule pr-id3-2′*)
next case *id3-3* **show** *?case* **by** (*rule pr-id3-3′*)
next case *comp1-1* **from** *comp1-1* **show** *?case* **by** (*simp add*: *PrimRec1′.comp*)
next case *comp1-2* **from** *comp1-2* **show** *?case* **by** (*simp add*: *pr-comp1-2′*)
next case *comp1-3* **from** *comp1-3* **show** *?case* **by** (*simp add*: *pr-comp1-3′*)
next case *comp2-1* **from** *comp2-1* **show** *?case* **by** (*simp add*: *pr-comp2-1′*)
next case *comp2-2* **from** *comp2-2* **show** *?case* **by** (*simp add*: *pr-comp2-2′*)
next case *comp2-3* **from** *comp2-3* **show** *?case* **by** (*simp add*: *pr-comp2-3′*)
next case *comp3-1* **from** *comp3-1* **show** *?case* **by** (*simp add*: *pr-comp3-1′*)
next case *comp3-2* **from** *comp3-2* **show** *?case* **by** (*simp add*: *pr-comp3-2′*)
next case *comp3-3* **from** *comp3-3* **show** *?case* **by** (*simp add*: *pr-comp3-3′*)
next case *prim-rec*
fix *g h* **assume** *A1*: *g* ∈ *PrimRec1′* **and** *pr-conv-3-to-1 h* ∈ *PrimRec1′*
then have *UnaryRecOp g* (*pr-conv-3-to-1 h*) ∈ *PrimRec1′* **by** (*rule PrimRec1′.un-rec*)
moreover have *UnaryRecOp g* (*pr-conv-3-to-1 h*) = *pr-conv-2-to-1* (*PrimRecOp g h*) **by** (*simp add*: *UnaryRecOp-def*)
ultimately show *pr-conv-2-to-1* (*PrimRecOp g h*) ∈ *PrimRec1′* **by** *simp*
qed

theorem *pr-1-eq-1′*: *PrimRec1* = *PrimRec1′*
proof −
have *S1*: ⋀ *f*. *f* ∈ *PrimRec1* ⟶ *f* ∈ *PrimRec1′* **by** (*simp add*: *lm′*)
have *S2*: ⋀ *f*. *f* ∈ *PrimRec1′* ⟶ *f* ∈ *PrimRec1* **by** (*simp add*: *primrec′-into-primrec*)
from *S1 S2* **show** *?thesis* **by** *blast*
qed

4.2 The scheme datatype

```
datatype PrimScheme = Base-zero | Base-suc | Base-fst | Base-snd
                    | Comp-op PrimScheme PrimScheme
                    | Pair-op PrimScheme PrimScheme
                    | Rec-op PrimScheme PrimScheme

primrec
  sch-to-pr :: PrimScheme ⇒ (nat ⇒ nat)
where
  sch-to-pr Base-zero = (λ x. 0)
| sch-to-pr Base-suc = Suc
| sch-to-pr Base-fst = c-fst
| sch-to-pr Base-snd = c-snd
| sch-to-pr (Comp-op t1 t2)  = (λ x. (sch-to-pr t1) ((sch-to-pr t2) x))
| sch-to-pr (Pair-op t1 t2)  = c-f-pair (sch-to-pr t1) (sch-to-pr t2)
| sch-to-pr (Rec-op t1 t2)  = UnaryRecOp (sch-to-pr t1) (sch-to-pr t2)

lemma sch-to-pr-into-pr: sch-to-pr sch ∈ PrimRec1 by (simp add: pr-1-eq-1',
induct sch, simp-all add: PrimRec1'.intros)

lemma sch-to-pr-srj: f ∈ PrimRec1 ⟹ (∃ sch. f = sch-to-pr sch)
proof −
 assume f ∈ PrimRec1 then have A1: f ∈ PrimRec1' by (simp add: pr-1-eq-1')
 from A1 show ?thesis
 proof (induct f rule: PrimRec1'.induct)
   have (λ x. 0) = sch-to-pr Base-zero by simp
   then show ∃sch. (λu. 0) = sch-to-pr sch by (rule exI)
 next
   have Suc = sch-to-pr Base-suc by simp
   then show ∃sch. Suc = sch-to-pr sch by (rule exI)
 next
   have c-fst = sch-to-pr Base-fst by simp
   then show ∃sch. c-fst = sch-to-pr sch by (rule exI)
 next
   have c-snd = sch-to-pr Base-snd by simp
   then show ∃sch. c-snd = sch-to-pr sch by (rule exI)
 next
   fix f1 f2 assume B1: ∃sch. f1 = sch-to-pr sch and B2: ∃sch. f2 = sch-to-pr
sch
   from B1 obtain sch1 where S1: f1 = sch-to-pr sch1 ..
   from B2 obtain sch2 where S2: f2 = sch-to-pr sch2 ..
   from S1 S2 have (λ x. f1 (f2 x)) = sch-to-pr (Comp-op sch1 sch2) by simp
   then show ∃sch. (λx. f1 (f2 x)) = sch-to-pr sch by (rule exI)
 next
   fix f1 f2 assume B1: ∃sch. f1 = sch-to-pr sch and B2: ∃sch. f2 = sch-to-pr
sch
   from B1 obtain sch1 where S1: f1 = sch-to-pr sch1 ..
   from B2 obtain sch2 where S2: f2 = sch-to-pr sch2 ..
   from S1 S2 have c-f-pair f1 f2 = sch-to-pr (Pair-op sch1 sch2) by simp
```

```
    then show ∃ sch. c-f-pair f1 f2 = sch-to-pr sch by (rule exI)
  next
    fix f1 f2 assume B1: ∃ sch. f1 = sch-to-pr sch and B2: ∃ sch. f2 = sch-to-pr
sch
    from B1 obtain sch1 where S1: f1 = sch-to-pr sch1 ..
    from B2 obtain sch2 where S2: f2 = sch-to-pr sch2 ..
    from S1 S2 have UnaryRecOp f1 f2 = sch-to-pr (Rec-op sch1 sch2) by simp
    then show ∃ sch. UnaryRecOp f1 f2 = sch-to-pr sch by (rule exI)
  qed
qed

definition
  loc-f :: nat ⇒ PrimScheme ⇒ PrimScheme ⇒ PrimScheme where
  loc-f n sch1 sch2 =
    (if n=0 then Base-zero else
     if n=1 then Base-suc else
     if n=2 then Base-fst else
     if n=3 then Base-snd else
     if n=4 then (Comp-op sch1 sch2) else
     if n=5 then (Pair-op sch1 sch2) else
     if n=6 then (Rec-op sch1 sch2) else
     Base-zero)

definition
  mod7 :: nat ⇒ nat where
  mod7 = (λ x. x mod 7)

lemma c-snd-snd-lt [termination-simp]: c-snd (c-snd (Suc (Suc x))) < Suc (Suc
x)
proof −
  let ?y = Suc (Suc x)
  have ?y > 1 by simp
  then have c-snd ?y < ?y by (rule c-snd-less-arg)
  moreover have c-snd (c-snd ?y) ≤ c-snd ?y by (rule c-snd-le-arg)
  ultimately show ?thesis by simp
qed

lemma c-fst-snd-lt [termination-simp]: c-fst (c-snd (Suc (Suc x))) < Suc (Suc x)
proof −
  let ?y = Suc (Suc x)
  have ?y > 1 by simp
  then have c-snd ?y < ?y by (rule c-snd-less-arg)
  moreover have c-fst (c-snd ?y) ≤ c-snd ?y by (rule c-fst-le-arg)
  ultimately show ?thesis by simp
qed

fun nat-to-sch :: nat ⇒ PrimScheme where
  nat-to-sch 0 = Base-zero
| nat-to-sch (Suc 0) = Base-zero
```

```
| nat-to-sch x = (let u=mod7 (c-fst x); v=c-snd x; v1=c-fst v; v2 = c-snd v;
sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2)
```

```
primrec sch-to-nat :: PrimScheme ⇒ nat where
  sch-to-nat Base-zero = 0
| sch-to-nat Base-suc = c-pair 1 0
| sch-to-nat Base-fst = c-pair 2 0
| sch-to-nat Base-snd = c-pair 3 0
| sch-to-nat (Comp-op t1 t2) = c-pair 4 (c-pair (sch-to-nat t1) (sch-to-nat t2))
| sch-to-nat (Pair-op t1 t2) = c-pair 5 (c-pair (sch-to-nat t1) (sch-to-nat t2))
| sch-to-nat (Rec-op t1 t2)  = c-pair 6 (c-pair (sch-to-nat t1) (sch-to-nat t2))
```

```
lemma loc-srj-lm-1: nat-to-sch (Suc (Suc x)) = (let u=mod7 (c-fst (Suc (Suc x)));
v=c-snd (Suc (Suc x)); v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch
v2 in loc-f u sch1 sch2) by simp
```

```
lemma loc-srj-lm-2: x > 1 ⟹ nat-to-sch x = (let u=mod7 (c-fst x); v=c-snd x;
v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1
sch2)
proof −
  assume A1: x > 1
  let ?y = x−(2::nat)
  from A1 have S1: x = Suc (Suc ?y) by arith
   have S2: nat-to-sch (Suc (Suc ?y)) = (let u=mod7 (c-fst (Suc (Suc ?y)));
v=c-snd (Suc (Suc ?y)); v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch
v2 in loc-f u sch1 sch2) by (rule loc-srj-lm-1)
  from S1 S2 show ?thesis by simp
qed
```

```
lemma loc-srj-0: nat-to-sch (c-pair 1 0) = Base-suc
proof −
  let ?x = c-pair 1 0
  have S1: ?x = 2 by (simp add: c-pair-def sf-def)
  then have S2: ?x = Suc (Suc 0) by simp
  let ?y = Suc (Suc 0)
  have S3: nat-to-sch ?y = (let u=mod7 (c-fst ?y); v=c-snd ?y; v1=c-fst v; v2 =
c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is - = ?R)
by (rule loc-srj-lm-1)
  have S4: c-fst ?y = 1
  proof −
    from S2 have c-fst ?y = c-fst ?x by simp
    then show ?thesis by simp
  qed
  have S5: c-snd ?y = 0
  proof −
    from S2 have c-snd ?y = c-snd ?x by simp
    then show ?thesis by simp
  qed
  from S4 have S6: mod7 (c-fst ?y) = 1 by (simp add: mod7-def)
```

```
  from S3 S5 S6 have S9: ?R = loc-f 1 Base-zero Base-zero by (simp add: Let-def
c-fst-at-0 c-snd-at-0)
  then have S10: ?R = Base-suc by (simp add: loc-f-def)
  with S3 have S11: nat-to-sch ?y = Base-suc by simp
  from S2 this show ?thesis by simp
qed

lemma nat-to-sch-at-2: nat-to-sch 2 = Base-suc
proof -
  have S1: c-pair 1 0 = 2 by (simp add: c-pair-def sf-def)
  have S2: nat-to-sch (c-pair 1 0) = Base-suc by (rule loc-srj-0)
  from S1 S2 show ?thesis by simp
qed

lemma loc-srj-1: nat-to-sch (c-pair 2 0) = Base-fst
proof -
  let ?x = c-pair 2 0
  have S1: ?x = 5 by (simp add: c-pair-def sf-def)
  then have S2: ?x = Suc (Suc 3) by simp
  let ?y = Suc (Suc 3)
  have S3: nat-to-sch ?y = (let u=mod7 (c-fst ?y); v=c-snd ?y; v1=c-fst v; v2 =
c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is - = ?R)
by (rule loc-srj-lm-1)
  have S4: c-fst ?y = 2
  proof -
    from S2 have c-fst ?y = c-fst ?x by simp
    then show ?thesis by simp
  qed
  have S5: c-snd ?y = 0
  proof -
    from S2 have c-snd ?y = c-snd ?x by simp
    then show ?thesis by simp
  qed
  from S4 have S6: mod7 (c-fst ?y) = 2 by (simp add: mod7-def)
  from S3 S5 S6 have S9: ?R = loc-f 2 Base-zero Base-zero by (simp add: Let-def
c-fst-at-0 c-snd-at-0)
  then have S10: ?R = Base-fst by (simp add: loc-f-def)
  with S3 have S11: nat-to-sch ?y = Base-fst by simp
  from S2 this show ?thesis by simp
qed

lemma loc-srj-2: nat-to-sch (c-pair 3 0) = Base-snd
proof -
  let ?x = c-pair 3 0
  have S1: ?x > 1 by (simp add: c-pair-def sf-def)
  from S1 have S2: nat-to-sch ?x = (let u=mod7 (c-fst ?x); v=c-snd ?x; v1=c-fst
v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is
- = ?R) by (rule loc-srj-lm-2)
  have S3: c-fst ?x = 3 by simp
```

```
  have S4: c-snd ?x = 0 by simp
  from S3 have S6: mod7 (c-fst ?x) = 3 by (simp add: mod7-def)
  from S3 S4 S6 have S7: ?R = loc-f 3 Base-zero Base-zero by (simp add: Let-def c-fst-at-0 c-snd-at-0)
  then have S8: ?R = Base-snd by (simp add: loc-f-def)
  with S2 have S10: nat-to-sch ?x = Base-snd by simp
  from S2 this show ?thesis by simp
qed

lemma loc-srj-3: ⟦nat-to-sch (sch-to-nat sch1) = sch1; nat-to-sch (sch-to-nat sch2) = sch2⟧
        ⟹ nat-to-sch (c-pair 4 (c-pair (sch-to-nat sch1) (sch-to-nat sch2))) = Comp-op sch1 sch2
proof −
  assume A1: nat-to-sch (sch-to-nat sch1) = sch1
  assume A2: nat-to-sch (sch-to-nat sch2) = sch2
  let ?x = c-pair 4 (c-pair (sch-to-nat sch1) (sch-to-nat sch2))
  have S1: ?x > 1 by (simp add: c-pair-def sf-def)
  from S1 have S2: nat-to-sch ?x = (let u=mod7 (c-fst ?x); v=c-snd ?x; v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is - = ?R) by (rule loc-srj-lm-2)
  have S3: c-fst ?x = 4 by simp
  have S4: c-snd ?x = c-pair (sch-to-nat sch1) (sch-to-nat sch2) by simp
  from S3 have S5: mod7 (c-fst ?x) = 4 by (simp add: mod7-def)
  from A1 A2 S4 S5 have ?R = Comp-op sch1 sch2 by (simp add: Let-def c-fst-at-0 c-snd-at-0 loc-f-def)
  with S2 show ?thesis by simp
qed

lemma loc-srj-3-1: nat-to-sch (c-pair 4 (c-pair n1 n2)) = Comp-op (nat-to-sch n1) (nat-to-sch n2)
proof −
  let ?x = c-pair 4 (c-pair n1 n2)
  have S1: ?x > 1 by (simp add: c-pair-def sf-def)
  from S1 have S2: nat-to-sch ?x = (let u=mod7 (c-fst ?x); v=c-snd ?x; v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is - = ?R) by (rule loc-srj-lm-2)
  have S3: c-fst ?x = 4 by simp
  have S4: c-snd ?x = c-pair n1 n2 by simp
  from S3 have S5: mod7 (c-fst ?x) = 4 by (simp add: mod7-def)
  from S4 S5 have ?R = Comp-op (nat-to-sch n1) (nat-to-sch n2) by (simp add: Let-def c-fst-at-0 c-snd-at-0 loc-f-def)
  with S2 show ?thesis by simp
qed

lemma loc-srj-4: ⟦nat-to-sch (sch-to-nat sch1) = sch1; nat-to-sch (sch-to-nat sch2) = sch2⟧
        ⟹ nat-to-sch (c-pair 5 (c-pair (sch-to-nat sch1) (sch-to-nat sch2))) = Pair-op sch1 sch2
```

proof −
assume *A1*: *nat-to-sch* (*sch-to-nat sch1*) = *sch1*
assume *A2*: *nat-to-sch* (*sch-to-nat sch2*) = *sch2*
let *?x* = *c-pair 5* (*c-pair* (*sch-to-nat sch1*) (*sch-to-nat sch2*))
have *S1*: *?x* > *1* **by** (*simp add*: *c-pair-def sf-def*)
from *S1* **have** *S2*: *nat-to-sch ?x* = (*let u=mod7* (*c-fst ?x*); *v=c-snd ?x*; *v1=c-fst v*; *v2* = *c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) (**is** - = *?R*) **by** (*rule loc-srj-lm-2*)
have *S3*: *c-fst ?x* = *5* **by** *simp*
have *S4*: *c-snd ?x* = *c-pair* (*sch-to-nat sch1*) (*sch-to-nat sch2*) **by** *simp*
from *S3* **have** *S5*: *mod7* (*c-fst ?x*) = *5* **by** (*simp add*: *mod7-def*)
from *A1 A2 S4 S5* **have** *?R* = *Pair-op sch1 sch2* **by** (*simp add*: *Let-def c-fst-at-0 c-snd-at-0 loc-f-def*)
with *S2* **show** *?thesis* **by** *simp*
qed

lemma *loc-srj-4-1*: *nat-to-sch* (*c-pair 5* (*c-pair n1 n2*)) = *Pair-op* (*nat-to-sch n1*) (*nat-to-sch n2*)
proof −
let *?x* = *c-pair 5* (*c-pair n1 n2*)
have *S1*: *?x* > *1* **by** (*simp add*: *c-pair-def sf-def*)
from *S1* **have** *S2*: *nat-to-sch ?x* = (*let u=mod7* (*c-fst ?x*); *v=c-snd ?x*; *v1=c-fst v*; *v2* = *c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) (**is** - = *?R*) **by** (*rule loc-srj-lm-2*)
have *S3*: *c-fst ?x* = *5* **by** *simp*
have *S4*: *c-snd ?x* = *c-pair n1 n2* **by** *simp*
from *S3* **have** *S5*: *mod7* (*c-fst ?x*) = *5* **by** (*simp add*: *mod7-def*)
from *S4 S5* **have** *?R* = *Pair-op* (*nat-to-sch n1*) (*nat-to-sch n2*) **by** (*simp add*: *Let-def c-fst-at-0 c-snd-at-0 loc-f-def*)
with *S2* **show** *?thesis* **by** *simp*
qed

lemma *loc-srj-5*: ⟦*nat-to-sch* (*sch-to-nat sch1*) = *sch1*; *nat-to-sch* (*sch-to-nat sch2*) = *sch2*⟧
⟹ *nat-to-sch* (*c-pair 6* (*c-pair* (*sch-to-nat sch1*) (*sch-to-nat sch2*))) = *Rec-op sch1 sch2*
proof −
assume *A1*: *nat-to-sch* (*sch-to-nat sch1*) = *sch1*
assume *A2*: *nat-to-sch* (*sch-to-nat sch2*) = *sch2*
let *?x* = *c-pair 6* (*c-pair* (*sch-to-nat sch1*) (*sch-to-nat sch2*))
have *S1*: *?x* > *1* **by** (*simp add*: *c-pair-def sf-def*)
from *S1* **have** *S2*: *nat-to-sch ?x* = (*let u=mod7* (*c-fst ?x*); *v=c-snd ?x*; *v1=c-fst v*; *v2* = *c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) (**is** - = *?R*) **by** (*rule loc-srj-lm-2*)
have *S3*: *c-fst ?x* = *6* **by** *simp*
have *S4*: *c-snd ?x* = *c-pair* (*sch-to-nat sch1*) (*sch-to-nat sch2*) **by** *simp*
from *S3* **have** *S5*: *mod7* (*c-fst ?x*) = *6* **by** (*simp add*: *mod7-def*)
from *A1 A2 S4 S5* **have** *?R* = *Rec-op sch1 sch2* **by** (*simp add*: *Let-def c-fst-at-0 c-snd-at-0 loc-f-def*)

```
  with S2 show ?thesis by simp
qed
```

```
lemma loc-srj-5-1: nat-to-sch (c-pair 6 (c-pair n1 n2)) = Rec-op (nat-to-sch n1) (nat-to-sch n2)
proof -
  let ?x = c-pair 6 (c-pair n1 n2)
  have S1: ?x > 1 by (simp add: c-pair-def sf-def)
  from S1 have S2: nat-to-sch ?x = (let u=mod7 (c-fst ?x); v=c-snd ?x; v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) (is - = ?R) by (rule loc-srj-lm-2)
  have S3: c-fst ?x = 6 by simp
  have S4: c-snd ?x = c-pair n1 n2 by simp
  from S3 have S5: mod7 (c-fst ?x) = 6 by (simp add: mod7-def)
  from S4 S5 have ?R = Rec-op (nat-to-sch n1) (nat-to-sch n2) by (simp add: Let-def c-fst-at-0 c-snd-at-0 loc-f-def)
  with S2 show ?thesis by simp
qed
```

```
theorem nat-to-sch-srj: nat-to-sch (sch-to-nat sch) = sch
apply(induct sch, auto simp add: loc-srj-0 loc-srj-1 loc-srj-2 loc-srj-3 loc-srj-4 loc-srj-5)
apply(insert loc-srj-0)
apply(simp)
done
```

4.3 Indexes of primitive recursive functions of one variables

```
definition
  nat-to-pr :: nat ⇒ (nat ⇒ nat) where
  nat-to-pr = (λ x. sch-to-pr (nat-to-sch x))
```

```
theorem nat-to-pr-into-pr: nat-to-pr n ∈ PrimRec1 by (simp add: nat-to-pr-def sch-to-pr-into-pr)
```

```
lemma nat-to-pr-srj: f ∈ PrimRec1 ⟹ (∃ n. f = nat-to-pr n)
proof -
  assume f ∈ PrimRec1
  then have S1: (∃ t. f = sch-to-pr t) by (rule sch-to-pr-srj)
  from S1 obtain t where S2: f = sch-to-pr t ..
  let ?n = sch-to-nat t
  have S3: nat-to-pr ?n = sch-to-pr (nat-to-sch ?n) by (simp add: nat-to-pr-def)
  have S4: nat-to-sch ?n = t by (rule nat-to-sch-srj)
  from S3 S4 have S5: nat-to-pr ?n = sch-to-pr t by simp
  from S2 S5 have nat-to-pr ?n = f by simp
  then have f = nat-to-pr ?n by simp
  then show ?thesis ..
qed
```

lemma *nat-to-pr-at-0*: *nat-to-pr 0* = (λ *x*. *0*) **by** (*simp add*: *nat-to-pr-def*)

definition
index-of-pr :: (*nat* ⇒ *nat*) ⇒ *nat* **where**
index-of-pr f = (*SOME n*. *f* = *nat-to-pr n*)

theorem *index-of-pr-is-real*: *f* ∈ *PrimRec1* ⟹ *nat-to-pr* (*index-of-pr f*) = *f*
proof −
assume *f* ∈ *PrimRec1*
hence ∃ *n*. *f* = *nat-to-pr n* **by** (*rule nat-to-pr-srj*)
hence *f* = *nat-to-pr* (*SOME n*. *f* = *nat-to-pr n*) **by** (*rule someI-ex*)
thus *?thesis* **by** (*simp add*: *index-of-pr-def*)
qed

definition
comp-by-index :: *nat* ⇒ *nat* ⇒ *nat* **where**
comp-by-index = (λ *n1 n2*. *c-pair 4* (*c-pair n1 n2*))

definition
pair-by-index :: *nat* ⇒ *nat* ⇒ *nat* **where**
pair-by-index = (λ *n1 n2*. *c-pair 5* (*c-pair n1 n2*))

definition
rec-by-index :: *nat* ⇒ *nat* ⇒ *nat* **where**
rec-by-index = (λ *n1 n2*. *c-pair 6* (*c-pair n1 n2*))

lemma *comp-by-index-is-pr*: *comp-by-index* ∈ *PrimRec2*
unfolding *comp-by-index-def*
using *const-is-pr-2* [*of 4*] **by** *prec*

lemma *comp-by-index-inj*: *comp-by-index x1 y1* = *comp-by-index x2 y2* ⟹ *x1*=*x2* ∧ *y1*=*y2*
proof −
assume *comp-by-index x1 y1* = *comp-by-index x2 y2*
hence *c-pair 4* (*c-pair x1 y1*) = *c-pair 4* (*c-pair x2 y2*) **by** (*unfold comp-by-index-def*)
hence *c-pair x1 y1* = *c-pair x2 y2* **by** (*rule c-pair-inj2*)
thus *?thesis* **by** (*rule c-pair-inj*)
qed

lemma *comp-by-index-inj1*: *comp-by-index x1 y1* = *comp-by-index x2 y2* ⟹ *x1* = *x2* **by** (*frule comp-by-index-inj*, *drule conjunct1*)

lemma *comp-by-index-inj2*: *comp-by-index x1 y1* = *comp-by-index x2 y2* ⟹ *y1* = *y2* **by** (*frule comp-by-index-inj*, *drule conjunct2*)

lemma *comp-by-index-main*: *nat-to-pr* (*comp-by-index n1 n2*) = (λ *x*. (*nat-to-pr n1*) ((*nat-to-pr n2*) *x*)) **by** (*unfold comp-by-index-def*, *unfold nat-to-pr-def*, *simp add*: *loc-srj-3-1*)

lemma *pair-by-index-is-pr*: *pair-by-index* ∈ *PrimRec2* **by** (*unfold pair-by-index-def*, *insert const-is-pr-2* [**where** *?n*=(*5*::*nat*)], *prec*)

lemma *pair-by-index-inj*: *pair-by-index x1 y1* = *pair-by-index x2 y2* ⟹ *x1*=*x2* ∧ *y1*=*y2*
proof −
 assume *pair-by-index x1 y1* = *pair-by-index x2 y2*
 hence *c-pair 5* (*c-pair x1 y1*) = *c-pair 5* (*c-pair x2 y2*) **by** (*unfold pair-by-index-def*)
 hence *c-pair x1 y1* = *c-pair x2 y2* **by** (*rule c-pair-inj2*)
 thus *?thesis* **by** (*rule c-pair-inj*)
qed

lemma *pair-by-index-inj1*: *pair-by-index x1 y1* = *pair-by-index x2 y2* ⟹ *x1* = *x2* **by** (*frule pair-by-index-inj*, *drule conjunct1*)

lemma *pair-by-index-inj2*: *pair-by-index x1 y1* = *pair-by-index x2 y2* ⟹ *y1* = *y2* **by** (*frule pair-by-index-inj*, *drule conjunct2*)

lemma *pair-by-index-main*: *nat-to-pr* (*pair-by-index n1 n2*) = *c-f-pair* (*nat-to-pr n1*) (*nat-to-pr n2*) **by** (*unfold pair-by-index-def*, *unfold nat-to-pr-def*, *simp add*: *loc-srj-4-1*)

lemma *nat-to-sch-of-pair-by-index* [*simp*]: *nat-to-sch* (*pair-by-index n1 n2*) = *Pair-op* (*nat-to-sch n1*) (*nat-to-sch n2*)
 by (*simp add*: *pair-by-index-def loc-srj-4-1*)

lemma *rec-by-index-is-pr*: *rec-by-index* ∈ *PrimRec2* **by** (*unfold rec-by-index-def*, *insert const-is-pr-2* [**where** *?n*=(*6*::*nat*)], *prec*)

lemma *rec-by-index-inj*: *rec-by-index x1 y1* = *rec-by-index x2 y2* ⟹ *x1*=*x2* ∧ *y1*=*y2*
proof −
 assume *rec-by-index x1 y1* = *rec-by-index x2 y2*
 hence *c-pair 6* (*c-pair x1 y1*) = *c-pair 6* (*c-pair x2 y2*) **by** (*unfold rec-by-index-def*)
 hence *c-pair x1 y1* = *c-pair x2 y2* **by** (*rule c-pair-inj2*)
 thus *?thesis* **by** (*rule c-pair-inj*)
qed

lemma *rec-by-index-inj1*: *rec-by-index x1 y1* = *rec-by-index x2 y2* ⟹ *x1* = *x2* **by** (*frule rec-by-index-inj*, *drule conjunct1*)

lemma *rec-by-index-inj2*: *rec-by-index x1 y1* = *rec-by-index x2 y2* ⟹ *y1* = *y2* **by** (*frule rec-by-index-inj*, *drule conjunct2*)

lemma *rec-by-index-main*: *nat-to-pr* (*rec-by-index n1 n2*) = *UnaryRecOp* (*nat-to-pr n1*) (*nat-to-pr n2*) **by** (*unfold rec-by-index-def*, *unfold nat-to-pr-def*, *simp add*: *loc-srj-5-1*)

4.4 s-1-1 theorem for primitive recursive functions of one variable

definition
index-of-const :: *nat* ⇒ *nat* **where**
index-of-const = *PrimRecOp1 0* (λ *x y*. *c-pair 4* (*c-pair 2 y*))

lemma *index-of-const-is-pr*: *index-of-const* ∈ *PrimRec1*
proof −
have (λ *x y*. *c-pair* (*4*::*nat*) (*c-pair* (*2*::*nat*) *y*)) ∈ *PrimRec2* **by** (*insert const-is-pr-2* [**where** *?n*=(*4*::*nat*)], *prec*)
then show *?thesis* **by** (*simp add*: *index-of-const-def pr-rec1*)
qed

lemma *index-of-const-at-0*: *index-of-const 0* = *0* **by** (*simp add*: *index-of-const-def*)

lemma *index-of-const-at-suc*: *index-of-const* (*Suc u*) = *c-pair 4* (*c-pair 2* (*index-of-const u*)) **by** (*unfold index-of-const-def*, *induct u*, *auto*)

lemma *index-of-const-main*: *nat-to-pr* (*index-of-const n*) = (λ *x*. *n*) (**is** *?P n*)
proof (*induct n*)
show *?P 0* **by** (*simp add*: *index-of-const-at-0 nat-to-pr-at-0*)
next
fix *n* **assume** *?P n*
then show *?P* (*Suc n*) **by** ((*simp add*: *index-of-const-at-suc nat-to-sch-at-2 nat-to-pr-def loc-srj-3-1*))
qed

lemma *index-of-const-lm-1*: (*nat-to-pr* (*index-of-const n*)) *0* = *n* **by** (*simp add*: *index-of-const-main*)

lemma *index-of-const-inj*: *index-of-const n1* = *index-of-const n2* ⟹ *n1* = *n2*
proof −
assume *index-of-const n1* = *index-of-const n2*
then have (*nat-to-pr* (*index-of-const n1*)) *0* = (*nat-to-pr* (*index-of-const n2*)) *0* **by** *simp*
thus *?thesis* **by** (*simp add*: *index-of-const-lm-1*)
qed

definition *index-of-zero* = *sch-to-nat Base-zero*
definition *index-of-suc* = *sch-to-nat Base-suc*
definition *index-of-c-fst* = *sch-to-nat Base-fst*
definition *index-of-c-snd* = *sch-to-nat Base-snd*
definition *index-of-id* = *pair-by-index index-of-c-fst index-of-c-snd*

lemma *index-of-zero-main*: *nat-to-pr index-of-zero* = (λ *x*. *0*) **by** (*simp add*: *index-of-zero-def nat-to-pr-def*)

lemma *index-of-suc-main*: *nat-to-pr index-of-suc* = *Suc*
apply(*simp add*: *index-of-suc-def nat-to-pr-def*)

apply(*insert loc-srj-0*)
apply(*simp*)
done

lemma *index-of-c-fst-main*: *nat-to-pr index-of-c-fst* = *c-fst* **by** (*simp add*: *index-of-c-fst-def nat-to-pr-def loc-srj-1*)

lemma [*simp*]: *nat-to-sch index-of-c-fst* = *Base-fst* **by** (*unfold index-of-c-fst-def*, *rule nat-to-sch-srj*)

lemma *index-of-c-snd-main*: *nat-to-pr index-of-c-snd* = *c-snd* **by** (*simp add*: *index-of-c-snd-def nat-to-pr-def loc-srj-2*)

lemma [*simp*]: *nat-to-sch index-of-c-snd* = *Base-snd* **by** (*unfold index-of-c-snd-def*, *rule nat-to-sch-srj*)

lemma *index-of-id-main*: *nat-to-pr index-of-id* = (λ *x*. *x*) **by** (*simp add*: *index-of-id-def nat-to-pr-def c-f-pair-def*)

definition
index-of-c-pair-n :: *nat* ⇒ *nat* **where**
index-of-c-pair-n = (λ *n*. *pair-by-index* (*index-of-const n*) *index-of-id*)

lemma *index-of-c-pair-n-is-pr*: *index-of-c-pair-n* ∈ *PrimRec1*
proof −
have (λ *x*. *index-of-id*) ∈ *PrimRec1* **by** (*rule const-is-pr*)
with *pair-by-index-is-pr index-of-const-is-pr* **have** (λ *n*. *pair-by-index* (*index-of-const n*) *index-of-id*) ∈ *PrimRec1* **by** *prec*
then show *?thesis* **by** (*fold index-of-c-pair-n-def*)
qed

lemma *index-of-c-pair-n-main*: *nat-to-pr* (*index-of-c-pair-n n*) = (λ *x*. *c-pair n x*)
proof −
have *nat-to-pr* (*index-of-c-pair-n n*) = *nat-to-pr* (*pair-by-index* (*index-of-const n*) *index-of-id*) **by** (*simp add*: *index-of-c-pair-n-def*)
also have . . . = *c-f-pair* (*nat-to-pr* (*index-of-const n*)) (*nat-to-pr index-of-id*) **by** (*simp add*: *pair-by-index-main*)
also have . . . = *c-f-pair* (λ *x*. *n*) (λ *x*. *x*) **by** (*simp add*: *index-of-const-main index-of-id-main*)
finally show *?thesis* **by** (*simp add*: *c-f-pair-def*)
qed

lemma *index-of-c-pair-n-inj*: *index-of-c-pair-n x1* = *index-of-c-pair-n x2* ⟹ *x1*=*x2*
proof −
assume *index-of-c-pair-n x1* = *index-of-c-pair-n x2*
hence *pair-by-index* (*index-of-const x1*) *index-of-id* = *pair-by-index* (*index-of-const x2*) *index-of-id* **by** (*unfold index-of-c-pair-n-def*)
hence *index-of-const x1* = *index-of-const x2* **by** (*rule pair-by-index-inj1*)
thus *?thesis* **by** (*rule index-of-const-inj*)

qed

definition
s1-1 :: *nat* ⇒ *nat* ⇒ *nat* **where**
s1-1 = (λ *n x*. *comp-by-index n* (*index-of-c-pair-n x*))

lemma *s1-1-is-pr*: *s1-1* ∈ *PrimRec2* **by** (*unfold s1-1-def*, *insert comp-by-index-is-pr index-of-c-pair-n-is-pr*, *prec*)

theorem *s1-1-th*: (λ *y*. (*nat-to-pr n*) (*c-pair x y*)) = *nat-to-pr* (*s1-1 n x*)
proof −
have *nat-to-pr* (*s1-1 n x*) = *nat-to-pr* (*comp-by-index n* (*index-of-c-pair-n x*)) **by** (*simp add*: *s1-1-def*)
also have ... = (λ *z*. (*nat-to-pr n*) ((*nat-to-pr* (*index-of-c-pair-n x*)) *z*)) **by** (*simp add*: *comp-by-index-main*)
also have ... = (λ *z*. (*nat-to-pr n*) ((λ *u*. *c-pair x u*) *z*)) **by** (*simp add*: *index-of-c-pair-n-main*)
finally show *?thesis* **by** *simp*
qed

lemma *s1-1-inj*: *s1-1 x1 y1* = *s1-1 x2 y2* ⟹ *x1*=*x2* ∧ *y1*=*y2*
proof −
assume *s1-1 x1 y1* = *s1-1 x2 y2*
then have *comp-by-index x1* (*index-of-c-pair-n y1*) = *comp-by-index x2* (*index-of-c-pair-n y2*) **by** (*unfold s1-1-def*)
then have *S1*: *x1*=*x2* ∧ *index-of-c-pair-n y1* = *index-of-c-pair-n y2* **by** (*rule comp-by-index-inj*)
then have *S2*: *x1*=*x2* **..**
from *S1* **have** *index-of-c-pair-n y1* = *index-of-c-pair-n y2* **..**
then have *y1* = *y2* **by** (*rule index-of-c-pair-n-inj*)
with *S2* **show** *?thesis* **..**
qed

lemma *s1-1-inj1*: *s1-1 x1 y1* = *s1-1 x2 y2* ⟹ *x1*=*x2* **by** (*frule s1-1-inj*, *drule conjunct1*)

lemma *s1-1-inj2*: *s1-1 x1 y1* = *s1-1 x2 y2* ⟹ *y1*=*y2* **by** (*frule s1-1-inj*, *drule conjunct2*)

primrec
pr-index-enumerator :: *nat* ⇒ *nat* ⇒ *nat*
where
pr-index-enumerator n 0 = *n*
| *pr-index-enumerator n* (*Suc m*) = *comp-by-index index-of-id* (*pr-index-enumerator n m*)

theorem *pr-index-enumerator-is-pr*: *pr-index-enumerator* ∈ *PrimRec2*
proof −
define *g* **where** *g x* = *x* **for** *x* :: *nat*

have *g-is-pr*: *g* ∈ *PrimRec1* **by** (*unfold g-def*, *rule pr-id1-1*)
define *h* **where** *h a b c* = *comp-by-index index-of-id b* **for** *a b c* :: *nat*
from *comp-by-index-is-pr* **have** *h-is-pr*: *h* ∈ *PrimRec3* **unfolding** *h-def* **by** *prec*
let *?f* = *pr-index-enumerator*
from *g-def* **have** *f-at-0*: ∀ *x*. *?f x 0* = *g x* **by** *auto*
from *h-def* **have** *f-at-Suc*: ∀ *x y*. *?f x* (*Suc y*) = *h x* (*?f x y*) *y* **by** *auto*
from *g-is-pr h-is-pr f-at-0 f-at-Suc* **show** *?thesis* **by** (*rule pr-rec-last-scheme*)
qed

lemma *pr-index-enumerator-increase1*: *pr-index-enumerator n m* < *pr-index-enumerator* (*n*+*1*) *m*
proof (*induct m*)
show *pr-index-enumerator n 0* < *pr-index-enumerator* (*n* + *1*) *0* **by** *simp*
next fix *na* **assume** *A*: *pr-index-enumerator n na* < *pr-index-enumerator* (*n* + *1*) *na*
show *pr-index-enumerator n* (*Suc na*) < *pr-index-enumerator* (*n* + *1*) (*Suc na*)
proof −
let *?a* = *pr-index-enumerator n na*
let *?b* = *pr-index-enumerator* (*n*+*1*) *na*
have *S1*: *pr-index-enumerator n* (*Suc na*) = *comp-by-index index-of-id ?a* **by** *simp*
have *L1*: *pr-index-enumerator* (*n*+*1*) (*Suc na*) = *comp-by-index index-of-id ?b* **by** *simp*
from *A* **have** *c-pair index-of-id ?a* < *c-pair index-of-id ?b* **by** (*rule c-pair-strict-mono2*)
then have *c-pair 4* (*c-pair index-of-id ?a*) < *c-pair 4* (*c-pair index-of-id ?b*) **by** (*rule c-pair-strict-mono2*)
then have *comp-by-index index-of-id ?a* < *c-pair 4* (*c-pair index-of-id ?b*) **by** (*simp add*: *comp-by-index-def*)
then have *comp-by-index index-of-id ?a* < *comp-by-index index-of-id ?b* **by** (*simp add*: *comp-by-index-def*)
with *S1 L1* **show** *?thesis* **by** *auto*
qed
qed

lemma *pr-index-enumerator-increase2*: *pr-index-enumerator n m* < *pr-index-enumerator n* (*m* + *1*)
proof −
let *?a* = *pr-index-enumerator n m*
have *S1*: *pr-index-enumerator n* (*m* + *1*) = *comp-by-index index-of-id ?a* **by** *simp*
have *S2*: *comp-by-index index-of-id ?a* = *c-pair 4* (*c-pair index-of-id ?a*) **by** (*simp add*: *comp-by-index-def*)
have *S3*: *4* + *c-pair index-of-id ?a* ≤ *c-pair 4* (*c-pair index-of-id ?a*) **by** (*rule sum-le-c-pair*)
then have *S4*: *c-pair index-of-id ?a* < *c-pair 4* (*c-pair index-of-id ?a*) **by** *auto*
have *S5*: *?a* ≤ *c-pair index-of-id ?a* **by** (*rule arg2-le-c-pair*)
from *S4 S5* **have** *S6*: *?a* < *c-pair 4* (*c-pair index-of-id ?a*) **by** *auto*
with *S1 S2* **show** *?thesis* **by** *auto*
qed

```
lemma f-inc-mono: (∀ (x::nat). (f::nat⇒nat) x < f (x+1)) ⟹ ∀ (x::nat) (y::nat).
(x < y ⟶ f x < f y)
proof (rule allI, rule allI)
  fix x y assume A: ∀ (x::nat). f x < f (x+1) show  x < y ⟶ f x < f y
  proof
    assume A1: x < y
    have L1: ⋀ u v. f u < f (u + (v+1))
    proof −
      fix u v show f u < f (u + (v+1))
      proof (induct v)
        from A show f u < f (u + (0 + 1)) by auto
      next
        fix v n
        assume A2: f u < f (u + (n + 1))
        from A have S1: f (u + (n + 1)) < f (u + (Suc n + 1)) by auto
        from A2 S1 show  f u < f (u + (Suc n + 1)) by (rule less-trans)
      qed
    qed
  let ?v = (y − x) − 1
  from A1 have S2: y = x + (?v + 1) by auto
  have f x < f (x + (?v + 1)) by (rule L1)
  with S2 show f x < f y by auto
  qed
qed

lemma pr-index-enumerator-mono1: n1 < n2 ⟹ pr-index-enumerator n1 m <
pr-index-enumerator n2 m
proof −
  assume A: n1 < n2
  define f where f x = pr-index-enumerator x m for x
  have f-inc: ∀ x. f x < f (x+1)
  proof
   fix x show f x < f (x+1) by (unfold f-def, rule pr-index-enumerator-increase1)
  qed
  from f-inc have ∀ x y. (x < y ⟶ f x < f y) by (rule f-inc-mono)
  with A f-def show ?thesis by auto
qed

lemma pr-index-enumerator-mono2: m1 < m2 ⟹ pr-index-enumerator n m1 <
pr-index-enumerator n m2
proof −
  assume A: m1 < m2
  define f where f x = pr-index-enumerator n x for x
  have f-inc: ∀ x. f x < f (x+1)
  proof
   fix x show f x < f (x+1) by (unfold f-def, rule pr-index-enumerator-increase2)
  qed
  from f-inc have ∀ x y. (x < y ⟶ f x < f y) by (rule f-inc-mono)
```

```
  with A f-def show ?thesis by auto
qed

lemma f-mono-inj: ∀ (x::nat) (y::nat). (x < y ⟶ (f::nat⇒nat) x < f y) ⟹ ∀
(x::nat) (y::nat). (f x = f y ⟶ x = y)
proof (rule allI, rule allI)
  fix x y assume A: ∀ x y. x < y ⟶ f x < f y show f x = f y ⟶ x = y
  proof
    assume A1: f x = f y show x = y
    proof (rule ccontr)
      assume A2: x ≠ y show False
      proof cases
        assume A3: x < y
        from A A3 have f x < f y by auto
        with A1 show False by auto
      next
        assume ¬ x < y with A2 have A4: y < x by auto
        from A A4 have f y < f x by auto
        with A1 show False by auto
      qed
    qed
  qed
qed

theorem pr-index-enumerator-inj1: pr-index-enumerator n1 m = pr-index-enumerator
n2 m ⟹ n1 = n2
proof −
  assume A: pr-index-enumerator n1 m = pr-index-enumerator n2 m
  define f where f x = pr-index-enumerator x m for x
  have f-mono: ∀ x y. (x < y ⟶ f x < f y)
  proof (rule allI, rule allI)
   fix x y show x < y ⟶ f x < f y by (unfold f-def, simp add: pr-index-enumerator-mono1)
  qed
  from f-mono have ∀ x y. (f x = f y ⟶ x = y) by (rule f-mono-inj)
  with A f-def show ?thesis by auto
qed

theorem pr-index-enumerator-inj2: pr-index-enumerator n m1 = pr-index-enumerator
n m2 ⟹ m1 = m2
proof −
  assume A: pr-index-enumerator n m1 = pr-index-enumerator n m2
  define f where f x = pr-index-enumerator n x for x
  have f-mono: ∀ x y. (x < y ⟶ f x < f y)
  proof (rule allI, rule allI)
   fix x y show x < y ⟶ f x < f y by (unfold f-def, simp add: pr-index-enumerator-mono2)
  qed
  from f-mono have ∀ x y. (f x = f y ⟶ x = y) by (rule f-mono-inj)
  with A f-def show ?thesis by auto
qed
```

theorem *pr-index-enumerator-main*: *nat-to-pr n* = *nat-to-pr* (*pr-index-enumerator n m*)
proof (*induct m*)
  show *nat-to-pr n* = *nat-to-pr* (*pr-index-enumerator n 0*) **by** *simp*
next
  fix *na* **assume** *A*: *nat-to-pr n* = *nat-to-pr* (*pr-index-enumerator n na*)
  show *nat-to-pr n* = *nat-to-pr* (*pr-index-enumerator n* (*Suc na*))
  proof −
    let *?a* = *pr-index-enumerator n na*
    have *S1*: *pr-index-enumerator n* (*Suc na*) = *comp-by-index index-of-id ?a* **by** *simp*
    have *nat-to-pr* (*comp-by-index index-of-id ?a*) = (λ *x*. (*nat-to-pr index-of-id*) (*nat-to-pr ?a x*)) **by** (*rule comp-by-index-main*)
    with *index-of-id-main* **have** *nat-to-pr* (*comp-by-index index-of-id ?a*) = *nat-to-pr ?a* **by** *simp*
    with *A S1* **show** *?thesis* **by** *simp*
  qed
qed

end

5 Finite sets

theory *PRecFinSet*
imports *PRecFun*
begin

We introduce a particular mapping *nat-to-set* from natural numbers to finite sets of natural numbers and a particular mapping *set-to-nat* from finite sets of natural numbers to natural numbers. See [1] and [2] for more information.

definition
  c-in :: *nat* ⇒ *nat* ⇒ *nat* **where**
  c-in = (λ *x u*. (*u div* (*2* ^ *x*)) *mod 2*)

lemma *c-in-is-pr*: *c-in* ∈ *PrimRec2*
proof −
  from *mod-is-pr power-is-pr div-is-pr* **have** (λ *x u*. (*u div* (*2* ^ *x*)) *mod 2*) ∈ *PrimRec2* **by** *prec*
  with *c-in-def* **show** *?thesis* **by** *auto*
qed

definition
  nat-to-set :: *nat* ⇒ *nat set* **where**
  nat-to-set u ≡ {*x*. *2*^*x* ≤ *u* ∧ *c-in x u* = *1*}

lemma *c-in-upper-bound*: *c-in x u* = *1* ⟹ *2* ^ *x* ≤ *u*
proof −

```
  assume A: c-in x u = 1
  then have S1: (u div (2^x)) mod 2 = 1 by (unfold c-in-def)
  then have S2: u div (2^x) > 0 by arith
  show ?thesis
  proof (rule ccontr)
    assume ¬ 2 ^ x ≤ u
    then have u < 2^x by auto
    then have u div (2^x) = 0 by (rule div-less)
    with S2 show False by auto
  qed
qed

lemma nat-to-set-upper-bound: x ∈ nat-to-set u ⟹ 2 ^ x ≤ u by (simp add:
nat-to-set-def)

lemma x-lt-2-x: x < 2 ^ x
  by (rule less-exp)

lemma nat-to-set-upper-bound1: x ∈ nat-to-set u ⟹ x < u
proof −
  assume x ∈ nat-to-set u
  then have S1: 2 ^ x ≤ u by (simp add: nat-to-set-def)
  have S2: x < 2 ^ x by (rule x-lt-2-x)
  from S2 S1 show ?thesis
    by (rule less-le-trans)
qed

lemma nat-to-set-upper-bound2: nat-to-set u ⊆ {i. i < u}
proof −
  from nat-to-set-upper-bound1 show ?thesis by blast
qed

lemma nat-to-set-is-finite: finite (nat-to-set u)
proof −
  have S1: finite {i. i<u}
  proof −
    let ?B = {i. i<u}
    let ?f = (λ (x::nat). x)
    have ?B = ?f ‘ ?B by auto
    then show finite ?B by (rule nat-seg-image-imp-finite)
  qed
  have S2: nat-to-set u ⊆ {i. i<u} by (rule nat-to-set-upper-bound2)
  from S2 S1 show ?thesis by (rule finite-subset)
qed

lemma x-in-u-eq: (x ∈ nat-to-set u) = (c-in x u = 1) by (auto simp add: nat-to-set-def
c-in-upper-bound)

definition
```

```
  log2 :: nat ⇒ nat where
  log2 = (λ x. Least(%z. x < 2^(z+1)))

lemma log2-at-0: log2 0 = 0
proof −
  let ?v = log2 0
  have S1: 0 ≤ ?v by auto
  have S2: ?v = Least(%(z::nat). (0::nat)<2^(z+1)) by (simp add: log2-def)
  have S3: (0::nat)<2^(0+1) by auto
  from S3 have S4: Least(%(z::nat). (0::nat)<2^(z+1)) ≤ 0 by (rule Least-le)
  from S2 S4 have S5: ?v ≤ 0 by auto
  from S1 S5 have S6: ?v = 0 by auto
  thus ?thesis by auto
qed

lemma log2-at-1: log2 1 = 0
proof −
  let ?v = log2 1
  have S1: 0 ≤ ?v by auto
  have S2: ?v = Least(%(z::nat). (1::nat)<2^(z+1)) by (simp add: log2-def)
  have S3: (1::nat)<2^(0+1) by auto
  from S3 have S4: Least(%(z::nat). (1::nat)<2^(z+1)) ≤ 0 by (rule Least-le)
  from S2 S4 have S5: ?v ≤ 0 by auto
  from S1 S5 have S6: ?v = 0 by auto
  thus ?thesis by auto
qed

lemma log2-le: x > 0 ⟹ 2 ^ log2 x ≤ x
proof −
  assume A: x > 0
  show ?thesis
  proof (cases)
    assume A1: log2 x = 0
    with A show ?thesis by auto
  next
    assume A1: log2 x ≠ 0
    then have S1: log2 x > 0 by auto
    define y where y = log2 x − 1
    from S1 y-def have S2: log2 x = y + 1 by auto
    then have S3: y < log2 x by auto
    have 2^(y+1) ≤ x
    proof (rule ccontr)
      assume A2: ¬ 2^(y+1) ≤ x then have x < 2^(y+1) by auto
      then have log2 x ≤ y by (simp add: log2-def Least-le)
      with S3 show False by auto
   qed
   with S2 show ?thesis by auto
  qed
qed
```

```
lemma log2-gt: x < 2 ^ (log2 x + 1)
proof -
  have x < 2^x by (rule x-lt-2-x)
  then have S1: x < 2^(x+1)
    by (simp add: numeral-2-eq-2)
  define y where y = x
  from S1 y-def have S2: x < 2^(y+1) by auto
  let ?P = λ z. x < 2^(z+1)
  from S2 have S3: ?P y by auto
  then have S4: ?P (Least ?P) by (rule LeastI)
  from log2-def have S5: log2 x = Least ?P by (unfold log2-def, auto)
  from S4 S5 show ?thesis by auto
qed

lemma x-div-x: x > 0 ⟹ (x::nat) div x = 1 by auto
lemma div-ge: (k::nat) ≤ m div n ⟹ n∗k ≤ m
proof -
  assume A: k ≤ m div n
  have S1: n ∗ (m div n) + m mod n = m by (rule mult-div-mod-eq)
  have S2: 0 ≤ m mod n by auto
  from S1 S2 have S3: n ∗ (m div n) ≤ m by arith
  from A have S4: n ∗ k ≤  n ∗ (m div n) by auto
  from S4 S3 show ?thesis by (rule order-trans)
qed
lemma div-lt: m < n∗k ⟹ m div n < (k::nat)
proof -
  assume A: m < n∗k
  show ?thesis
  proof (rule ccontr)
    assume ¬ m div n < k
    then have S1: k ≤ m div n by auto
    then have S2: n∗k ≤ m by (rule div-ge)
    with A show False by auto
  qed
qed

lemma log2-lm1: u > 0 ⟹ u div 2 ^ (log2 u) = 1
proof -
  assume A: u > 0
  then have S1: 2^(log2 u) ≤ u by (rule log2-le)
  have S2: u < 2^(log2 u+1) by (rule log2-gt)
  then have S3: u < (2^log2 u)∗2 by simp
  have (2::nat) > 0 by simp
  then have (2::nat)^log2 u > 0 by simp
  then have S4: (2::nat)^log2 u div 2^log2 u = 1 by auto
   from S1 have S5: (2::nat)^log2 u div 2^log2 u  ≤ u div 2^log2 u by (rule div-le-mono)
  with S4 have S6: 1 ≤ u div 2^log2 u by auto
```

```
  from S3 have S7: u div 2^log2 u < 2 by (rule div-lt)
  from S6 S7 show ?thesis by auto
qed

lemma log2-lm2: u > 0 ⟹ c-in (log2 u) u = 1
proof −
  assume A: u > 0
  then have S1: u div 2 ^ (log2 u) = 1 by (rule log2-lm1)
  have c-in (log2 u) u = (u div 2 ^ (log2 u)) mod 2 by (simp add: c-in-def)
  also from S1 have ... = 1 mod 2 by simp
  also have ... = 1 by auto
  finally show ?thesis by auto
qed

lemma log2-lm3: log2 u < x ⟹ c-in x u = 0
proof −
  assume A: log2 u < x
  then have S1: (log2 u)+1 ≤ x by auto
  have S2: 1 ≤ (2::nat) by auto
  from S1 S2 have S3: (2::nat)^((log2 u)+1) ≤ 2^x by (rule power-increasing)
  have S4: u < (2::nat)^((log2 u)+1) by (rule log2-gt)
  from S3 S4 have S5: u < 2^x by auto
  then have S6: u div 2^x = 0 by (rule div-less)
  have c-in x u = (u div 2^x) mod 2 by (simp add: c-in-def)
  also from S6 have ... = 0 mod 2 by simp
  also have ... = 0 by auto
  finally have ?thesis by auto
  thus ?thesis by auto
qed

lemma log2-lm4: c-in x u = 1 ⟹ x ≤ log2 u
proof −
  assume A: c-in x u = 1
  show ?thesis
  proof (rule ccontr)
    assume ¬ x ≤ log2 u
    then have S1: log2 u < x by auto
    then have S2: c-in x u = 0 by (rule log2-lm3)
    with A show False by auto
  qed
qed

lemma nat-to-set-lub: x ∈ nat-to-set u ⟹ x ≤ log2 u
proof −
  assume x ∈ nat-to-set u
  then have S1: c-in x u = 1 by (simp add: x-in-u-eq)
  then show ?thesis by (rule log2-lm4)
qed
```

```
lemma log2-lm5: u > 0 ⟹ log2 u ∈ nat-to-set u
proof −
  assume A: u > 0
  then have c-in (log2 u) u = 1 by (rule log2-lm2)
  then show ?thesis by (simp add: x-in-u-eq)
qed

lemma pos-imp-ne: u > 0 ⟹ nat-to-set u ≠ {}
proof −
  assume u > 0
  then have log2 u ∈ nat-to-set u by (rule log2-lm5)
  thus ?thesis by auto
qed

lemma empty-is-zero: nat-to-set u = {} ⟹ u = 0
proof (rule ccontr)
  assume A1: nat-to-set u = {}
  assume A2: u ≠ 0 then have S1: u > 0 by auto
  from S1 have nat-to-set u ≠ {} by (rule pos-imp-ne)
  with A1 show False by auto
qed

lemma log2-is-max: u > 0 ⟹ log2 u = Max (nat-to-set u)
proof −
  assume A: u > 0
  then have S1: log2 u ∈ nat-to-set u by (rule log2-lm5)
  define max where max = Max (nat-to-set u)
  from A have ne: nat-to-set u ≠ {} by (rule pos-imp-ne)
  have finite: finite (nat-to-set u) by (rule nat-to-set-is-finite)
  from max-def finite ne have max-in: max ∈ nat-to-set u by simp
  from max-in have S2: c-in max u = 1 by (simp add: x-in-u-eq)
  then have S3: max ≤ log2 u by (rule log2-lm4)
  from finite ne S1 max-def have S4: log2 u ≤ max by simp
  from S3 S4 max-def show ?thesis by auto
qed

lemma zero-is-empty: nat-to-set 0 = {}
proof −
  have S1: {i. i<(0::nat)} = {} by blast
  have S2: nat-to-set 0 ⊆ {i. i<0} by (rule nat-to-set-upper-bound2)
  from S1 S2 show ?thesis by auto
qed

lemma ne-imp-pos: nat-to-set u ≠ {} ⟹ u > 0
proof (rule ccontr)
  assume A1: nat-to-set u ≠ {}
  assume ¬ 0 < u then have u = 0 by auto
  then have nat-to-set u = {} by (simp add: zero-is-empty)
  with A1 show False by auto
```

```
qed

lemma div-mod-lm: y < x ⟹ ((u + (2::nat) ^ x) div (2::nat)^y) mod 2 = (u div (2::nat)^y) mod 2
proof −
  assume y-lt-x: y < x
  let ?n = (2::nat)^y
  have n-pos: 0 < ?n by auto
  let ?s = x−y
  from y-lt-x have s-pos: 0 < ?s by auto
  from y-lt-x have S3: x = y + ?s by auto
  from S3 have (2::nat)^x = (2::nat)^(y + ?s) by auto
  moreover have (2::nat)^(y +?s) = (2::nat)^y ∗ 2^ ?s by (rule power-add)
  ultimately have (2::nat)^x = 2^y ∗ 2^?s by auto
  then have S4: u + (2::nat)^x = u + (2::nat)^y ∗ 2^?s by auto
  from n-pos have S5: (u + (2::nat)^y ∗ 2^?s) div 2^y = 2^?s + (u div 2^y) by simp
  from S4 S5 have S6: (u + (2::nat)^x) div 2^y = 2^?s + (u div 2^y) by auto
  from s-pos have S8: ?s = (?s − 1) + 1 by auto
  have (2::nat) ^((?s − (1::nat)) + (1::nat)) = (2::nat) ^(?s − (1::nat)) ∗ 2^1 by (rule power-add)
  with S8 have S9: (2::nat) ^ ?s = (2::nat) ^(?s − (1::nat)) ∗ 2 by auto
  then have S10: 2^?s + (u div 2^y) = (u div 2^y) + (2::nat) ^(?s − (1::nat)) ∗ 2 by auto
  have S11: ((u div 2^y) + (2::nat) ^(?s − (1::nat)) ∗ 2) mod 2 = (u div 2^y) mod 2 by (rule mod-mult-self1)
  from S6 S10 S11 show ?thesis by auto
qed

lemma add-power: u < 2^x ⟹ nat-to-set (u + 2^x) = nat-to-set u ∪ {x}
proof −
  assume A: u < 2^x
  have log2-is-x: log2 (u+2^x) = x
  proof (unfold log2-def, rule Least-equality)
    from A show u+2^x < 2^(x+1) by auto
  next
    fix z
    assume A1: u + 2^x < 2^(z+1)
    show x ≤ z
    proof (rule ccontr)
      assume ¬ x ≤ z
      then have z < x by auto
      then have L1: z+1 ≤ x by auto
      have L2: 1 ≤ (2::nat) by auto
     from L1 L2 have L3: (2::nat)^(z+1) ≤ (2::nat)^x by (rule power-increasing)
      with A1 show False by auto
    qed
  qed
  show ?thesis
```

```
  proof (rule subset-antisym)
    show nat-to-set (u + 2 ^ x) ⊆ nat-to-set u ∪ {x}
    proof fix y
      assume A1: y ∈ nat-to-set (u + 2 ^ x)
      show y ∈ nat-to-set u ∪ {x}
      proof
        assume y ∉ {x} then have S1: y ≠ x by auto
        from A1 have y ≤ log2 (u + 2 ^ x) by (rule nat-to-set-lub)
        with log2-is-x have y ≤ x by auto
        with S1 have y-lt-x: y < x by auto
        from A1 have c-in y (u + 2 ^ x) = 1 by (simp add: x-in-u-eq)
        then have S2: ((u + 2 ^ x) div 2^y) mod 2 = 1 by (unfold c-in-def)
          from y-lt-x have ((u + (2::nat) ^ x) div (2::nat)^y) mod 2 = (u div
(2::nat)^y) mod 2 by (rule div-mod-lm)
        with S2 have (u div 2^y) mod 2 = 1 by auto
        then have c-in y u = 1 by (simp add: c-in-def)
        then show y ∈ nat-to-set u by (simp add: x-in-u-eq)
      qed
    qed
  next
  show nat-to-set u ∪ {x} ⊆ nat-to-set (u + 2 ^ x)
  proof fix y
    assume A1: y ∈ nat-to-set u ∪ {x}
    show y ∈ nat-to-set (u + 2 ^ x)
    proof cases
      assume y ∈ {x}
      then have y=x by auto
      then have y = log2 (u + 2 ^ x) by (simp add: log2-is-x)
      then show ?thesis by (simp add: log2-lm5)
    next
      assume y-notin: y ∉ {x}
      then have y-ne-x: y ≠ x by auto
      from A1 y-notin have y-in: y ∈ nat-to-set u by auto
      have y-lt-x: y < x
      proof (rule ccontr)
        assume ¬ y < x
        with y-ne-x have y-gt-x: x < y by auto
        have 1 < (2::nat) by auto
       from y-gt-x this have L1: (2::nat)^x < 2^y by (rule power-strict-increasing)
        from y-in have L2: 2^y ≤ u by (rule nat-to-set-upper-bound)
        from L1 L2 have (2::nat)^x < u by arith
        with A show False by auto
      qed
      from y-in have c-in y u = 1 by (simp add: x-in-u-eq)
      then have S2: (u div 2^y) mod 2 = 1 by (unfold c-in-def)
       from y-lt-x have ((u + (2::nat) ^ x) div (2::nat)^y) mod 2 = (u div
(2::nat)^y) mod 2 by (rule div-mod-lm)
      with S2 have ((u + (2::nat) ^ x)div 2^y) mod 2 = 1 by auto
      then have c-in y (u + (2::nat) ^ x) = 1 by (simp add: c-in-def)
```

```
      then show y ∈ nat-to-set (u + (2::nat) ^ x) by (simp add: x-in-u-eq)
    qed
  qed
  qed
qed

theorem nat-to-set-inj: nat-to-set u = nat-to-set v ⟹ u = v
proof −
  assume A: nat-to-set u = nat-to-set v
  let ?P = λ (n::nat). (∀ (D::nat set). finite D ∧ card D ≤ n ⟶ (∀ u v. nat-to-set
u = D ∧ nat-to-set v = D ⟶ u = v))
  have P-at-0: ?P 0
  proof fix D show finite D ∧ card D ≤ 0 ⟶ (∀ u v. nat-to-set u = D ∧ nat-to-set
v = D ⟶ u = v)
    proof (rule impI)
      assume A1: finite D ∧ card D ≤ 0
      from A1 have S1: finite D by auto
      from A1 have S2: card D = 0 by auto
      from S1 S2 have S3: D = {} by auto
      show (∀ u v. nat-to-set u = D ∧ nat-to-set v = D ⟶ u = v)
      proof (rule allI, rule allI) fix u v show nat-to-set u = D ∧ nat-to-set v =
D ⟶ u = v
        proof
          assume A2: nat-to-set u = D ∧ nat-to-set v = D
          from A2 have L1: nat-to-set u = D by auto
          from A2 have L2: nat-to-set v = D by auto
          from L1 S3 have nat-to-set u = {} by auto
          then have u-z: u = 0 by (rule empty-is-zero)
          from L2 S3 have nat-to-set v = {} by auto
          then have v-z: v = 0 by (rule empty-is-zero)
          from u-z v-z show u=v by auto
        qed
      qed
    qed
  qed
  have P-at-Suc: ⋀ n. ?P n ⟹ ?P (Suc n)
  proof − fix n
    assume A-n: ?P n
    show ?P (Suc n)
    proof fix D show finite D ∧ card D ≤ Suc n ⟶ (∀ u v. nat-to-set u = D ∧
nat-to-set v = D ⟶ u = v)
    proof (rule impI)
      assume A1: finite D ∧ card D ≤ Suc n
      from A1 have S1: finite D by auto
      from A1 have S2: card D ≤ Suc n by auto
      show (∀ u v. nat-to-set u = D ∧ nat-to-set v = D ⟶ u = v)
      proof (rule allI, rule allI, rule impI)
        fix u v
        assume A2: nat-to-set u = D ∧ nat-to-set v = D
```

```
from A2 have d-u-d: nat-to-set u = D by auto
from A2 have d-v-d: nat-to-set v = D by auto
show u = v
proof (cases)
 assume A3: D = {}
 from A3 d-u-d have nat-to-set u = {} by auto
 then have u-z: u = 0 by (rule empty-is-zero)
 from A3 d-v-d have nat-to-set v = {} by auto
 then have v-z: v = 0 by (rule empty-is-zero)
 from u-z v-z show u = v by auto
next
 assume A3: D ≠ {}
 from A3 d-u-d have nat-to-set u ≠ {} by auto
 then have u-pos: u > 0 by (rule ne-imp-pos)
 from A3 d-v-d have nat-to-set v ≠ {} by auto
 then have v-pos: v > 0 by (rule ne-imp-pos)
 define m where m = Max D
 from S1 m-def A3 have m-in: m ∈ D by auto
 from d-u-d m-def have m-u: m = Max (nat-to-set u) by auto
 from d-v-d m-def have m-v: m = Max (nat-to-set v) by auto
 from u-pos m-u log2-is-max have m-log-u: m = log2 u by auto
 from v-pos m-v log2-is-max have m-log-v: m = log2 v by auto
 define D1 where D1 = D − {m}
 define u1 where u1 = u − 2^m
 define v1 where v1 = v − 2^m
 have card-D1: card D1 ≤ n
 proof −
   from D1-def S1 m-in have card D1 = (card D) − 1 by (simp add: card-Diff-singleton)
  with S2 show ?thesis by auto
 qed
 have u-u1: u = u1 + 2^m
 proof −
  from u-pos have L1: 2 ^ log2 u ≤ u by (rule log2-le)
  with m-log-u have L2: 2 ^ m ≤ u by auto
  with u1-def show ?thesis by auto
 qed
 have u1-d1: nat-to-set u1 = D1
 proof −
  from m-log-u log2-gt have u < 2^(m+1) by auto
  with u-u1 have u1-lt-2-m: u1 < 2^m by auto
  with u-u1 have L1: nat-to-set u = nat-to-set u1 ∪ {m} by (simp add: add-power)
  have m-notin: m ∉ nat-to-set u1
  proof (rule ccontr)
   assume ¬ m ∉ nat-to-set u1 then have m ∈ nat-to-set u1 by auto
   then have 2^m ≤ u1 by (rule nat-to-set-upper-bound)
   with u1-lt-2-m show False by auto
  qed
```

```
            from L1 m-notin have nat-to-set u1 = nat-to-set u − {m} by auto
            with d-u-d have nat-to-set u1 = D − {m} by auto
            with D1-def show ?thesis by auto
          qed
          have v-v1: v = v1 + 2^m
          proof −
            from v-pos have L1: 2 ^ log2 v ≤ v by (rule log2-le)
            with m-log-v have L2: 2 ^ m ≤ v by auto
            with v1-def show ?thesis by auto
          qed
          have v1-d1: nat-to-set v1 = D1
          proof −
            from m-log-v log2-gt have v < 2^(m+1) by auto
            with v-v1 have v1-lt-2-m: v1 < 2^m by auto
            with v-v1 have L1: nat-to-set v = nat-to-set v1 ∪ {m} by (simp add: add-power)
            have m-notin: m ∉ nat-to-set v1
            proof (rule ccontr)
              assume ¬ m ∉ nat-to-set v1 then have m ∈ nat-to-set v1 by auto
              then have 2^m ≤ v1 by (rule nat-to-set-upper-bound)
              with v1-lt-2-m show False by auto
            qed
            from L1 m-notin have nat-to-set v1 = nat-to-set v − {m} by auto
            with d-v-d have nat-to-set v1 = D − {m} by auto
            with D1-def show ?thesis by auto
          qed
          from S1 D1-def have P1: finite D1 by auto
          with card-D1 have P2: finite D1 ∧ card D1 ≤ n by auto
          from A-n P2 have (∀ u v. nat-to-set u = D1 ∧ nat-to-set v = D1 ⟶ u = v) by auto
          with u1-d1 v1-d1 have u1 = v1 by auto
          with u-u1 v-v1 show u = v by auto
        qed
      qed
    qed
  qed
qed
  from P-at-0 P-at-Suc have main: ⋀ n. ?P n by (rule nat.induct)
  define D where D = nat-to-set u
  from D-def A have P1: nat-to-set u = D by auto
  from D-def A have P2: nat-to-set v = D by auto
  from D-def nat-to-set-is-finite have d-finite: finite D by auto
  define n where n = card D
  from n-def d-finite have card-le: card D ≤ n by auto
  from d-finite card-le have P3: finite D ∧ card D ≤ n by auto
  with main have P4: ∀ u v. nat-to-set u = D ∧ nat-to-set v = D ⟶ u = v by auto
  with P1 P2 show u = v by auto
qed
```

```
definition
  set-to-nat :: nat set => nat where
  set-to-nat = (λ D. sum (λ x. 2 ^ x) D)

lemma two-power-sum: sum (λ x. (2::nat) ^ x) {i. i< Suc m} = (2 ^ Suc m) −
1
proof (induct m)
  show sum (λ x. (2::nat) ^ x) {i. i< Suc 0} = (2 ^ Suc 0) − 1 by auto
next
  fix n
  assume A: sum (λ x. (2::nat) ^ x) {i. i< Suc n} = (2 ^ Suc n) − 1
  show sum (λ x. (2::nat) ^ x) {i. i< Suc (Suc n)} = (2 ^ Suc (Suc n)) − 1
  proof −
    let ?f = λ x. (2::nat) ^ x
    have S1: {i. i< Suc (Suc n)} = {i. i ≤ Suc n} by auto
    have S2: {i. i ≤ Suc n} = {i. i < Suc n} ∪ { Suc n} by auto
     from S1 S2 have S3: {i. i< Suc (Suc n)} = {i. i < Suc n} ∪ { Suc n} by
auto
    have S4: {i. i < Suc n} = (λ x. x) ‘ {i. i < Suc n} by auto
    then have S5: finite {i. i < Suc n} by (rule nat-seg-image-imp-finite)
    have S6: Suc n ∉ {i. i < Suc n} by auto
    from S5 S6 sum.insert have S7: sum ?f ({i. i< Suc n} ∪ {Suc n}) = 2 ^ Suc
n + sum ?f {i. i< Suc n} by auto
     from S3 have sum ?f {i. i< Suc (Suc n)} = sum ?f ({i. i< Suc n} ∪ {Suc
n}) by auto
    also from S7 have ... = 2 ^ Suc n + sum ?f {i. i< Suc n} by auto
    also from A have ... = 2 ^ Suc n + (((2::nat) ^ Suc n)−(1::nat)) by auto
    also have ... = (2 ^ Suc (Suc n)) − 1 by auto
    finally show ?thesis by auto
  qed
qed

lemma finite-interval: finite {i. (i::nat)<m}
proof −
  have {i. i < m} = (λ x. x) ‘ {i. i < m} by auto
  then show ?thesis by (rule nat-seg-image-imp-finite)
qed

lemma set-to-nat-at-empty: set-to-nat {} = 0 by (unfold set-to-nat-def, rule sum.empty)

lemma set-to-nat-of-interval: set-to-nat {i. (i::nat)<m} = 2 ^ m − 1
proof (induct m)
  show set-to-nat {i. i < 0} = 2 ^ 0 − 1
  proof −
    have S1: {i. (i::nat) < 0} = {} by auto
    with set-to-nat-at-empty have set-to-nat {i. i<0} = 0 by auto
    thus ?thesis by auto
  qed
```

```
next
  fix n show set-to-nat {i. i < Suc n} = 2 ^ Suc n − 1 by (unfold set-to-nat-def,
rule two-power-sum)
qed

lemma set-to-nat-mono: ⟦ finite B; A ⊆ B⟧ ⟹ set-to-nat A ≤ set-to-nat B
proof −
  assume b-finite: finite B
  assume a-le-b: A ⊆ B
  let ?f = λ (x::nat). (2::nat) ^ x
  have S1: set-to-nat A = sum ?f A by (simp add: set-to-nat-def)
  have S2: set-to-nat B = sum ?f B by (simp add: set-to-nat-def)
  have S3: ⋀ x. x ∈ B − A ⟹ 0 ≤ ?f x by auto
  from b-finite a-le-b S3 have sum ?f A ≤ sum ?f B by (rule sum-mono2)
  with S1 S2 show ?thesis by auto
qed

theorem nat-to-set-srj: finite (D::nat set) ⟹ nat-to-set (set-to-nat D) = D
proof −
  assume A: finite D
   let ?P = λ (n::nat). (∀ (D::nat set). finite D ∧ card D = n ⟶ nat-to-set
(set-to-nat D) = D)
  have P-at-0: ?P 0
  proof (rule allI)
    fix D
    show finite D ∧ card D = 0 ⟶ nat-to-set (set-to-nat D) = D
    proof
      assume A1: finite D ∧ card D = 0
      from A1 have S1: finite D by auto
      from A1 have S2: card D = 0 by auto
      from S1 S2 have S3: D = {} by auto
      with set-to-nat-def have set-to-nat D = sum (λ x. 2 ^ x) D by simp
      with S3 sum.empty have set-to-nat D = 0 by auto
      with zero-is-empty S3 show nat-to-set (set-to-nat D) = D by auto
    qed
  qed
  have P-at-Suc: ⋀ n. ?P n ⟹ ?P (Suc n)
  proof − fix n
    assume A-n: ?P n
    show ?P (Suc n)
    proof
      fix D show finite D ∧ card D = Suc n ⟶ nat-to-set (set-to-nat D) = D
      proof
        assume A1: finite D ∧ card D = Suc n
        from A1 have S1: finite D by auto
        from A1 have S2: card D = Suc n by auto
        define m where m = Max D
        from S2 have D-ne: D ≠ {} by auto
        with S1 m-def have m-in: m ∈ D by auto
```

```
        define D1 where D1 = D − {m}
        from S1 D1-def have d1-finite: finite D1 by auto
            from D1-def m-in S1 have card D1 = card D − 1 by (simp add:
card-Diff-singleton)
        with S2 have card-d1: card D1 = n by auto
        from d1-finite card-d1 have finite D1 ∧ card D1 = n by auto
        with A-n have S3: nat-to-set (set-to-nat D1) = D1 by auto
        define u where u = set-to-nat D
        define u1 where u1 = set-to-nat D1
        from S1 m-in have sum (λ (x::nat). (2::nat) ^ x) D = 2 ^ m + sum (λ x.
2 ^ x) (D − {m})
          by (rule sum.remove)
        with set-to-nat-def have set-to-nat D = 2 ^ m + set-to-nat (D − {m}) by
auto
        with u-def u1-def D1-def have u-u1: u = u1 + 2 ^ m by auto
        from S3 u1-def have d1-u1: nat-to-set u1 = D1 by auto
        have u1-lt: u1 < 2 ^ m
        proof −
          have L1: D1 ⊆ {i. i<m}
          proof fix x
            assume A1: x ∈ D1
            show x ∈ {i. i<m}
            proof
              from A1 D1-def have L1-1: x ∈ D by auto
              from S1 D-ne L1-1 m-def have L1-2: x ≤ m by auto
              with A1 L1-1 D1-def have x ≠ m by auto
              with L1-2 show x < m by auto
            qed
          qed
          have L2: finite {i. i<m} by (rule finite-interval)
               from L2 L1 have set-to-nat D1 ≤ set-to-nat {i. i<m} by (rule
set-to-nat-mono)
          with u1-def have u1 ≤ set-to-nat {i. i<m} by auto
          with set-to-nat-of-interval have L3: u1 ≤ 2 ^ m − 1 by auto
          have 0 < (2::nat) ^ m by auto
          then have (2::nat) ^ m − 1 < (2::nat) ^ m by auto
          with L3 show ?thesis by arith
        qed
        from u-def have nat-to-set (set-to-nat D) = nat-to-set u by auto
        also from u-u1 have ... = nat-to-set (u1 + 2 ^ m) by auto
        also from u1-lt have ... = nat-to-set u1 ∪ {m} by (rule add-power)
        also from d1-u1 have ... = D1 ∪ {m} by auto
        also from D1-def m-in have ... = D by auto
        finally show nat-to-set (set-to-nat D) = D by auto
      qed
    qed
  qed
  from P-at-0 P-at-Suc have main: ⋀ n. ?P n by (rule nat.induct)
  from A main show ?thesis by auto
```

```
qed

theorem nat-to-set-srj1: finite (D::nat set) ⟹ ∃ u. nat-to-set u = D
proof –
  assume A: finite D
  show ∃ u. nat-to-set u = D
  proof
    from A show nat-to-set (set-to-nat D) = D by (rule nat-to-set-srj)
  qed
qed

lemma sum-of-pr-is-pr: g ∈ PrimRec1 ⟹ (λ n. sum g {i. i<n}) ∈ PrimRec1
proof –
  assume g-is-pr: g ∈ PrimRec1
  define f where f n = sum g {i. i<n} for n
  from f-def have f-at-0: f 0 = 0 by auto
  define h where h a b = g a + b for a b
  from g-is-pr have h-is-pr: h ∈ PrimRec2 unfolding h-def by prec
  have f-at-Suc: ∀ y. f (Suc y) = h y (f y)
  proof
    fix y show f (Suc y) = h y (f y)
    proof –
      from f-def have S1: f (Suc y) = sum g {i. i < Suc y} by auto
      have S2: {i. i < Suc y} = {i. i < y} ∪ {y} by auto
      have S3: finite {i. i < y} by (rule finite-interval)
      have S4: y ∉ {i. i < y} by auto
      from S1 S2 have f (Suc y) = sum g ({i. (i::nat) < y} ∪ {y}) by auto
      also from S3 S4 sum.insert have ... = g y + sum g {i. i<y} by auto
      also from f-def have ... = g y + f y by auto
      also from h-def have ... = h y (f y) by auto
      finally show ?thesis by auto
    qed
  qed
  from h-is-pr f-at-0 f-at-Suc have f-is-pr: f ∈ PrimRec1 by (rule pr-rec1-scheme)
  with f-def [abs-def] show ?thesis by auto
qed

lemma sum-of-pr-is-pr2: p ∈ PrimRec2 ⟹ (λ n m. sum (λ x. p x m) {i. i<n})
∈ PrimRec2
proof –
  assume p-is-pr: p ∈ PrimRec2
  define f where f n m = sum (λ x. p x m) {i. i<n} for n m
  define g :: nat ⇒ nat where g x = 0 for x
  have g-is-pr: g ∈ PrimRec1 by (unfold g-def, rule const-is-pr [where ?n=0])
  have f-at-0: ∀ x. f 0 x = g x
  proof
    fix x from f-def g-def show f 0 x = g x by auto
  qed
  define h where h a b c = p a c + b for a b c
```

```
  from p-is-pr have h-is-pr: h ∈ PrimRec3 unfolding h-def by prec
  have f-at-Suc: ∀ x y. f (Suc y) x = h y (f y x) x
  proof (rule allI, rule allI)
    fix x y show f (Suc y) x = h y (f y x) x
    proof −
      from f-def have S1: f (Suc y) x = sum (λ z. p z x) {i. i < Suc y} by auto
      have S2: {i. i < Suc y} = {i. i < y} ∪ {y} by auto
      have S3: finite {i. i < y} by (rule finite-interval)
      have S4: y ∉ {i. i < y} by auto
      define g1 where g1 z = p z x for z
      from S1 S2 g1-def have f (Suc y) x = sum g1 ({i. (i::nat) < y} ∪ {y}) by
auto
      also from S3 S4 sum.insert have … = g1 y + sum g1 {i. i<y} by auto
      also from f-def g1-def have … = g1 y + f y x by auto
      also from h-def g1-def have … = h y (f y x) x by auto
      finally show ?thesis by auto
    qed
  qed
  from g-is-pr h-is-pr f-at-0 f-at-Suc have f-is-pr: f ∈ PrimRec2 by (rule pr-rec-scheme)
  with f-def [abs-def] show ?thesis by auto
qed

lemma sum-is-pr: g ∈ PrimRec1 ⟹ (λ u. sum g (nat-to-set u)) ∈ PrimRec1
proof −
  assume g-is-pr: g ∈ PrimRec1
  define g1 where g1 x u = (if (c-in x u = 1) then (g x) else 0) for x u
  have g1-is-pr: g1 ∈ PrimRec2
  proof (unfold g1-def, rule if-eq-is-pr2)
    show c-in ∈ PrimRec2 by (rule c-in-is-pr)
  next
    show (λx y. 1) ∈ PrimRec2 by (rule const-is-pr-2 [where ?n=1])
  next
    from g-is-pr show (λx y. g x) ∈ PrimRec2 by prec
  next
    show (λx y. 0) ∈ PrimRec2 by (rule const-is-pr-2 [where ?n=0])
  qed
  define f where f u = sum (λ x. g1 x u) {i. (i::nat) < u} for u
  define f1 where f1 u v = sum (λ x. g1 x v) {i. (i::nat) < u} for u v
  from g1-is-pr have (λ (u::nat) v. sum (λ x. g1 x v) {i. (i::nat) < u}) ∈ PrimRec2
    by (rule sum-of-pr-is-pr2)
  with f1-def [abs-def] have f1-is-pr: f1 ∈ PrimRec2 by auto
  from f-def f1-def have f-f1: f = (λ u. f1 u u) by auto
  from f1-is-pr have (λ u. f1 u u) ∈ PrimRec1 by prec
  with f-f1 have f-is-pr: f ∈ PrimRec1 by auto
  have f-is-result: f = (λ u. sum g (nat-to-set u))
  proof
    fix u show f u = sum g (nat-to-set u)
    proof −
      define U where U = {i. i < u}
```

```
      define A where A = {x ∈ U. c-in x u = 1}
      define B where B = {x ∈ U. c-in x u ≠ 1}
      have U-finite: finite U by (unfold U-def, rule finite-interval)
      from A-def U-finite have A-finite: finite A by auto
      from B-def U-finite have B-finite: finite B by auto
      from U-def A-def B-def have U-A-B: U = A ∪ B by auto
      from U-def A-def B-def have A-B: A ∩ B = {} by auto
      from B-def g1-def have B-z: sum (λ x. g1 x u) B = 0 by auto
     have u-in-U: nat-to-set u ⊆ U by (unfold U-def, rule nat-to-set-upper-bound2)
      from u-in-U x-in-u-eq A-def have A-u: A = nat-to-set u by auto
     from A-u x-in-u-eq g1-def have A-res: sum (λ x. g1 x u) A = sum g (nat-to-set
u) by auto
      from f-def have f u = sum (λ x. g1 x u) {i. (i::nat) < u} by auto
      also from U-def have ... = sum (λ x. g1 x u) U by auto
      also from U-A-B have ... = sum (λ x. g1 x u) (A ∪ B) by auto
      also from A-finite B-finite A-B have ... = sum (λ x. g1 x u) A + sum (λ
x. g1 x u) B by (rule sum.union-disjoint)
      also from B-z have ... = sum (λ x. g1 x u) A by auto
      also from A-res have ... = sum g (nat-to-set u) by auto
      finally show ?thesis by auto
    qed
  qed
  with f-is-pr show ?thesis by auto
qed

definition
  c-card :: nat ⇒ nat where
  c-card = (λ u. card (nat-to-set u))

theorem c-card-is-pr: c-card ∈ PrimRec1
proof −
  define g :: nat ⇒ nat where g x = 1 for x
  have g-is-pr: g ∈ PrimRec1 by (unfold g-def, rule const-is-pr)
  have c-card = (λ u. sum g (nat-to-set u))
  proof
    fix u show c-card u = sum g (nat-to-set u) by (unfold c-card-def, unfold g-def,
rule card-eq-sum)
  qed
  moreover from g-is-pr have (λ u. sum g (nat-to-set u)) ∈ PrimRec1 by (rule
sum-is-pr)
  ultimately show ?thesis by auto
qed

definition
  c-insert :: nat ⇒ nat ⇒ nat where
  c-insert = (λ x u. if c-in x u = 1 then u else u + 2^x)

lemma c-insert-is-pr: c-insert ∈ PrimRec2
proof (unfold c-insert-def, rule if-eq-is-pr2)
```

```
  show c-in ∈ PrimRec2 by (rule c-in-is-pr)
next
  show (λx y. 1) ∈ PrimRec2 by (rule const-is-pr-2)
next
  show (λx y. y) ∈ PrimRec2 by (rule pr-id2-2)
next
  from power-is-pr show (λx y. y + 2 ^ x) ∈ PrimRec2 by prec
qed

lemma [simp]: set-to-nat (nat-to-set u) = u
proof −
  define D where D = nat-to-set u
  from D-def nat-to-set-is-finite have D-finite: finite D by auto
  then have nat-to-set (set-to-nat D) = D by (rule nat-to-set-srj)
  with D-def have nat-to-set (set-to-nat D) = nat-to-set u by auto
  then have set-to-nat D = u by (rule nat-to-set-inj)
  with D-def show ?thesis by auto
qed

lemma insert-lemma: x ∉ nat-to-set u ⟹ set-to-nat (nat-to-set u ∪ {x}) = u +
2 ^ x
proof −
  assume A: x ∉ nat-to-set u
  define D where D = nat-to-set u
  from A D-def have S1: x ∉ D by auto
  have finite (nat-to-set u) by (rule nat-to-set-is-finite)
  with D-def have D-finite: finite D by auto
  let ?f = λ (x::nat). (2::nat)^x
  from set-to-nat-def have set-to-nat (D ∪ {x}) = sum ?f (D ∪ {x}) by auto
  also from D-finite S1 have ... = ?f x + sum ?f D by simp
  also from set-to-nat-def have ... = 2 ^ x + set-to-nat D by auto
  finally have set-to-nat (D ∪ {x}) = set-to-nat D + 2 ^ x by auto
  with D-def show ?thesis by auto
qed

lemma c-insert-df: c-insert = (λ x u. set-to-nat ((nat-to-set u) ∪ {x}))
proof (rule ext, rule ext)
  fix x u show c-insert x u = set-to-nat (nat-to-set u ∪ {x})
  proof (cases)
    assume A: x ∈ nat-to-set u
    then have nat-to-set u ∪ {x} = nat-to-set u by auto
    then have S1: set-to-nat (nat-to-set u ∪ {x}) = u by auto
    from A have c-in x u = 1 by (simp add: x-in-u-eq)
    then have c-insert x u = u by (unfold c-insert-def, simp)
    with S1 show ?thesis by auto
  next
    assume A: x ∉ nat-to-set u
    then have S1: c-in x u ≠ 1 by (simp add: x-in-u-eq)
    then have S2: c-insert x u = u + 2 ^ x by (unfold c-insert-def, simp)
```

```
   from A have set-to-nat (nat-to-set u ∪ {x}) = u + 2 ^ x by (rule insert-lemma)
    with S2 show ?thesis by auto
  qed
qed

definition
  c-remove :: nat ⇒ nat ⇒ nat where
  c-remove = (λ x u. if c-in x u = 0 then u else u − 2^x)

lemma c-remove-is-pr: c-remove ∈ PrimRec2
proof (unfold c-remove-def, rule if-eq-is-pr2)
  show c-in ∈ PrimRec2 by (rule c-in-is-pr)
next
  show (λx y. 0) ∈ PrimRec2 by (rule const-is-pr-2)
next
  show (λx y. y) ∈ PrimRec2 by (rule pr-id2-2)
next
  from power-is-pr show (λx y. y − 2 ^ x) ∈ PrimRec2 by prec
qed

lemma remove-lemma: x ∈ nat-to-set u ⟹ set-to-nat (nat-to-set u − {x}) = u − 2 ^ x
proof −
  assume A: x ∈ nat-to-set u
  define D where D = nat-to-set u − {x}
  from A D-def have S1: x ∉ D by auto
  have finite (nat-to-set u) by (rule nat-to-set-is-finite)
  with D-def have D-finite: finite D by auto
  let ?f = λ (x::nat). (2::nat)^x
  from set-to-nat-def have set-to-nat (D ∪ {x}) = sum ?f (D ∪ {x}) by auto
  also from D-finite S1 have . . . = ?f x + sum ?f D by simp
  also from set-to-nat-def have . . . = 2 ^ x + set-to-nat D by auto
  finally have S2: set-to-nat (D ∪ {x}) = set-to-nat D + 2 ^ x by auto
  from A D-def have D ∪ {x} = nat-to-set u by auto
  with S2 have S3: u = set-to-nat D + 2 ^ x by auto
  from A have S4: 2 ^ x ≤ u by (rule nat-to-set-upper-bound)
  with S3 D-def show ?thesis by auto
qed

lemma c-remove-df: c-remove = (λ x u. set-to-nat ((nat-to-set u) − {x}))
proof (rule ext, rule ext)
  fix x u show c-remove x u = set-to-nat (nat-to-set u − {x})
  proof (cases)
    assume A: x ∈ nat-to-set u
    then have S1: c-in x u = 1 by (simp add: x-in-u-eq)
    then have S2: c-remove x u = u − 2^x by (simp add: c-remove-def)
      from A have set-to-nat (nat-to-set u − {x}) = u − 2 ^ x by (rule remove-lemma)
    with S2 show ?thesis by auto
```

```
  next
    assume A: x ∉ nat-to-set u
    then have S1: c-in x u ≠ 1 by (simp add: x-in-u-eq)
    then have S2: c-remove x u = u by (simp add: c-remove-def c-in-def)
    from A have nat-to-set u − {x} = nat-to-set u by auto
    with S2 show ?thesis by auto
  qed
qed

definition
  c-union :: nat ⇒ nat ⇒ nat where
  c-union = (λ u v. set-to-nat (nat-to-set u ∪ nat-to-set v))

theorem c-union-is-pr: c-union ∈ PrimRec2
proof −
  define f where f y x = set-to-nat ((nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set (c-snd
x). z < y})
    for y x
  have f-is-pr: f ∈ PrimRec2
  proof −
    define g where g = c-fst
    from c-fst-is-pr g-def have g-is-pr: g ∈ PrimRec1 by auto
    define h where h a b c = (if c-in a (c-snd c) = 1 then c-insert a b else b) for
a b c
   from c-in-is-pr c-insert-is-pr have h-is-pr: h ∈ PrimRec3 unfolding h-def by
prec
    have f-at-0: ∀ x. f 0 x = g x
    proof
      fix x show f 0 x = g x by (unfold f-def, unfold g-def, simp)
    qed
    have f-at-Suc: ∀ x y. f (Suc y) x = h y (f y x) x
    proof (rule allI, rule allI)
      fix x y show f (Suc y) x = h y (f y x) x
      proof (cases)
        assume A: c-in y (c-snd x) = 1
        then have S1: y ∈ (nat-to-set (c-snd x)) by (simp add: x-in-u-eq)
        from A h-def have S2: h y (f y x) x = c-insert y (f y x) by auto
        from S1 have S3: {z ∈ nat-to-set (c-snd x). z < Suc y} = {z ∈ nat-to-set
(c-snd x). z < y} ∪ {y} by auto
           from nat-to-set-is-finite have S4: finite ((nat-to-set (c-fst x)) ∪ {z ∈
nat-to-set (c-snd x). z < y}) by auto
        with nat-to-set-srj f-def have S5: nat-to-set (f y x) = (nat-to-set (c-fst x))
∪ {z ∈ nat-to-set (c-snd x). z < y} by auto
        from f-def have S6: f (Suc y) x = set-to-nat ((nat-to-set (c-fst x)) ∪ {z ∈
nat-to-set (c-snd x). z < Suc y}) by simp
       also from S3 have ... = set-to-nat (((nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set
(c-snd x). z < y}) ∪ {y}) by auto
        also from S5 have ... = set-to-nat (nat-to-set (f y x) ∪ {y}) by auto
        also have ... = c-insert y (f y x) by (simp add: c-insert-df)
```

```
        finally show ?thesis by (simp add: S2)
      next
        assume A: ¬ c-in y (c-snd x) = 1
        then have S1: y ∉ (nat-to-set (c-snd x)) by (simp add: x-in-u-eq)
        from A h-def have S2: h y (f y x) x = f y x by auto
        have S3: {z ∈ nat-to-set (c-snd x). z < Suc y} = {z ∈ nat-to-set (c-snd x). z < y}
        proof -
          have {z ∈ nat-to-set (c-snd x). z < Suc y} = {z ∈ nat-to-set (c-snd x). z < y} ∪ {z ∈ nat-to-set (c-snd x). z = y}
            by auto
          with S1 show ?thesis by auto
        qed
          from nat-to-set-is-finite have S4: finite ((nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set (c-snd x). z < y}) by auto
        with nat-to-set-srj f-def have S5: nat-to-set (f y x) = (nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set (c-snd x). z < y} by auto
        from f-def have S6: f (Suc y) x = set-to-nat ((nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set (c-snd x). z < Suc y}) by simp
        also from S3 have ... = set-to-nat (((nat-to-set (c-fst x)) ∪ {z ∈ nat-to-set (c-snd x). z < y})) by auto
        also from S5 have ... = set-to-nat (nat-to-set (f y x)) by auto
        also have ... = f y x by simp
        finally show ?thesis by (simp add: S2)
      qed
    qed
    from g-is-pr h-is-pr f-at-0 f-at-Suc show ?thesis by (rule pr-rec-scheme)
  qed
  define union where union u v = f v (c-pair u v) for u v
  from f-is-pr have union-is-pr: union ∈ PrimRec2 unfolding union-def by prec
  have ⋀ u v. union u v = set-to-nat (nat-to-set u ∪ nat-to-set v)
  proof -
    fix u v show union u v = set-to-nat (nat-to-set u ∪ nat-to-set v)
    proof -
      from nat-to-set-upper-bound1 have {z ∈ nat-to-set v. z < v} = nat-to-set v by auto
      with union-def f-def show ?thesis by auto
    qed
  qed
  then have union = (λ u v. set-to-nat (nat-to-set u ∪ nat-to-set v)) by (simp add: ext)
  with c-union-def have c-union = union by simp
  with union-is-pr show ?thesis by simp
qed

definition
  c-diff :: nat ⇒ nat ⇒ nat where
  c-diff = (λ u v. set-to-nat (nat-to-set u − nat-to-set v))
```

```
theorem c-diff-is-pr: c-diff ∈ PrimRec2
proof −
 define f where f y x = set-to-nat ((nat-to-set (c-fst x)) − {z ∈ nat-to-set (c-snd
x). z < y})
    for y x
  have f-is-pr: f ∈ PrimRec2
  proof −
    define g where g = c-fst
    from c-fst-is-pr g-def have g-is-pr: g ∈ PrimRec1 by auto
     define h where h a b c = (if c-in a (c-snd c) = 1 then c-remove a b else b)
for a b c
     from c-in-is-pr c-remove-is-pr have h-is-pr: h ∈ PrimRec3 unfolding h-def
by prec
    have f-at-0: ∀ x. f 0 x = g x
    proof
      fix x show f 0 x = g x by (unfold f-def, unfold g-def, simp)
    qed
    have f-at-Suc: ∀ x y. f (Suc y) x = h y (f y x) x
    proof (rule allI, rule allI)
      fix x y show f (Suc y) x = h y (f y x) x
      proof (cases)
        assume A: c-in y (c-snd x) = 1
        then have S1: y ∈ (nat-to-set (c-snd x)) by (simp add: x-in-u-eq)
        from A h-def have S2: h y (f y x) x = c-remove y (f y x) by auto
        have (nat-to-set (c-fst x)) − ({z ∈ nat-to-set (c-snd x). z < y} ∪ {y}) =
             ((nat-to-set (c-fst x)) − ({z ∈ nat-to-set (c-snd x). z < y}) − {y}) by
auto
         then have lm1: set-to-nat (nat-to-set (c-fst x) − ({z ∈ nat-to-set (c-snd
x). z < y} ∪ {y})) =
                           set-to-nat (nat-to-set (c-fst x) − {z ∈ nat-to-set (c-snd x). z <
y} − {y}) by auto
        from S1 have S3: {z ∈ nat-to-set (c-snd x). z < Suc y} = {z ∈ nat-to-set
(c-snd x). z < y} ∪ {y} by auto
           from nat-to-set-is-finite have S4: finite ((nat-to-set (c-fst x)) − {z ∈
nat-to-set (c-snd x). z < y}) by auto
        with nat-to-set-srj f-def have S5: nat-to-set (f y x) = (nat-to-set (c-fst x))
− {z ∈ nat-to-set (c-snd x). z < y} by auto
        from f-def have S6: f (Suc y) x = set-to-nat ((nat-to-set (c-fst x)) − {z ∈
nat-to-set (c-snd x). z < Suc y}) by simp
       also from S3 have ... = set-to-nat ((nat-to-set (c-fst x)) − ({z ∈ nat-to-set
(c-snd x). z < y} ∪ {y})) by auto
        also have ... = set-to-nat (((nat-to-set (c-fst x)) − ({z ∈ nat-to-set (c-snd
x). z < y}) − {y})) by (rule lm1)
        also from S5 have ... = set-to-nat (nat-to-set (f y x) − {y}) by auto
        also have ... = c-remove y (f y x) by (simp add: c-remove-df)
        finally show ?thesis by (simp add: S2)
      next
        assume A: ¬ c-in y (c-snd x) = 1
        then have S1: y ∉ (nat-to-set (c-snd x)) by (simp add: x-in-u-eq)
```

from *A h-def* **have** *S2*: *h y* (*f y x*) *x* = *f y x* **by** *auto*
have *S3*: {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *Suc y*} = {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *y*}
proof −
have {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *Suc y*} = {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *y*} ∪ {*z* ∈ *nat-to-set* (*c-snd x*). *z* = *y*}
by *auto*
with *S1* **show** *?thesis* **by** *auto*
qed
from *nat-to-set-is-finite* **have** *S4*: *finite* ((*nat-to-set* (*c-fst x*)) − {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *y*}) **by** *auto*
with *nat-to-set-srj f-def* **have** *S5*: *nat-to-set* (*f y x*) = (*nat-to-set* (*c-fst x*)) − {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *y*} **by** *auto*
from *f-def* **have** *S6*: *f* (*Suc y*) *x* = *set-to-nat* ((*nat-to-set* (*c-fst x*)) − {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *Suc y*}) **by** *simp*
also from *S3* **have** ... = *set-to-nat* (((*nat-to-set* (*c-fst x*)) − {*z* ∈ *nat-to-set* (*c-snd x*). *z* < *y*})) **by** *auto*
also from *S5* **have** ... = *set-to-nat* (*nat-to-set* (*f y x*)) **by** *auto*
also have ... = *f y x* **by** *simp*
finally show *?thesis* **by** (*simp add*: *S2*)
qed
qed
from *g-is-pr h-is-pr f-at-0 f-at-Suc* **show** *?thesis* **by** (*rule pr-rec-scheme*)
qed
define *diff* **where** *diff u v* = *f v* (*c-pair u v*) **for** *u v*
from *f-is-pr* **have** *diff-is-pr*: *diff* ∈ *PrimRec2* **unfolding** *diff-def* **by** *prec*
have ⋀ *u v*. *diff u v* = *set-to-nat* (*nat-to-set u* − *nat-to-set v*)
proof −
fix *u v* **show** *diff u v* = *set-to-nat* (*nat-to-set u* − *nat-to-set v*)
proof −
from *nat-to-set-upper-bound1* **have** {*z* ∈ *nat-to-set v*. *z* < *v*} = *nat-to-set v* **by** *auto*
with *diff-def f-def* **show** *?thesis* **by** *auto*
qed
qed
then have *diff* = (λ *u v*. *set-to-nat* (*nat-to-set u* − *nat-to-set v*)) **by** (*simp add*: *ext*)
with *c-diff-def* **have** *c-diff* = *diff* **by** *simp*
with *diff-is-pr* **show** *?thesis* **by** *simp*
qed

definition
c-intersect :: *nat* ⇒ *nat* ⇒ *nat* **where**
c-intersect = (λ *u v*. *set-to-nat* (*nat-to-set u* ∩ *nat-to-set v*))

theorem *c-intersect-is-pr*: *c-intersect* ∈ *PrimRec2*
proof −
define *f* **where** *f u v* = *c-diff* (*c-union u v*) (*c-union* (*c-diff u v*) (*c-diff v u*)) **for** *u v*

from *c-diff-is-pr c-union-is-pr* **have** *f-is-pr*: *f* ∈ *PrimRec2* **unfolding** *f-def* **by** *prec*
have ⋀ *u v*. *f u v* = *c-intersect u v*
proof −
fix *u v* **show** *f u v* = *c-intersect u v*
proof −
let *?A* = *nat-to-set u*
let *?B* = *nat-to-set v*
have *A-fin*: *finite ?A* **by** (*rule nat-to-set-is-finite*)
have *B-fin*: *finite ?B* **by** (*rule nat-to-set-is-finite*)
have *S1*: *c-union u v* = *set-to-nat* (*?A* ∪ *?B*) **by** (*simp add*: *c-union-def*)
have *S2*: *c-diff u v* = *set-to-nat* (*?A* − *?B*) **by** (*simp add*: *c-diff-def*)
have *S3*: *c-diff v u* = *set-to-nat* (*?B* − *?A*) **by** (*simp add*: *c-diff-def*)
from *S2 A-fin B-fin* **have** *S4*: *nat-to-set* (*c-diff u v*) = *?A* − *?B* **by** (*simp add*: *nat-to-set-srj*)
from *S3 A-fin B-fin* **have** *S5*: *nat-to-set* (*c-diff v u*) = *?B* − *?A* **by** (*simp add*: *nat-to-set-srj*)
from *S4 S5* **have** *S6*: *c-union* (*c-diff u v*) (*c-diff v u*) = *set-to-nat* ((*?A* − *?B*) ∪ (*?B* − *?A*)) **by** (*simp add*: *c-union-def*)
from *S1 A-fin B-fin* **have** *S7*: *nat-to-set* (*c-union u v*) = *?A* ∪ *?B* **by** (*simp add*: *nat-to-set-srj*)
from *S6 A-fin B-fin* **have** *S8*: *nat-to-set* (*c-union* (*c-diff u v*) (*c-diff v u*)) = (*?A* − *?B*) ∪ (*?B* − *?A*) **by** (*simp add*: *nat-to-set-srj*)
from *S7 S8* **have** *S9*: *f u v* = *set-to-nat* ((*?A* ∪ *?B*) − ((*?A* − *?B*) ∪ (*?B* − *?A*))) **by** (*simp add*: *c-diff-def f-def*)
have *S10*: *?A* ∩ *?B* = (*?A* ∪ *?B*) − ((*?A* − *?B*) ∪ (*?B* − *?A*)) **by** *auto*
with *S9* **have** *S11*: *f u v* = *set-to-nat* (*?A* ∩ *?B*) **by** *auto*
have *c-intersect u v* = *set-to-nat* (*?A* ∩ *?B*) **by** (*simp add*: *c-intersect-def*)
with *S11* **show** *?thesis* **by** *auto*
qed
qed
then have *f* = *c-intersect* **by** (*simp add*: *ext*)
with *f-is-pr* **show** *?thesis* **by** *auto*
qed

end

6 The function which is universal for primitive recursive functions of one variable

theory *PRecUnGr*
imports *PRecFun2 PRecList*
begin

We introduce a particular function which is universal for primitive recursive functions of one variable.

definition
g-comp :: *nat* ⇒ *nat* ⇒ *nat* **where**

```
  g-comp c-ls key = (
    let n = c-fst key; x = c-snd key; m = c-snd n;
    m1 = c-fst m; m2 = c-snd m in
    — We have key = <n, x>; n = <?, m>; m = <m1, m2>.
    if c-assoc-have-key c-ls (c-pair m2 x) = 0 then
      (let y = c-assoc-value c-ls (c-pair m2 x) in
       if c-assoc-have-key c-ls (c-pair m1 y) = 0 then
         (let z = c-assoc-value c-ls (c-pair m1 y) in
         c-cons (c-pair key z) c-ls)
       else c-ls
      )
    else c-ls
  )

definition
  g-pair :: nat ⇒ nat ⇒ nat where
  g-pair c-ls key = (
    let n = c-fst key; x = c-snd key; m = c-snd n;
    m1 = c-fst m; m2 = c-snd m in
    — We have key = <n, x>; n = <?, m>; m = <m1, m2>.
    if c-assoc-have-key c-ls (c-pair m1 x) = 0 then
      (let y1 = c-assoc-value c-ls (c-pair m1 x) in
       if c-assoc-have-key c-ls (c-pair m2 x) = 0 then
         (let y2 = c-assoc-value c-ls (c-pair m2 x) in
         c-cons (c-pair key (c-pair y1 y2)) c-ls)
       else c-ls
      )
    else c-ls
  )

definition
  g-rec :: nat ⇒ nat ⇒ nat where
  g-rec c-ls key = (
    let n = c-fst key; x = c-snd key; m = c-snd n;
    m1 = c-fst m; m2 = c-snd m; y1 = c-fst x; x1 = c-snd x in
    — We have key = <n, x>; n = <?, m>; m = <m1, m2>; x = <y1, x1>.
    if y1 = 0 then
    (
      if c-assoc-have-key c-ls (c-pair m1 x1) = 0 then
        c-cons (c-pair key (c-assoc-value c-ls (c-pair m1 x1))) c-ls
      else c-ls
    )
    else
    (
     let y2 = y1−(1::nat) in
      if c-assoc-have-key c-ls (c-pair n (c-pair y2 x1)) = 0 then
      (
        let t1 = c-assoc-value c-ls (c-pair n (c-pair y2 x1)); t2 = c-pair (c-pair y2 t1) x1 in
```

```
      if c-assoc-have-key c-ls (c-pair m2 t2) = 0 then
        c-cons (c-pair key (c-assoc-value c-ls (c-pair m2 t2))) c-ls
      else c-ls
    )
    else c-ls
  )
)

definition
  g-step :: nat ⇒ nat ⇒ nat where
  g-step c-ls key = (
    let n = c-fst key; x = c-snd key; n1 = (c-fst n) mod 7 in
    if n1 = 0 then c-cons (c-pair key 0) c-ls else
    if n1 = 1 then c-cons (c-pair key (Suc x)) c-ls else
    if n1 = 2 then c-cons (c-pair key (c-fst x)) c-ls else
    if n1 = 3 then c-cons (c-pair key (c-snd x)) c-ls else
    if n1 = 4 then g-comp c-ls key else
    if n1 = 5 then g-pair c-ls key else
    if n1 = 6 then g-rec c-ls key else
    c-ls
  )

definition
  pr-gr :: nat ⇒ nat where
  pr-gr-def: pr-gr = PrimRecOp1 0 (λ a b. g-step b (c-fst a))

lemma pr-gr-at-0: pr-gr 0 = 0 by (simp add: pr-gr-def)

lemma pr-gr-at-Suc: pr-gr (Suc x) = g-step (pr-gr x) (c-fst x) by (simp add:
pr-gr-def)

definition
  univ-for-pr :: nat ⇒ nat where
  univ-for-pr = pr-conv-2-to-1 nat-to-pr

theorem univ-is-not-pr: univ-for-pr ∉ PrimRec1
proof (rule ccontr)
  assume ¬ univ-for-pr ∉ PrimRec1 then have A1: univ-for-pr ∈ PrimRec1 by
simp
  let ?f = λ n. univ-for-pr (c-pair n n) + 1
  let ?n0 = index-of-pr ?f
  from A1 have S1: ?f ∈ PrimRec1 by prec
  then have S2: nat-to-pr ?n0 = ?f by (rule index-of-pr-is-real)
  then have S3: nat-to-pr ?n0 ?n0 = ?f ?n0 by simp
  have S4: ?f ?n0 = univ-for-pr (c-pair ?n0 ?n0) + 1 by simp
  from S3 S4 show False by (simp add: univ-for-pr-def pr-conv-2-to-1-def)
qed

definition
```

```
  c-is-sub-fun :: nat ⇒ (nat ⇒ nat) ⇒ bool where
  c-is-sub-fun ls f ⟷ (∀ x. c-assoc-have-key ls x = 0 ⟶ c-assoc-value ls x = f
x)

lemma c-is-sub-fun-lm-1: ⟦ c-is-sub-fun ls f; c-assoc-have-key ls x = 0 ⟧ ⟹
c-assoc-value ls x = f x
apply(unfold c-is-sub-fun-def)
apply(auto)
done

lemma c-is-sub-fun-lm-2: c-is-sub-fun ls f ⟹ c-is-sub-fun (c-cons (c-pair x (f x))
ls) f
proof −
  assume A1: c-is-sub-fun ls f
  show ?thesis
  proof (unfold c-is-sub-fun-def, rule allI, rule impI)
    fix xa assume A2: c-assoc-have-key (c-cons (c-pair x (f x)) ls) xa = 0 show
c-assoc-value (c-cons (c-pair x (f x)) ls) xa = f xa
    proof cases
      assume C1: xa = x
      then show c-assoc-value (c-cons (c-pair x (f x)) ls) xa = f xa by (simp add:
PRecList.c-assoc-lm-2)
    next
      assume C2: ¬ xa = x
     then have S1: c-assoc-have-key (c-cons (c-pair x (f x)) ls) xa = c-assoc-have-key
ls xa by (rule c-assoc-lm-3)
     from C2 have S2: c-assoc-value (c-cons (c-pair x (f x)) ls) xa = c-assoc-value
ls xa by (rule c-assoc-lm-4)
      from A2 S1 have S3: c-assoc-have-key ls xa = 0 by simp
      from A1 S3 have c-assoc-value ls xa = f xa by (rule c-is-sub-fun-lm-1)
      with S2 show ?thesis by simp
    qed
  qed
qed

lemma mod7-lm: (n::nat) mod 7 = 0 ∨
               (n::nat) mod 7 = 1 ∨
               (n::nat) mod 7 = 2 ∨
               (n::nat) mod 7 = 3 ∨
               (n::nat) mod 7 = 4 ∨
               (n::nat) mod 7 = 5 ∨
               (n::nat) mod 7 = 6 by arith

lemma nat-to-sch-at-pos: x > 0 ⟹ nat-to-sch x = (let u=(c-fst x) mod 7;
v=c-snd x; v1=c-fst v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2
in loc-f u sch1 sch2)
proof −
  assume A: x > 0
  show ?thesis
```

```
  proof cases
    assume A1: x = 1
    then have S1: c-fst x = 0
    proof −
      have 1 = c-pair 0 1 by (simp add: c-pair-def sf-def)
      then have c-fst 1 = c-fst (c-pair 0 1) by simp
      then have c-fst 1 = 0 by simp
      with A1 show ?thesis by simp
    qed
    from A1 have S2: nat-to-sch x = Base-zero by simp
    from S1 S2 show nat-to-sch x = (let u=(c-fst x) mod 7; v=c-snd x; v1=c-fst
v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2)
    apply(insert S1 S2)
    apply(simp add: Let-def loc-f-def)
    done
  next
    assume ¬ x = 1
    from A this have A2: x > 1 by simp
    from this have nat-to-sch x = (let u=mod7 (c-fst x); v=c-snd x; v1=c-fst v;
v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) by
(rule loc-srj-lm-2)
    from this show nat-to-sch x = (let u=(c-fst x) mod 7; v=c-snd x; v1=c-fst
v; v2 = c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) by
(simp add: mod7-def)
  qed
qed

lemma nat-to-sch-0: c-fst n mod 7 = 0 ⟹ nat-to-sch n = Base-zero
proof −
  assume A: c-fst n mod 7 = 0
  show ?thesis
  proof cases
    assume n=0
    then show nat-to-sch n = Base-zero by simp
  next
    assume ¬ n = 0 then have n > 0 by simp
    then have nat-to-sch n = (let u=(c-fst n) mod 7; v=c-snd n; v1=c-fst v; v2
= c-snd v; sch1=nat-to-sch v1; sch2=nat-to-sch v2 in loc-f u sch1 sch2) by (rule
nat-to-sch-at-pos)
    with A show nat-to-sch n = Base-zero by (simp add: Let-def loc-f-def)
  qed
qed

lemma loc-lm-1: c-fst n mod 7 ≠ 0 ⟹ n > 0
proof −
  assume A: c-fst n mod 7 ≠ 0
  have n = 0 ⟹ False
  proof −
    assume n = 0
```

then have *c-fst n mod 7 = 0* **by** (*simp add*: *c-fst-at-0*)
with *A* **show** *?thesis* **by** *simp*
qed
then have ¬ *n = 0* **by** *auto*
then show *?thesis* **by** *simp*
qed

lemma *loc-lm-2*: *c-fst n mod 7 ≠ 0 ⟹ nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v=c-snd n*; *v1=c-fst v*; *v2 = c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*)
proof −
assume *c-fst n mod 7 ≠ 0*
then have *n > 0* **by** (*rule loc-lm-1*)
then show *?thesis* **by** (*rule nat-to-sch-at-pos*)
qed

lemma *nat-to-sch-1*: *c-fst n mod 7 = 1 ⟹ nat-to-sch n = Base-suc*
proof −
assume *A1*: *c-fst n mod 7 = 1*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v=c-snd n*; *v1=c-fst v*; *v2 = c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp add*: *loc-lm-2*)
with *A1* **show** *nat-to-sch n = Base-suc* **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-sch-2*: *c-fst n mod 7 = 2 ⟹ nat-to-sch n = Base-fst*
proof −
assume *A1*: *c-fst n mod 7 = 2*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v=c-snd n*; *v1=c-fst v*; *v2 = c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp add*: *loc-lm-2*)
with *A1* **show** *nat-to-sch n = Base-fst* **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-sch-3*: *c-fst n mod 7 = 3 ⟹ nat-to-sch n = Base-snd*
proof −
assume *A1*: *c-fst n mod 7 = 3*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v=c-snd n*; *v1=c-fst v*; *v2 = c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp add*: *loc-lm-2*)
with *A1* **show** *nat-to-sch n = Base-snd* **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-sch-4*: *c-fst n mod 7 = 4 ⟹ nat-to-sch n = Comp-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))
proof −
assume *A1*: *c-fst n mod 7 = 4*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v=c-snd n*; *v1=c-fst v*; *v2 = c-snd v*; *sch1=nat-to-sch v1*; *sch2=nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp*

add: *loc-lm-2*)
with *A1* **show** *nat-to-sch n* = *Comp-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-sch-5*: *c-fst n mod 7* = *5* ⟹ *nat-to-sch n* = *Pair-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))
proof −
assume *A1*: *c-fst n mod 7* = *5*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v*=*c-snd n*; *v1*=*c-fst v*; *v2* = *c-snd v*; *sch1*=*nat-to-sch v1*; *sch2*=*nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp add*: *loc-lm-2*)
with *A1* **show** *nat-to-sch n* = *Pair-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-sch-6*: *c-fst n mod 7* = *6* ⟹ *nat-to-sch n* = *Rec-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))
proof −
assume *A1*: *c-fst n mod 7* = *6*
then have *nat-to-sch n* = (*let u*=(*c-fst n*) *mod 7*; *v*=*c-snd n*; *v1*=*c-fst v*; *v2* = *c-snd v*; *sch1*=*nat-to-sch v1*; *sch2*=*nat-to-sch v2 in loc-f u sch1 sch2*) **by** (*simp add*: *loc-lm-2*)
with *A1* **show** *nat-to-sch n* = *Rec-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*simp add*: *Let-def loc-f-def*)
qed

lemma *nat-to-pr-lm-0*: *c-fst n mod 7* = *0* ⟹ *nat-to-pr n x* = *0*
proof −
assume *A*: *c-fst n mod 7* = *0*
have *S1*: *nat-to-pr n x* = *sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
from *A* **have** *S2*: *nat-to-sch n* = *Base-zero* **by** (*rule nat-to-sch-0*)
from *S1 S2* **show** *?thesis* **by** *simp*
qed

lemma *nat-to-pr-lm-1*: *c-fst n mod 7* = *1* ⟹ *nat-to-pr n x* = *Suc x*
proof −
assume *A*: *c-fst n mod 7* = *1*
have *S1*: *nat-to-pr n x* = *sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
from *A* **have** *S2*: *nat-to-sch n* = *Base-suc* **by** (*rule nat-to-sch-1*)
from *S1 S2* **show** *?thesis* **by** *simp*
qed

lemma *nat-to-pr-lm-2*: *c-fst n mod 7* = *2* ⟹ *nat-to-pr n x* = *c-fst x*
proof −
assume *A*: *c-fst n mod 7* = *2*
have *S1*: *nat-to-pr n x* = *sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
from *A* **have** *S2*: *nat-to-sch n* = *Base-fst* **by** (*rule nat-to-sch-2*)
from *S1 S2* **show** *?thesis* **by** *simp*

qed

lemma *nat-to-pr-lm-3*: *c-fst n mod 7 = 3* ⟹ *nat-to-pr n x = c-snd x*
proof –
  assume *A*: *c-fst n mod 7 = 3*
  have *S1*: *nat-to-pr n x = sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
  from *A* **have** *S2*: *nat-to-sch n = Base-snd* **by** (*rule nat-to-sch-3*)
  from *S1 S2* **show** *?thesis* **by** *simp*
qed

lemma *nat-to-pr-lm-4*: *c-fst n mod 7 = 4* ⟹ *nat-to-pr n x =* (*nat-to-pr* (*c-fst* (*c-snd n*)) (*nat-to-pr* (*c-snd* (*c-snd n*)) *x*))
proof –
  assume *A*: *c-fst n mod 7 = 4*
  have *S1*: *nat-to-pr n x = sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
  from *A* **have** *S2*: *nat-to-sch n = Comp-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*rule nat-to-sch-4*)
  from *S1 S2* **have** *S3*: *nat-to-pr n x = sch-to-pr* (*Comp-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))) *x* **by** *simp*
  from *S3* **have** *S4*: *nat-to-pr n x =* (*sch-to-pr* (*nat-to-sch* (*c-fst* (*c-snd n*)))) ((*sch-to-pr* (*nat-to-sch* (*c-snd* (*c-snd n*)))) *x*) **by** *simp*
  from *S4* **show** *?thesis* **by** (*simp add*: *nat-to-pr-def*)
qed

lemma *nat-to-pr-lm-5*: *c-fst n mod 7 = 5* ⟹ *nat-to-pr n x =* (*c-f-pair* (*nat-to-pr* (*c-fst* (*c-snd n*))) (*nat-to-pr* (*c-snd* (*c-snd n*)))) *x*
proof –
  assume *A*: *c-fst n mod 7 = 5*
  have *S1*: *nat-to-pr n x = sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
  from *A* **have** *S2*: *nat-to-sch n = Pair-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*rule nat-to-sch-5*)
  from *S1 S2* **have** *S3*: *nat-to-pr n x = sch-to-pr* (*Pair-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))) *x* **by** *simp*
  from *S3* **show** *?thesis* **by** (*simp add*: *nat-to-pr-def*)
qed

lemma *nat-to-pr-lm-6*: *c-fst n mod 7 = 6* ⟹ *nat-to-pr n x =* (*UnaryRecOp* (*nat-to-pr* (*c-fst* (*c-snd n*))) (*nat-to-pr* (*c-snd* (*c-snd n*)))) *x*
proof –
  assume *A*: *c-fst n mod 7 = 6*
  have *S1*: *nat-to-pr n x = sch-to-pr* (*nat-to-sch n*) *x* **by** (*simp add*: *nat-to-pr-def*)
  from *A* **have** *S2*: *nat-to-sch n = Rec-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*))) **by** (*rule nat-to-sch-6*)
  from *S1 S2* **have** *S3*: *nat-to-pr n x = sch-to-pr* (*Rec-op* (*nat-to-sch* (*c-fst* (*c-snd n*))) (*nat-to-sch* (*c-snd* (*c-snd n*)))) *x* **by** *simp*
  from *S3* **show** *?thesis* **by** (*simp add*: *nat-to-pr-def*)
qed

lemma *univ-for-pr-lm-0*: *c-fst* (*c-fst key*) *mod 7 = 0* ⟹ *univ-for-pr key = 0*

proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *0*
  have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
  with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-0*)
qed

lemma *univ-for-pr-lm-1*: *c-fst* (*c-fst key*) *mod 7* = *1* ⟹ *univ-for-pr key* = *Suc* (*c-snd key*)
proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *1*
  have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
  with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-1*)
qed

lemma *univ-for-pr-lm-2*: *c-fst* (*c-fst key*) *mod 7* = *2* ⟹ *univ-for-pr key* = *c-fst* (*c-snd key*)
proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *2*
  have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
  with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-2*)
qed

lemma *univ-for-pr-lm-3*: *c-fst* (*c-fst key*) *mod 7* = *3* ⟹ *univ-for-pr key* = *c-snd* (*c-snd key*)
proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *3*
  have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
  with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-3*)
qed

lemma *univ-for-pr-lm-4*: *c-fst* (*c-fst key*) *mod 7* = *4* ⟹ *univ-for-pr key* = (*nat-to-pr* (*c-fst* (*c-snd* (*c-fst key*))) (*nat-to-pr* (*c-snd* (*c-snd* (*c-fst key*))) (*c-snd key*)))
proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *4*
  have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
  with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-4*)

qed

lemma *univ-for-pr-lm-4-1*: *c-fst* (*c-fst key*) *mod 7* = *4* ⟹ *univ-for-pr key* = *univ-for-pr* (*c-pair* (*c-fst* (*c-snd* (*c-fst key*))) (*univ-for-pr* (*c-pair* (*c-snd* (*c-snd* (*c-fst key*))) (*c-snd key*))))
proof −
  assume *A*: *c-fst* (*c-fst key*) *mod 7* = *4*

have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-4 univ-for-pr-def pr-conv-2-to-1-def*)
qed

lemma *univ-for-pr-lm-5*: *c-fst* (*c-fst key*) *mod 7* = *5* ⟹ *univ-for-pr key* = *c-pair* (*univ-for-pr* (*c-pair* (*c-fst* (*c-snd* (*c-fst key*))) (*c-snd key*))) (*univ-for-pr* (*c-pair* (*c-snd* (*c-snd* (*c-fst key*))) (*c-snd key*)))
proof −
assume *A*: *c-fst* (*c-fst key*) *mod 7* = *5*
have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
with *A* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-5 c-f-pair-def univ-for-pr-def pr-conv-2-to-1-def*)
qed

lemma *univ-for-pr-lm-6-1*: ⟦*c-fst* (*c-fst key*) *mod 7* = *6*; *c-fst* (*c-snd key*) = *0*⟧ ⟹ *univ-for-pr key* = *univ-for-pr* (*c-pair* (*c-fst* (*c-snd* (*c-fst key*))) (*c-snd* (*c-snd key*)))
proof −
assume *A1*: *c-fst* (*c-fst key*) *mod 7* = *6*
assume *A2*: *c-fst* (*c-snd key*) = *0*
have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
with *A1 A2* **show** *?thesis* **by** (*simp add*: *nat-to-pr-lm-6 UnaryRecOp-def univ-for-pr-def pr-conv-2-to-1-def*)
qed

lemma *univ-for-pr-lm-6-2*: ⟦*c-fst* (*c-fst key*) *mod 7* = *6*; *c-fst* (*c-snd key*) = *Suc u*⟧ ⟹ *univ-for-pr key* = *univ-for-pr*
(*c-pair* (*c-snd* (*c-snd* (*c-fst key*)))
(*c-pair* (*c-pair u* (*univ-for-pr* (*c-pair* (*c-fst key*) (*c-pair u* (*c-snd* (*c-snd key*)))))) (*c-snd* (*c-snd key*))))
proof −
assume *A1*: *c-fst* (*c-fst key*) *mod 7* = *6*
assume *A2*: *c-fst* (*c-snd key*) = *Suc u*
have *S1*: *univ-for-pr key* = *nat-to-pr* (*c-fst key*) (*c-snd key*) **by** (*simp add*: *univ-for-pr-def pr-conv-2-to-1-def*)
with *A1 A2* **show** *?thesis*
apply(*simp add*: *nat-to-pr-lm-6 UnaryRecOp-def univ-for-pr-def pr-conv-2-to-1-def*)
apply(*simp add*: *pr-conv-1-to-3-def*)
done
qed

lemma *univ-for-pr-lm-6-3*: ⟦*c-fst* (*c-fst key*) *mod 7* = *6*; *c-fst* (*c-snd key*) ≠ *0*⟧ ⟹ *univ-for-pr key* = *univ-for-pr*
(*c-pair* (*c-snd* (*c-snd* (*c-fst key*)))
(*c-pair* (*c-pair* (*c-fst* (*c-snd key*) − *1*) (*univ-for-pr* (*c-pair* (*c-fst key*) (*c-pair* (*c-fst* (*c-snd key*) − *1*) (*c-snd* (*c-snd key*)))))) (*c-snd* (*c-snd key*))))

```
proof -
  assume A1: c-fst (c-fst key) mod 7 = 6
  assume A2: c-fst (c-snd key) ≠ 0 then have
  A3: c-fst (c-snd key) > 0 by simp
  let ?u = c-fst (c-snd key) - (1::nat)
  from A3 have S1: c-fst (c-snd key) = Suc ?u by simp
  from A1 S1 have S2: univ-for-pr key = univ-for-pr
          (c-pair (c-snd (c-snd (c-fst key)))
               (c-pair (c-pair ?u (univ-for-pr (c-pair (c-fst key) (c-pair ?u (c-snd
(c-snd key)))))) (c-snd (c-snd key)))) by (rule univ-for-pr-lm-6-2)
  thus ?thesis by simp
qed

lemma g-comp-lm-0: ⟦ c-fst (c-fst key) mod 7 = 4; c-is-sub-fun ls univ-for-pr;
g-comp ls key ≠ ls⟧ ⟹ g-comp ls key = c-cons (c-pair key (univ-for-pr key)) ls
proof -
  assume A1: c-fst (c-fst key) mod 7 = 4
  assume A2: c-is-sub-fun ls univ-for-pr
  assume A3: g-comp ls key ≠ ls
  let ?n = c-fst key
  let ?x = c-snd key
  let ?m = c-snd ?n
  let ?m1 = c-fst ?m
  let ?m2 = c-snd ?m
  let ?k1 = c-pair ?m2 ?x
  have S1: c-assoc-have-key ls ?k1 = 0
  proof (rule ccontr)
    assume A1-1: c-assoc-have-key ls ?k1 ≠ 0
    then have g-comp ls key = ls by(simp add: g-comp-def)
    with A3 show False by simp
  qed
  let ?y = c-assoc-value ls ?k1
  from A2 S1 have S2: ?y = univ-for-pr ?k1 by (rule c-is-sub-fun-lm-1)
  let ?k2 = c-pair ?m1 ?y
  have S3: c-assoc-have-key ls ?k2 = 0
  proof (rule ccontr)
    assume A3-1: c-assoc-have-key ls ?k2 ≠ 0
    then have g-comp ls key = ls by (simp add: g-comp-def Let-def)
    with A3 show False by simp
  qed
  let ?z = c-assoc-value ls ?k2
  from A2 S3 have S4: ?z = univ-for-pr ?k2 by (rule c-is-sub-fun-lm-1)
  from S2 have S5: ?k2 = c-pair ?m1 (univ-for-pr ?k1) by simp
  from S4 S5 have S6: ?z = univ-for-pr (c-pair ?m1 (univ-for-pr ?k1)) by simp
  from A1 S6 have S7: ?z = univ-for-pr key by (simp add: univ-for-pr-lm-4-1)
  from S1 S3 S7 show ?thesis by (simp add: g-comp-def Let-def)
qed

lemma g-comp-lm-1: ⟦ c-fst (c-fst key) mod 7 = 4; c-is-sub-fun ls univ-for-pr⟧
```

```
⟹ c-is-sub-fun (g-comp ls key) univ-for-pr
proof –
  assume A1: c-fst (c-fst key) mod 7 = 4
  assume A2: c-is-sub-fun ls univ-for-pr
  show ?thesis
  proof cases
    assume g-comp ls key = ls
    with A2 show c-is-sub-fun (g-comp ls key) univ-for-pr by simp
  next
    assume g-comp ls key ≠ ls
     from A1 A2 this have S1: g-comp ls key = c-cons (c-pair key (univ-for-pr
key)) ls by (rule g-comp-lm-0)
   with A2 show c-is-sub-fun (g-comp ls key) univ-for-pr by (simp add: c-is-sub-fun-lm-2)
  qed
qed

lemma g-pair-lm-0: ⟦ c-fst (c-fst key) mod 7 = 5; c-is-sub-fun ls univ-for-pr; g-pair
ls key ≠ ls⟧ ⟹ g-pair ls key = c-cons (c-pair key (univ-for-pr key)) ls
proof –
  assume A1: c-fst (c-fst key) mod 7 = 5
  assume A2: c-is-sub-fun ls univ-for-pr
  assume A3: g-pair ls key ≠ ls
  let ?n = c-fst key
  let ?x = c-snd key
  let ?m = c-snd ?n
  let ?m1 = c-fst ?m
  let ?m2 = c-snd ?m
  let ?k1 = c-pair ?m1 ?x
  have S1: c-assoc-have-key ls ?k1 = 0
  proof (rule ccontr)
    assume A1-1: c-assoc-have-key ls ?k1 ≠ 0
    then have g-pair ls key = ls by(simp add: g-pair-def)
    with A3 show False by simp
  qed
  let ?y1 = c-assoc-value ls ?k1
  from A2 S1 have S2: ?y1 = univ-for-pr ?k1 by (rule c-is-sub-fun-lm-1)
  let ?k2 = c-pair ?m2 ?x
  have S3: c-assoc-have-key ls ?k2 = 0
  proof (rule ccontr)
    assume A3-1: c-assoc-have-key ls ?k2 ≠ 0
    then have g-pair ls key = ls by (simp add: g-pair-def Let-def)
    with A3 show False by simp
  qed
  let ?y2 = c-assoc-value ls ?k2
  from A2 S3 have S4: ?y2 = univ-for-pr ?k2 by (rule c-is-sub-fun-lm-1)
  let ?z = c-pair ?y1 ?y2
  from S2 S4 have S5: ?z = c-pair (univ-for-pr ?k1) (univ-for-pr ?k2) by simp
  from A1 S5 have S6: ?z = univ-for-pr key by (simp add: univ-for-pr-lm-5)
  from S1 S3 S6 show ?thesis by (simp add: g-pair-def Let-def)
```

```
qed

lemma g-pair-lm-1: ⟦ c-fst (c-fst key) mod 7 = 5; c-is-sub-fun ls univ-for-pr⟧ ⟹
c-is-sub-fun (g-pair ls key) univ-for-pr
proof −
  assume A1: c-fst (c-fst key) mod 7 = 5
  assume A2: c-is-sub-fun ls univ-for-pr
  show ?thesis
  proof cases
    assume g-pair ls key = ls
    with A2 show c-is-sub-fun (g-pair ls key) univ-for-pr by simp
  next
    assume g-pair ls key ≠ ls
   from A1 A2 this have S1: g-pair ls key = c-cons (c-pair key (univ-for-pr key))
ls by (rule g-pair-lm-0)
   with A2 show c-is-sub-fun (g-pair ls key) univ-for-pr by (simp add: c-is-sub-fun-lm-2)
  qed
qed

lemma g-rec-lm-0: ⟦ c-fst (c-fst key) mod 7 = 6; c-is-sub-fun ls univ-for-pr; g-rec
ls key ≠ ls⟧ ⟹ g-rec ls key = c-cons (c-pair key (univ-for-pr key)) ls
proof −
  assume A1: c-fst (c-fst key) mod 7 = 6
  assume A2: c-is-sub-fun ls univ-for-pr
  assume A3: g-rec ls key ≠ ls
  let ?n = c-fst key
  let ?x = c-snd key
  let ?m = c-snd ?n
  let ?m1 = c-fst ?m
  let ?m2 = c-snd ?m
  let ?y1 = c-fst ?x
  let ?x1 = c-snd ?x
  show ?thesis
  proof cases
    assume A1-1: ?y1 = 0
    let ?k1 = c-pair ?m1 ?x1
    have S1-1: c-assoc-have-key ls ?k1 = 0
    proof (rule ccontr)
      assume c-assoc-have-key ls ?k1 ≠ 0
      with A1-1 have g-rec ls key = ls by(simp add: g-rec-def)
      with A3 show False by simp
    qed
    let ?v = c-assoc-value ls ?k1
    from A2 S1-1 have S1-2: ?v = univ-for-pr ?k1 by (rule c-is-sub-fun-lm-1)
   from A1 A1-1 S1-2 have S1-3: ?v = univ-for-pr key by (simp add: univ-for-pr-lm-6-1)
    from A1-1 S1-1 S1-3 show ?thesis by (simp add: g-rec-def Let-def)
  next
    assume A2-1: ?y1 ≠ 0 then have A2-2: ?y1 > 0 by simp
    let ?y2 = ?y1 − (1::nat)
```

```
    let ?k2 = c-pair ?n (c-pair ?y2 ?x1)
    have S2-1: c-assoc-have-key ls ?k2 = 0
    proof (rule ccontr)
      assume c-assoc-have-key ls ?k2 ≠ 0
      with A2-1 have g-rec ls key = ls by (simp add: g-rec-def Let-def)
      with A3 show False by simp
    qed
    let ?t1 = c-assoc-value ls ?k2
    from A2 S2-1 have S2-2: ?t1 = univ-for-pr ?k2 by (rule c-is-sub-fun-lm-1)
    let ?t2 = c-pair (c-pair ?y2 ?t1) ?x1
    let ?k3 = c-pair ?m2 ?t2
    have S2-3: c-assoc-have-key ls ?k3 = 0
    proof (rule ccontr)
      assume c-assoc-have-key ls ?k3 ≠ 0
      with A2-1 have g-rec ls key = ls by (simp add: g-rec-def Let-def)
      with A3 show False by simp
    qed
    let ?u = c-assoc-value ls ?k3
    from A2 S2-3 have S2-4: ?u = univ-for-pr ?k3 by (rule c-is-sub-fun-lm-1)
    from S2-4 S2-2 have S2-5: ?u = univ-for-pr (c-pair ?m2 (c-pair (c-pair ?y2
(univ-for-pr ?k2)) ?x1)) by simp
   from A1 A2-1 S2-5 have S2-6: ?u = univ-for-pr key by (simp add: univ-for-pr-lm-6-3)
    from A2-1 S2-1 S2-3 S2-6 show ?thesis by (simp add: g-rec-def Let-def)
  qed
qed

lemma g-rec-lm-1: ⟦ c-fst (c-fst key) mod 7 = 6; c-is-sub-fun ls univ-for-pr⟧ ⟹
c-is-sub-fun (g-rec ls key) univ-for-pr
proof −
  assume A1: c-fst (c-fst key) mod 7 = 6
  assume A2: c-is-sub-fun ls univ-for-pr
  show ?thesis
  proof cases
    assume g-rec ls key = ls
    with A2 show c-is-sub-fun (g-rec ls key) univ-for-pr by simp
  next
    assume g-rec ls key ≠ ls
    from A1 A2 this have S1: g-rec ls key = c-cons (c-pair key (univ-for-pr key))
ls by (rule g-rec-lm-0)
   with A2 show c-is-sub-fun (g-rec ls key) univ-for-pr by (simp add: c-is-sub-fun-lm-2)
  qed
qed

lemma g-step-lm-0: c-fst (c-fst key) mod 7 = 0 ⟹ g-step ls key = c-cons (c-pair
key 0) ls by (simp add: g-step-def)

lemma g-step-lm-1: c-fst (c-fst key) mod 7 = 1 ⟹ g-step ls key = c-cons (c-pair
key (Suc (c-snd key))) ls by (simp add: g-step-def Let-def)
```

lemma *g-step-lm-2*: *c-fst* (*c-fst key*) *mod 7* = *2* ⟹ *g-step ls key* = *c-cons* (*c-pair key* (*c-fst* (*c-snd key*))) *ls* **by** (*simp add*: *g-step-def Let-def*)

lemma *g-step-lm-3*: *c-fst* (*c-fst key*) *mod 7* = *3* ⟹ *g-step ls key* = *c-cons* (*c-pair key* (*c-snd* (*c-snd key*))) *ls* **by** (*simp add*: *g-step-def Let-def*)

lemma *g-step-lm-4*: *c-fst* (*c-fst key*) *mod 7* = *4* ⟹ *g-step ls key* = *g-comp ls key* **by** (*simp add*: *g-step-def*)

lemma *g-step-lm-5*: *c-fst* (*c-fst key*) *mod 7* = *5* ⟹ *g-step ls key* = *g-pair ls key* **by** (*simp add*: *g-step-def*)

lemma *g-step-lm-6*: *c-fst* (*c-fst key*) *mod 7* = *6* ⟹ *g-step ls key* = *g-rec ls key* **by** (*simp add*: *g-step-def*)

lemma *g-step-lm-7*: *c-is-sub-fun ls univ-for-pr* ⟹ *c-is-sub-fun* (*g-step ls key*) *univ-for-pr*
proof −
assume *A1*: *c-is-sub-fun ls univ-for-pr*
let *?n* = *c-fst key*
let *?x* = *c-snd key*
let *?n1* = (*c-fst ?n*) *mod 7*
have *S1*: *?n1* = *0* ⟹ *?thesis*
proof −
assume *A*: *?n1* = *0*
then have *S1-1*: *g-step ls key* = *c-cons* (*c-pair key 0*) *ls* **by** (*rule g-step-lm-0*)
from *A* **have** *S1-2*: *univ-for-pr key* = *0* **by** (*rule univ-for-pr-lm-0*)
from *A1* **have** *S1-3*: *c-is-sub-fun* (*c-cons* (*c-pair key* (*univ-for-pr key*)) *ls*) *univ-for-pr* **by** (*rule c-is-sub-fun-lm-2*)
from *S1-3 S1-1 S1-2* **show** *?thesis* **by** *simp*
qed
have *S2*: *?n1* = *1* ⟹ *?thesis*
proof −
assume *A*: *?n1* = *1*
then have *S2-1*: *g-step ls key* = *c-cons* (*c-pair key* (*Suc* (*c-snd key*))) *ls* **by** (*rule g-step-lm-1*)
from *A* **have** *S2-2*: *univ-for-pr key* = *Suc* (*c-snd key*) **by** (*rule univ-for-pr-lm-1*)
from *A1* **have** *S2-3*: *c-is-sub-fun* (*c-cons* (*c-pair key* (*univ-for-pr key*)) *ls*) *univ-for-pr* **by** (*rule c-is-sub-fun-lm-2*)
from *S2-3 S2-1 S2-2* **show** *?thesis* **by** *simp*
qed
have *S3*: *?n1* = *2* ⟹ *?thesis*
proof −
assume *A*: *?n1* = *2*
then have *S2-1*: *g-step ls key* = *c-cons* (*c-pair key* (*c-fst* (*c-snd key*))) *ls* **by** (*rule g-step-lm-2*)
from *A* **have** *S2-2*: *univ-for-pr key* = *c-fst* (*c-snd key*) **by** (*rule univ-for-pr-lm-2*)
from *A1* **have** *S2-3*: *c-is-sub-fun* (*c-cons* (*c-pair key* (*univ-for-pr key*)) *ls*) *univ-for-pr* **by** (*rule c-is-sub-fun-lm-2*)

from *S2-3 S2-1 S2-2* **show** *?thesis* **by** *simp*
qed
have *S4*: *?n1 = 3* ⟹ *?thesis*
proof –
assume *A*: *?n1 = 3*
then have *S2-1*: *g-step ls key = c-cons (c-pair key (c-snd (c-snd key))) ls* **by** (*rule g-step-lm-3*)
from *A* **have** *S2-2*: *univ-for-pr key = c-snd (c-snd key)* **by** (*rule univ-for-pr-lm-3*)
from *A1* **have** *S2-3*: *c-is-sub-fun (c-cons (c-pair key (univ-for-pr key)) ls) univ-for-pr* **by** (*rule c-is-sub-fun-lm-2*)
from *S2-3 S2-1 S2-2* **show** *?thesis* **by** *simp*
qed
have *S5*: *?n1 = 4* ⟹ *?thesis*
proof –
assume *A*: *?n1 = 4*
then have *S2-1*: *g-step ls key = g-comp ls key* **by** (*rule g-step-lm-4*)
from *A A1 S2-1* **show** *?thesis* **by** (*simp add*: *g-comp-lm-1*)
qed
have *S6*: *?n1 = 5* ⟹ *?thesis*
proof –
assume *A*: *?n1 = 5*
then have *S2-1*: *g-step ls key = g-pair ls key* **by** (*rule g-step-lm-5*)
from *A A1 S2-1* **show** *?thesis* **by** (*simp add*: *g-pair-lm-1*)
qed
have *S7*: *?n1 = 6* ⟹ *?thesis*
proof –
assume *A*: *?n1 = 6*
then have *S2-1*: *g-step ls key = g-rec ls key* **by** (*rule g-step-lm-6*)
from *A A1 S2-1* **show** *?thesis* **by** (*simp add*: *g-rec-lm-1*)
qed
have *S8*: *?n1=0* ∨ *?n1=1* ∨ *?n1=2* ∨ *?n1=3* ∨ *?n1=4* ∨ *?n1=5* ∨ *?n1=6* **by** (*rule mod7-lm*)
with *S1 S2 S3 S4 S5 S6 S7* **show** *?thesis* **by** *fast*
qed

theorem *pr-gr-1*: *c-is-sub-fun (pr-gr x) univ-for-pr*
apply(*induct x*)
apply(*simp add*: *pr-gr-at-0 c-is-sub-fun-def c-assoc-have-key-df*)
apply(*simp add*: *pr-gr-at-Suc*)
apply(*simp add*: *g-step-lm-7*)
done

lemma *comp-next*: *g-comp ls key = ls* ∨ *c-tl (g-comp ls key) = ls* **by**(*simp add*: *g-comp-def Let-def*)
lemma *pair-next*: *g-pair ls key = ls* ∨ *c-tl (g-pair ls key) = ls* **by**(*simp add*: *g-pair-def Let-def*)
lemma *rec-next*: *g-rec ls key = ls* ∨ *c-tl (g-rec ls key) = ls* **by**(*simp add*: *g-rec-def Let-def*)

```
lemma step-next: g-step ls key = ls ∨ c-tl (g-step ls key) = ls
apply(simp add: g-step-def comp-next pair-next rec-next Let-def)

done

lemma lm1: pr-gr (Suc x) = pr-gr x ∨ c-tl (pr-gr (Suc x)) = pr-gr x by(simp
add: pr-gr-at-Suc step-next)

lemma c-assoc-have-key-pos: c-assoc-have-key ls x = 0 ⟹ ls > 0
proof −
  assume A1: c-assoc-have-key ls x = 0
  thus ?thesis
  proof (cases)
    assume A2: ls = 0
    then have S1: c-assoc-have-key ls x = 1 by (simp add: c-assoc-have-key-df)
    with A1 have S2: False by auto
    then show ls > 0 by auto
  next
    assume A3: ¬ ls = 0
    then show ls > 0 by auto
  qed
qed

lemma lm2: c-assoc-have-key (c-tl ls) key = 0 ⟹ c-assoc-have-key ls key = 0
proof −
  assume A1: c-assoc-have-key (c-tl ls) key = 0
  from A1 have S1: c-tl ls > 0 by (rule c-assoc-have-key-pos)
  have S2: c-tl ls ≤ ls by (rule c-tl-le)
  from S1 S2 have S3: ls ≠ 0 by auto
  from A1 S3 show ?thesis by (auto simp add: c-assoc-have-key-lm-1)
qed

lemma lm3: c-assoc-have-key (pr-gr x) key = 0 ⟹ c-assoc-have-key (pr-gr (Suc
x)) key = 0
proof −
  assume A1: c-assoc-have-key (pr-gr x) key = 0
  have S1: pr-gr (Suc x) = pr-gr x ∨ c-tl (pr-gr (Suc x)) = pr-gr x by (rule lm1)
  from A1 have S2: pr-gr (Suc x) = pr-gr x ⟹ ?thesis by auto
  have S3: c-tl (pr-gr (Suc x)) = pr-gr x ⟹ ?thesis
  proof −
    assume c-tl (pr-gr (Suc x)) = pr-gr x (is c-tl ?ls = -)
    with A1 have c-assoc-have-key (c-tl ?ls) key = 0 by auto
    then show c-assoc-have-key ?ls key = 0 by (rule lm2)
  qed
  from S1 S2 S3 show ?thesis by auto
qed

lemma lm4: ⟦ c-assoc-have-key (pr-gr x) key = 0; 0 ≤ y⟧ ⟹ c-assoc-have-key
(pr-gr (x+y)) key = 0
```

```
apply(induct-tac y)
apply(auto)
apply(simp add: lm3)
done

lemma lm5: ⟦ c-assoc-have-key (pr-gr x) key = 0; x ≤ y ⟧ ⟹ c-assoc-have-key
(pr-gr y) key = 0
proof −
  assume A1: c-assoc-have-key (pr-gr x) key = 0
  assume A2: x ≤ y
  let ?z = y−x
  from A2 have S1: 0 ≤ ?z by auto
  from A2 have S2: y = x + ?z by auto
  from A1 S1 have S3: c-assoc-have-key (pr-gr (x+?z)) key = 0 by (rule lm4)
  from S2 S3 show ?thesis by auto
qed

lemma loc-upb-lm-1: n = 0 ⟹ (c-fst n) mod 7 = 0
apply(simp add: c-fst-at-0)
done

lemma loc-upb-lm-2: (c-fst n) mod 7 > 1 ⟹ c-snd n < n
proof −
  assume A1: c-fst n mod 7 > 1
  from A1 have S1: 1 < c-fst n by simp
  have S2: c-fst n ≤ n by (rule c-fst-le-arg)
  from S1 S2 have S3: 1 < n by simp
  from S3 have S4: n>1 by simp
  from S4 show ?thesis by (rule c-snd-less-arg)
qed

lemma loc-upb-lm-2-0: (c-fst n) mod 7 = 4 ⟶ c-fst (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 4
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-fst (c-snd n) ≤  c-snd n by (rule c-fst-le-arg)
  from S1 S2 show c-fst (c-snd n) < n by auto
qed

lemma loc-upb-lm-2-2: (c-fst n) mod 7 = 4 ⟶ c-snd (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 4
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-snd (c-snd n) ≤  c-snd n by (rule c-snd-le-arg)
  from S1 S2 show c-snd (c-snd n) < n by auto
qed
```

```
lemma loc-upb-lm-2-3: (c-fst n) mod 7 = 5 ⟶ c-fst (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 5
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-fst (c-snd n) ≤ c-snd n by (rule c-fst-le-arg)
  from S1 S2 show c-fst (c-snd n) < n by auto
qed

lemma loc-upb-lm-2-4: (c-fst n) mod 7 = 5 ⟶ c-snd (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 5
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-snd (c-snd n) ≤ c-snd n by (rule c-snd-le-arg)
  from S1 S2 show c-snd (c-snd n) < n by auto
qed

lemma loc-upb-lm-2-5: (c-fst n) mod 7 = 6 ⟶ c-fst (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 6
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-fst (c-snd n) ≤ c-snd n by (rule c-fst-le-arg)
  from S1 S2 show c-fst (c-snd n) < n by auto
qed

lemma loc-upb-lm-2-6: (c-fst n) mod 7 = 6 ⟶ c-snd (c-snd n) < n
proof
  assume A1: c-fst n mod 7 = 6
  then have S0: c-fst n mod 7 > 1 by auto
  then have S1: c-snd n < n by (rule loc-upb-lm-2)
  have S2: c-snd (c-snd n) ≤ c-snd n by (rule c-snd-le-arg)
  from S1 S2 show c-snd (c-snd n) < n by auto
qed

lemma loc-upb-lm-2-7: ⟦y2 = y1 − (1::nat); 0 < y1; x1 = c-snd x; y1 = c-fst x⟧
⟹ c-pair y2 x1 < x
proof −
  assume A1: y2 = y1 − (1::nat) and A2: 0 < y1 and A3: x1 = c-snd x and
A4: y1 = c-fst x
  from A1 A2 have S1: y2 < y1 by auto
  from S1 have S2: c-pair y2 x1 < c-pair y1 x1 by (rule c-pair-strict-mono1)
  from A3 A4 have S3: c-pair y1 x1 = x by auto
  from S2 S3 show c-pair y2 x1 < x by auto
qed

function loc-upb :: nat ⇒ nat ⇒ nat where
  aa: loc-upb n x = (
```

```
    let n1 = (c-fst n) mod 7 in
      if n1 = 0 then (c-pair (c-pair n x) 0) + 1 else
      if n1 = 1 then (c-pair (c-pair n x) 0) + 1 else
      if n1 = 2 then (c-pair (c-pair n x) 0) + 1 else
      if n1 = 3 then (c-pair (c-pair n x) 0) + 1 else
      if n1 = 4 then (
      let m = c-snd n; m1 = c-fst m; m2 = c-snd m;
      y = c-assoc-value (pr-gr (loc-upb m2 x)) (c-pair m2 x) in
        (c-pair (c-pair n x) (loc-upb m2 x + loc-upb m1 y)) + 1
      ) else
      if n1 = 5 then (
      let m = c-snd n; m1 = c-fst m; m2 = c-snd m in
        (c-pair (c-pair n x) (loc-upb m1 x + loc-upb m2 x)) + 1
      ) else
      if n1 = 6 then (
     let m = c-snd n; m1 = c-fst m; m2 = c-snd m; y1 = c-fst x; x1 = c-snd x in
        if y1 = 0 then (
          (c-pair (c-pair n x) (loc-upb m1 x1)) + 1
        ) else (
          let y2 = y1−(1::nat);
             t1 = c-assoc-value (pr-gr (loc-upb n (c-pair y2 x1))) (c-pair n (c-pair
y2 x1)); t2 = c-pair (c-pair y2 t1) x1 in
            (c-pair (c-pair n x) (loc-upb n (c-pair y2 x1) + loc-upb m2 t2)) + 1
        )
      )
      else 0
 )
by auto

termination
apply (relation measure (λ m. m) <*lex*> measure (λ n. n))
apply (simp-all add: loc-upb-lm-2-0 loc-upb-lm-2-2 loc-upb-lm-2-3 loc-upb-lm-2-4
loc-upb-lm-2-5 loc-upb-lm-2-6 loc-upb-lm-2-7)
apply auto
done

definition
  lex-p :: ((nat × nat) × nat × nat) set where
  lex-p = ((measure (λ m. m)) <*lex*> (measure (λ n. n)))

lemma wf-lex-p: wf(lex-p)
apply(simp add: lex-p-def)
apply(auto)
done

lemma lex-p-eq: ((n′,x′), (n,x)) ∈ lex-p = (n′<n ∨ n′=n ∧ x′<x)
apply(simp add: lex-p-def)
done
```

lemma *loc-upb-lex-0*: *c-fst n mod 7 = 0 ⟹ c-assoc-have-key* (*pr-gr* (*loc-upb n x*)) (*c-pair n x*) = *0*
proof −
  assume *A1*: *c-fst n mod 7 = 0*
  let *?key = c-pair n x*
  let *?s = c-pair ?key 0*
  let *?ls = pr-gr ?s*
  from *A1* **have** *loc-upb n x = ?s + 1* **by** *simp*
  then have *S1*: *pr-gr* (*loc-upb n x*) = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*simp add*: *pr-gr-at-Suc*)
  from *A1* **have** *S2*: *g-step ?ls ?key = c-cons* (*c-pair ?key 0*) *?ls* **by** (*simp add*: *g-step-def*)
  from *S1 S2* **have** *pr-gr* (*loc-upb n x*) = *c-cons* (*c-pair ?key 0*) *?ls* **by** *auto*
  thus *?thesis* **by** (*simp add*: *c-assoc-lm-1*)
qed

lemma *loc-upb-lex-1*: *c-fst n mod 7 = 1 ⟹ c-assoc-have-key* (*pr-gr* (*loc-upb n x*)) (*c-pair n x*) = *0*
proof −
  assume *A1*: *c-fst n mod 7 = 1*
  let *?key = c-pair n x*
  let *?s = c-pair ?key 0*
  let *?ls = pr-gr ?s*
  from *A1* **have** *loc-upb n x = ?s + 1* **by** *simp*
  then have *S1*: *pr-gr* (*loc-upb n x*) = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*simp add*: *pr-gr-at-Suc*)
  from *A1* **have** *S2*: *g-step ?ls ?key = c-cons* (*c-pair ?key* (*Suc x*)) *?ls* **by** (*simp add*: *g-step-def*)
  from *S1 S2* **have** *pr-gr* (*loc-upb n x*) = *c-cons* (*c-pair ?key* (*Suc x*)) *?ls* **by** *auto*
  thus *?thesis* **by** (*simp add*: *c-assoc-lm-1*)
qed

lemma *loc-upb-lex-2*: *c-fst n mod 7 = 2 ⟹ c-assoc-have-key* (*pr-gr* (*loc-upb n x*)) (*c-pair n x*) = *0*
proof −
  assume *A1*: *c-fst n mod 7 = 2*
  let *?key = c-pair n x*
  let *?s = c-pair ?key 0*
  let *?ls = pr-gr ?s*
  from *A1* **have** *loc-upb n x = ?s + 1* **by** *simp*
  then have *S1*: *pr-gr* (*loc-upb n x*) = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*simp add*: *pr-gr-at-Suc*)
  from *A1* **have** *S2*: *g-step ?ls ?key = c-cons* (*c-pair ?key* (*c-fst x*)) *?ls* **by** (*simp add*: *g-step-def*)
  from *S1 S2* **have** *pr-gr* (*loc-upb n x*) = *c-cons* (*c-pair ?key* (*c-fst x*)) *?ls* **by** *auto*
  thus *?thesis* **by** (*simp add*: *c-assoc-lm-1*)
qed

lemma *loc-upb-lex-3*: *c-fst n mod 7 = 3 ⟹ c-assoc-have-key* (*pr-gr* (*loc-upb n*

```
x)) (c-pair n x) = 0
proof -
  assume A1: c-fst n mod 7 = 3
  let ?key = c-pair n x
  let ?s = c-pair ?key 0
  let ?ls = pr-gr ?s
  from A1 have loc-upb n x = ?s + 1 by simp
  then have S1: pr-gr (loc-upb n x) = g-step (pr-gr ?s) (c-fst ?s) by (simp add:
pr-gr-at-Suc)
  from A1 have S2: g-step ?ls ?key = c-cons (c-pair ?key (c-snd x)) ?ls by (simp
add: g-step-def)
   from S1 S2 have pr-gr (loc-upb n x) = c-cons (c-pair ?key (c-snd x)) ?ls by
auto
  thus ?thesis by (simp add: c-assoc-lm-1)
qed

lemma loc-upb-lex-4: ⟦⋀ n′ x′. ((n′,x′), (n,x)) ∈ lex-p ⟹ c-assoc-have-key (pr-gr
(loc-upb n′ x′)) (c-pair n′ x′) = 0;
                        c-fst n mod 7 = 4⟧ ⟹
                        c-assoc-have-key (pr-gr (loc-upb n x)) (c-pair n x) = 0
proof -
  assume A1: ⋀ n′ x′. ((n′,x′), (n,x)) ∈ lex-p ⟹ c-assoc-have-key (pr-gr (loc-upb
n′ x′)) (c-pair n′ x′) = 0
  assume A2: c-fst n mod 7 = 4
  let ?key = c-pair n x
  let ?m1 = c-fst (c-snd n)
  let ?m2 = c-snd (c-snd n)
  define upb1 where upb1 = loc-upb ?m2 x
  from A2 have m2-lt-n: ?m2 < n by (simp add: loc-upb-lm-2-2)
  then have M2: ((?m2, x), (n,x)) ∈ lex-p by (simp add: lex-p-eq)
   with A1 upb1-def have S1: c-assoc-have-key (pr-gr upb1) (c-pair ?m2 x) = 0
by auto
  from M2 have M2′: ((?m2, x), n, x) ∈ measure (λm. m) <*lex*> measure (λn.
n) by (simp add: lex-p-def)
  have T1: c-is-sub-fun (pr-gr upb1) univ-for-pr by (rule pr-gr-1)
  from T1 S1 have T2: c-assoc-value (pr-gr upb1) (c-pair ?m2 x) = univ-for-pr
(c-pair ?m2 x) by (rule c-is-sub-fun-lm-1)
  define y where y = c-assoc-value (pr-gr upb1) (c-pair ?m2 x)
  from T2 y-def have T3: y = univ-for-pr (c-pair ?m2 x) by auto

  define upb2 where upb2 = loc-upb ?m1 y
  from A2 have ?m1 < n by (simp add: loc-upb-lm-2-0)
  then have M1: ((?m1, y), (n,x)) ∈ lex-p by (simp add: lex-p-eq)
  with A1 have S2: c-assoc-have-key (pr-gr (loc-upb ?m1 y)) (c-pair ?m1 y) = 0
by auto
  from M1 have M1′: ((?m1, y), n, x) ∈ measure (λm. m) <*lex*> measure (λn.
n) by (simp add: lex-p-def)
   from S1 upb1-def have S3: c-assoc-have-key (pr-gr upb1) (c-pair ?m2 x) = 0
by auto
```

from *S2 upb2-def* **have** *S4*: *c-assoc-have-key* (*pr-gr upb2*) (*c-pair ?m1 y*) = *0* **by** *auto*

let *?s* = *c-pair ?key* (*upb1* + *upb2*)
let *?ls* = *pr-gr ?s*
let *?sum-upb* = *upb1* +*upb2*
from *A2* **have** *?m1* < *n* **by** (*simp add*: *loc-upb-lm-2-0*)
then have ((*?m1*, *x*), (*n*,*x*)) ∈ *lex-p* **by** (*simp add*: *lex-p-eq*)
then have *M1''*: ((*?m1*, *x*), *n*, *x*) ∈ *measure* (λ*m*. *m*) <∗*lex*∗> *measure* (λ*n*. *n*) **by** (*simp add*: *lex-p-def*)
from *A2 M2' M1''* **have** *S11*: *loc-upb n x* = (*let y* = *c-assoc-value* (*pr-gr* (*loc-upb ?m2 x*)) (*c-pair ?m2 x*)
in (*c-pair* (*c-pair n x*)
(*loc-upb ?m2 x* + *loc-upb ?m1 y*)) + *1*)
by(*simp add*: *Let-def*)
define *upb* **where** *upb* = *loc-upb n x*
from *S11 y-def upb1-def upb2-def* **have** *loc-upb n x* = *?s* + *1* **by** (*simp add*: *Let-def*)
with *upb-def* **have** *S11*: *upb* = *?s* + *1* **by** *auto*

have *S7*: *?sum-upb* ≤ *?s* **by** (*rule arg2-le-c-pair*)
have *upb1-le-s*: *upb1* ≤ *?s*
proof −
have *S1*: *upb1* ≤ *?sum-upb* **by** (*rule Nat.le-add1*)
from *S1 S7* **show** *?thesis* **by** *auto*
qed
have *upb2-le-s*: *upb2* ≤ *?s*
proof −
have *S1*: *upb2* ≤ *?sum-upb* **by** (*rule Nat.le-add2*)
from *S1 S7* **show** *?thesis* **by** *auto*
qed

have *S18*: *pr-gr upb* = *g-comp ?ls ?key*
proof −
from *S11* **have** *S1*: *pr-gr upb* = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*simp add*: *pr-gr-at-Suc*)
from *A2* **have** *S2*: *g-step ?ls ?key* = *g-comp ?ls ?key* **by** (*simp add*: *g-step-def*)
from *S1 S2* **show** *?thesis* **by** *auto*
qed

from *S3 upb1-le-s* **have** *S19*: *c-assoc-have-key ?ls* (*c-pair ?m2 x*) = *0* **by** (*rule lm5*)
from *S4 upb2-le-s* **have** *S20*: *c-assoc-have-key ?ls* (*c-pair ?m1 y*) = *0* **by** (*rule lm5*)
have *T-ls*: *c-is-sub-fun ?ls univ-for-pr* **by** (*rule pr-gr-1*)
from *T-ls S19* **have** *T-ls2*: *c-assoc-value ?ls* (*c-pair ?m2 x*) = *univ-for-pr* (*c-pair ?m2 x*) **by** (*rule c-is-sub-fun-lm-1*)
from *T3 T-ls2* **have** *T-y*: *c-assoc-value ?ls* (*c-pair ?m2 x*) = *y* **by** *auto*
from *T-y S19 S20* **have** *S21*: *g-comp ?ls ?key* = *c-cons* (*c-pair ?key* (*c-assoc-value*

?ls (*c-pair ?m1 y*))) *?ls*
by(*unfold g-comp-def*)(*simp del*: *loc-upb.simps add*: *Let-def*)
from *S18 S21* **have** *pr-gr upb* = *c-cons* (*c-pair ?key* (*c-assoc-value ?ls* (*c-pair ?m1 y*))) *?ls* **by** *auto*
with *upb-def* **have** *pr-gr* (*loc-upb n x*) = *c-cons* (*c-pair ?key* (*c-assoc-value ?ls* (*c-pair ?m1 y*))) *?ls* **by** *auto*
thus *?thesis* **by** (*simp add*: *c-assoc-lm-1*)
qed

lemma *loc-upb-lex-5*: ⟦⋀ *n′ x′*. ((*n′*,*x′*), (*n*,*x*)) ∈ *lex-p* ⟹ *c-assoc-have-key* (*pr-gr* (*loc-upb n′ x′*)) (*c-pair n′ x′*) = *0*;
c-fst n mod 7 = *5*⟧ ⟹
c-assoc-have-key (*pr-gr* (*loc-upb n x*)) (*c-pair n x*) = *0*
proof −
assume *A1*: ⋀ *n′ x′*. ((*n′*,*x′*), (*n*,*x*)) ∈ *lex-p* ⟹ *c-assoc-have-key* (*pr-gr* (*loc-upb n′ x′*)) (*c-pair n′ x′*) = *0*
assume *A2*: *c-fst n mod 7* = *5*
let *?key* = *c-pair n x*
let *?m1* = *c-fst* (*c-snd n*)
let *?m2* = *c-snd* (*c-snd n*)
from *A2* **have** *?m1* < *n* **by** (*simp add*: *loc-upb-lm-2-3*)
then have ((*?m1*, *x*), (*n*,*x*)) ∈ *lex-p* **by** (*simp add*: *lex-p-eq*)
with *A1* **have** *S1*: *c-assoc-have-key* (*pr-gr* (*loc-upb ?m1 x*)) (*c-pair ?m1 x*) = *0* **by** *auto*
from *A2* **have** *?m2* < *n* **by** (*simp add*: *loc-upb-lm-2-4*)
then have ((*?m2*, *x*), (*n*,*x*)) ∈ *lex-p* **by** (*simp add*: *lex-p-eq*)
with *A1* **have** *S2*: *c-assoc-have-key* (*pr-gr* (*loc-upb ?m2 x*)) (*c-pair ?m2 x*) = *0* **by** *auto*
define *upb1* **where** *upb1* = *loc-upb ?m1 x*
define *upb2* **where** *upb2* = *loc-upb ?m2 x*
from *upb1-def S1* **have** *S3*: *c-assoc-have-key* (*pr-gr upb1*) (*c-pair ?m1 x*) = *0* **by** *auto*
from *upb2-def S2* **have** *S4*: *c-assoc-have-key* (*pr-gr upb2*) (*c-pair ?m2 x*) = *0* **by** *auto*
let *?sum-upb* = *upb1* +*upb2*
have *S5*: *upb1* ≤ *?sum-upb* **by** (*rule Nat.le-add1*)
have *S6*: *upb2* ≤ *?sum-upb* **by** (*rule Nat.le-add2*)
let *?s* = (*c-pair ?key ?sum-upb*)
have *S7*: *?sum-upb* ≤ *?s* **by** (*rule arg2-le-c-pair*)
from *S5 S7* **have** *S8*: *upb1* ≤ *?s* **by** *auto*
from *S6 S7* **have** *S9*: *upb2* ≤ *?s* **by** *auto*
let *?ls* = *pr-gr ?s*
from *A2 upb1-def upb2-def* **have** *S10*: *loc-upb n x* = *?s* + *1* **by** (*simp add*: *Let-def*)
define *upb* **where** *upb* = *loc-upb n x*
from *upb-def S10* **have** *S11*: *upb* = *?s* + *1* **by** *auto*
from *S11* **have** *S12*: *pr-gr upb* = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*simp add*: *pr-gr-at-Suc*)
from *S8 S10 upb-def* **have** *S13*: *upb1* ≤ *upb* **by** (*simp only*:)

from *S9 S10 upb-def* **have** *S14*: *upb2* ≤ *upb* **by** (*simp only*:)
from *S3 S13* **have** *S15*: *c-assoc-have-key* (*pr-gr upb*) (*c-pair ?m1 x*) = *0* **by** (*rule lm5*)
from *S4 S14* **have** *S16*: *c-assoc-have-key* (*pr-gr upb*) (*c-pair ?m2 x*) = *0* **by** (*rule lm5*)
from *A2* **have** *S17*: *g-step ?ls ?key* = *g-pair ?ls ?key* **by** (*simp add*: *g-step-def*)
from *S12 S17* **have** *S18*: *pr-gr upb* = *g-pair ?ls ?key* **by** *auto*
from *S3 S8* **have** *S19*: *c-assoc-have-key ?ls* (*c-pair ?m1 x*) = *0* **by** (*rule lm5*)
from *S4 S9* **have** *S20*: *c-assoc-have-key ?ls* (*c-pair ?m2 x*) = *0* **by** (*rule lm5*)
let *?y1* = *c-assoc-value ?ls* (*c-pair ?m1 x*)
let *?y2* = *c-assoc-value ?ls* (*c-pair ?m2 x*)
let *?y* = *c-pair ?y1 ?y2*
from *S19 S20* **have** *S21*: *g-pair ?ls ?key* = *c-cons* (*c-pair ?key ?y*) *?ls* **by** (*unfold g-pair-def*, *simp add*: *Let-def*)
from *S18 S21* **have** *S22*: *pr-gr upb* = *c-cons* (*c-pair ?key ?y*) *?ls* **by** *auto*
from *upb-def S22* **have** *S23*: *pr-gr* (*loc-upb n x*) = *c-cons* (*c-pair ?key ?y*) *?ls* **by** *auto*
from *S23* **show** *?thesis* **by** (*simp add*: *c-assoc-lm-1*)
qed

lemma *loc-upb-6-z*: ⟦*c-fst n mod 7* =*6*; *c-fst x* = *0*⟧ ⟹
loc-upb n x = *c-pair* (*c-pair n x*) (*loc-upb* (*c-fst* (*c-snd n*)) (*c-snd x*)) + *1* **by** (*simp add*: *Let-def*)

lemma *loc-upb-6*: ⟦*c-fst n mod 7* =*6*; *c-fst x* ≠ *0*⟧ ⟹ *loc-upb n x* = (
let m = *c-snd n*; *m1* = *c-fst m*; *m2* = *c-snd m*; *y1* = *c-fst x*; *x1* = *c-snd x*;
y2 = *y1* − *1*;
t1 = *c-assoc-value* (*pr-gr* (*loc-upb n* (*c-pair y2 x1*))) (*c-pair n* (*c-pair y2 x1*));
t2 = *c-pair* (*c-pair y2 t1*) *x1 in*
c-pair (*c-pair n x*) (*loc-upb n* (*c-pair y2 x1*) + (*loc-upb m2 t2*)) + *1*)
by (*simp add*: *Let-def*)

lemma *loc-upb-lex-6*: ⟦⋀ *n′ x′*. ((*n′*,*x′*), (*n*,*x*)) ∈ *lex-p* ⟹ *c-assoc-have-key* (*pr-gr* (*loc-upb n′ x′*)) (*c-pair n′ x′*) = *0*;
c-fst n mod 7 = *6*⟧ ⟹
c-assoc-have-key (*pr-gr* (*loc-upb n x*)) (*c-pair n x*) = *0*
proof −
assume *A1*: ⋀ *n′ x′*. ((*n′*,*x′*), (*n*,*x*)) ∈ *lex-p* ⟹ *c-assoc-have-key* (*pr-gr* (*loc-upb n′ x′*)) (*c-pair n′ x′*) = *0*
assume *A2*: *c-fst n mod 7* = *6*
let *?key* = *c-pair n x*
let *?m1* = *c-fst* (*c-snd n*)
let *?m2* = *c-snd* (*c-snd n*)
let *?y1* = *c-fst x*
let *?x1* = *c-snd x*
define *upb* **where** *upb* = *loc-upb n x*

```
 show ?thesis
 proof (cases)
  assume A: ?y1 = 0
  from A2 A have S1: loc-upb n x = c-pair ?key (loc-upb ?m1 (c-snd x)) + 1 by (rule loc-upb-6-z)
  define upb1 where upb1 = loc-upb ?m1 (c-snd x)
  from upb1-def S1 have S2: loc-upb n x = c-pair ?key upb1 + 1 by auto
  let ?s = c-pair ?key upb1
  from S2 have S3: pr-gr (loc-upb n x) = pr-gr (Suc ?s) by simp
  have pr-gr (Suc ?s) = g-step (pr-gr ?s) (c-fst ?s) by (rule pr-gr-at-Suc)
  with S3 have S4: pr-gr (loc-upb n x) = g-step (pr-gr ?s) ?key by auto
  let ?ls = pr-gr ?s
  from A2 have g-step ?ls ?key = g-rec ?ls ?key by (simp add: g-step-def)
  with S4 have S5: pr-gr (loc-upb n x) = g-rec ?ls ?key by auto
  have S6: c-assoc-have-key ?ls (c-pair ?m1 ?x1) = 0
  proof -
   from A2 have ?m1 < n by (simp add: loc-upb-lm-2-5)
   then have ((?m1,?x1), n, x) ∈ lex-p by (simp add: lex-p-eq)
   with A1 upb1-def have c-assoc-have-key (pr-gr upb1) (c-pair ?m1 ?x1) = 0 by auto
   also have upb1 ≤ ?s by (rule arg2-le-c-pair)
   ultimately show ?thesis by (rule lm5)
  qed
  from A S6 have g-rec ?ls ?key = c-cons (c-pair ?key (c-assoc-value ?ls (c-pair ?m1 ?x1))) ?ls by (simp add: g-rec-def Let-def)
  with S5 show ?thesis by (simp add: c-assoc-lm-1)
 next
  assume A: c-fst x ≠ 0 then have y1-pos: c-fst x > 0 by auto
  let ?y2 = ?y1 - 1
  from A2 A have loc-upb n x = (
                let m = c-snd n; m1 = c-fst m; m2 = c-snd m; y1 = c-fst x; x1 = c-snd x;
                 y2 = y1 - 1;
                t1 = c-assoc-value (pr-gr (loc-upb n (c-pair y2 x1))) (c-pair n (c-pair y2 x1));
                 t2 = c-pair (c-pair y2 t1) x1 in
                  c-pair (c-pair n x) (loc-upb n (c-pair y2 x1) + (loc-upb m2 t2)) + 1) by (rule loc-upb-6)
  then have S1: loc-upb n x = (
                 let
                  t1 = c-assoc-value (pr-gr (loc-upb n (c-pair ?y2 ?x1))) (c-pair n (c-pair ?y2 ?x1));
                 t2 = c-pair (c-pair ?y2 t1) ?x1 in
                 c-pair (c-pair n x) (loc-upb n (c-pair ?y2 ?x1) + (loc-upb ?m2 t2)) + 1) by (simp del: loc-upb.simps add: Let-def)
  let ?t1 = univ-for-pr (c-pair n (c-pair ?y2 ?x1))
  let ?t2 = c-pair (c-pair ?y2 ?t1) ?x1
  have S1-1: c-assoc-have-key (pr-gr (loc-upb n (c-pair ?y2 ?x1))) (c-pair n (c-pair ?y2 ?x1)) = 0
```

proof −
from *A* **have** *?y2* < *?y1* **by** *auto*
then have *c-pair ?y2 ?x1* < *c-pair ?y1 ?x1* **by** (*rule c-pair-strict-mono1*)
then have ((*n*, *c-pair ?y2 ?x1*),*n*,*x*) ∈ *lex-p* **by** (*simp add*: *lex-p-eq*)
with *A1* **show** *?thesis* **by** *auto*
qed
have *S2*: *c-assoc-value* (*pr-gr* (*loc-upb n* (*c-pair ?y2 ?x1*))) (*c-pair n* (*c-pair ?y2 ?x1*)) = *univ-for-pr* (*c-pair n* (*c-pair ?y2 ?x1*))
proof −
have *c-is-sub-fun* (*pr-gr* (*loc-upb n* (*c-pair ?y2 ?x1*))) *univ-for-pr* **by** (*rule pr-gr-1*)
with *S1-1* **show** *?thesis* **by** (*simp add*: *c-is-sub-fun-lm-1*)
qed
from *S1 S2* **have** *S3*: *loc-upb n x* = *c-pair* (*c-pair n x*) (*loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2*) + *1* **by** (*simp del*: *loc-upb.simps add*: *Let-def*)
let *?s* = *c-pair* (*c-pair n x*) (*loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2*)
from *S3* **have** *S4*: *pr-gr* (*loc-upb n x*) = *pr-gr* (*Suc ?s*) **by** (*simp del*: *loc-upb.simps*)
have *pr-gr* (*Suc ?s*) = *g-step* (*pr-gr ?s*) (*c-fst ?s*) **by** (*rule pr-gr-at-Suc*)
with *S4* **have** *S5*: *pr-gr* (*loc-upb n x*) = *g-step* (*pr-gr ?s*) *?key* **by** (*simp del*: *loc-upb.simps*)
let *?ls* = *pr-gr ?s*
from *A2* **have** *g-step ?ls ?key* = *g-rec ?ls ?key* **by** (*simp add*: *g-step-def*)
with *S5* **have** *S6*: *pr-gr* (*loc-upb n x*) = *g-rec ?ls ?key* **by** (*simp del*: *loc-upb.simps*)
have *S7*: *c-assoc-have-key ?ls* (*c-pair n* (*c-pair ?y2 ?x1*)) = *0*
proof −
have *loc-upb n* (*c-pair ?y2 ?x1*) ≤ *loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2* **by** (*auto simp del*: *loc-upb.simps*)
also have *loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2* ≤ *?s* **by** (*rule arg2-le-c-pair*)
ultimately have *S7-1*: *loc-upb n* (*c-pair ?y2 ?x1*) ≤ *?s* **by** (*auto simp del*: *loc-upb.simps*)
from *S1-1 S7-1* **show** *?thesis* **by** (*rule lm5*)
qed
have *S8*: *c-assoc-value ?ls* (*c-pair n* (*c-pair ?y2 ?x1*)) = *?t1*
proof −
have *c-is-sub-fun ?ls univ-for-pr* **by** (*rule pr-gr-1*)
with *S7* **show** *?thesis* **by** (*simp add*: *c-is-sub-fun-lm-1*)
qed
have *S9*: *c-assoc-have-key ?ls* (*c-pair ?m2 ?t2*) = *0*
proof −
from *A2* **have** *?m2* < *n* **by** (*simp add*: *loc-upb-lm-2-6*)
then have ((*?m2*,*?t2*), *n*, *x*) ∈ *lex-p* **by** (*simp add*: *lex-p-eq*)
with *A1* **have** *c-assoc-have-key* (*pr-gr* (*loc-upb ?m2 ?t2*)) (*c-pair ?m2 ?t2*) = *0* **by** *auto*
also have *loc-upb ?m2 ?t2* ≤ *?s*
proof −
have *loc-upb ?m2 ?t2* ≤ *loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2* **by** (*auto simp del*: *loc-upb.simps*)
also have *loc-upb n* (*c-pair ?y2 ?x1*) + *loc-upb ?m2 ?t2* ≤ *?s* **by** (*rule*

```
arg2-le-c-pair)
      ultimately show ?thesis by (auto simp del: loc-upb.simps)
     qed
     ultimately show ?thesis by (rule lm5)
   qed
    from A S7 S8 S9 have g-rec ?ls ?key = c-cons (c-pair ?key (c-assoc-value ?ls
(c-pair ?m2 ?t2))) ?ls by (simp del: loc-upb.simps add: g-rec-def Let-def)
    with S6 show ?thesis by (simp add: c-assoc-lm-1)
  qed
qed

lemma wf-upb-step-0:
  ⟦⋀ n′ x′. ((n′,x′), (n,x)) ∈ lex-p ⟹ c-assoc-have-key (pr-gr (loc-upb n′ x′))
(c-pair n′ x′) = 0⟧ ⟹
      c-assoc-have-key (pr-gr (loc-upb n x)) (c-pair n x) = 0
proof –
 assume A1: ⋀ n′ x′. ((n′,x′), (n,x)) ∈ lex-p ⟹ c-assoc-have-key (pr-gr (loc-upb
n′ x′)) (c-pair n′ x′) = 0
  let ?n1 = (c-fst n) mod 7
  have S1: ?n1 = 0 ⟹ ?thesis
  proof –
    assume A: ?n1 = 0
    thus ?thesis by (rule loc-upb-lex-0)
  qed
  have S2: ?n1 = 1 ⟹ ?thesis
  proof –
    assume A: ?n1 = 1
    thus ?thesis by (rule loc-upb-lex-1)
  qed
  have S3: ?n1 = 2 ⟹ ?thesis
  proof –
    assume A: ?n1 = 2
    thus ?thesis by (rule loc-upb-lex-2)
  qed
  have S4: ?n1 = 3 ⟹ ?thesis
  proof –
    assume A: ?n1 = 3
    thus ?thesis by (rule loc-upb-lex-3)
  qed
  have S5: ?n1 = 4 ⟹ ?thesis
  proof –
    assume A: ?n1 = 4
    from A1 A show ?thesis by (rule loc-upb-lex-4)
  qed
  have S6: ?n1 = 5 ⟹ ?thesis
  proof –
    assume A: ?n1 = 5
    from A1 A show ?thesis by (rule loc-upb-lex-5)
  qed
```

```
  have S7: ?n1 = 6 ⟹ ?thesis
  proof −
    assume A: ?n1 = 6
    from A1 A show ?thesis by (rule loc-upb-lex-6)
  qed
  have S8: ?n1=0 ∨ ?n1=1 ∨ ?n1=2 ∨ ?n1=3 ∨ ?n1=4 ∨ ?n1=5 ∨ ?n1=6
by (rule mod7-lm)
  from S1 S2 S3 S4 S5 S6 S7 S8 show ?thesis by fast
qed

lemma wf-upb-step:
  assumes A1: ⋀ p2. (p2, p1) ∈ lex-p ⟹
    c-assoc-have-key (pr-gr (loc-upb (fst p2) (snd p2))) (c-pair (fst p2) (snd p2))
= 0
  shows c-assoc-have-key (pr-gr (loc-upb (fst p1) (snd p1))) (c-pair (fst p1) (snd
p1)) = 0
proof −
  let ?n = fst p1
  let ?x = snd p1
  from A1 have S1: ⋀ p2. (p2, (?n, ?x)) ∈ lex-p ⟹
    c-assoc-have-key (pr-gr (loc-upb (fst p2) (snd p2))) (c-pair (fst p2) (snd p2))
= 0
    by auto
  have S2: (⋀ n′ x′. ((n′,x′), (fst p1, snd p1)) ∈ lex-p
    ⟹ c-assoc-have-key (pr-gr (loc-upb n′ x′)) (c-pair n′ x′) = 0) ⟹
      c-assoc-have-key (pr-gr (loc-upb (fst p1) (snd p1))) (c-pair (fst p1) (snd p1))
= 0
    by (rule wf-upb-step-0)
  then have S3: (⋀ n′ x′. ((n′,x′), p1) ∈ lex-p ⟹ c-assoc-have-key (pr-gr (loc-upb
n′ x′)) (c-pair n′ x′) = 0)
    ⟹ c-assoc-have-key (pr-gr (loc-upb (fst p1) (snd p1))) (c-pair (fst p1) (snd
p1)) = 0 by auto
  have S4: ⋀n′ x′. ((n′, x′), p1) ∈ lex-p ⟹ c-assoc-have-key (pr-gr (loc-upb n′
x′)) (c-pair n′ x′) = 0
  proof −
    fix n′ x′
    assume A4-1: ((n′, x′), p1) ∈ lex-p
    let ?p2 = (n′, x′)
    from A4-1 have S4-1: (?p2, p1) ∈ lex-p by auto
    from S4-1 have c-assoc-have-key (pr-gr (loc-upb (fst ?p2) (snd ?p2))) (c-pair
(fst ?p2) (snd ?p2)) = 0
      by (rule A1)
    then show c-assoc-have-key (pr-gr (loc-upb n′ x′)) (c-pair n′ x′) = 0 by auto
  qed
  from S4 S3 show ?thesis by auto
qed

theorem loc-upb-main: c-assoc-have-key (pr-gr (loc-upb n x)) (c-pair n x) = 0
proof −
```

```
  have loc-upb-lm: ⋀ p. c-assoc-have-key (pr-gr (loc-upb (fst p) (snd p))) (c-pair
(fst p) (snd p)) = 0
   proof − fix p show c-assoc-have-key (pr-gr (loc-upb (fst p) (snd p))) (c-pair
(fst p) (snd p)) = 0
    proof −
      have S1: wf lex-p by (auto simp add: lex-p-def)
      from S1 wf-upb-step show ?thesis by (rule wf-induct-rule)
    qed
  qed
  let ?p = (n,x)
   have c-assoc-have-key (pr-gr (loc-upb (fst ?p) (snd ?p))) (c-pair (fst ?p) (snd
?p)) = 0 by (rule loc-upb-lm)
  thus ?thesis by simp
qed

theorem pr-gr-value: c-assoc-value (pr-gr (loc-upb n x)) (c-pair n x) = univ-for-pr
(c-pair n x)
  by (simp del: loc-upb.simps add: loc-upb-main pr-gr-1 c-is-sub-fun-lm-1)

theorem g-comp-is-pr: g-comp ∈ PrimRec2
proof −
  from c-assoc-have-key-is-pr c-assoc-value-is-pr c-cons-is-pr have (λ x y. g-comp
x y) ∈ PrimRec2
    unfolding g-comp-def Let-def by prec
  thus ?thesis by auto
qed

theorem g-pair-is-pr: g-pair ∈ PrimRec2
proof −
  from c-assoc-have-key-is-pr c-assoc-value-is-pr c-cons-is-pr have (λ x y. g-pair
x y) ∈ PrimRec2
    unfolding g-pair-def Let-def by prec
  thus ?thesis by auto
qed

theorem g-rec-is-pr: g-rec ∈ PrimRec2
proof −
  from c-assoc-have-key-is-pr c-assoc-value-is-pr c-cons-is-pr have (λ x y. g-rec x
y) ∈ PrimRec2
    unfolding g-rec-def Let-def by prec
  thus ?thesis by auto
qed

theorem g-step-is-pr: g-step ∈ PrimRec2
proof −
 from g-comp-is-pr g-pair-is-pr g-rec-is-pr mod-is-pr c-assoc-have-key-is-pr c-assoc-value-is-pr
c-cons-is-pr have
    (λ ls key. g-step ls key) ∈ PrimRec2 unfolding g-step-def Let-def by prec
  thus ?thesis by auto
```

```
qed

theorem pr-gr-is-pr: pr-gr ∈ PrimRec1
proof −
  have S1: (λ x. pr-gr x) = PrimRecOp1 0 (λ x y. g-step y (c-fst x)) (is - = ?f)
  proof
    fix x
   show pr-gr x = ?f x by (induct x) (simp add: pr-gr-at-0, simp add: pr-gr-at-Suc)
  qed
  have S2: PrimRecOp1 0 (λ x y. g-step y (c-fst x)) ∈ PrimRec1
  proof (rule pr-rec1)
    from g-step-is-pr show (λx y. g-step y (c-fst x)) ∈ PrimRec2 by prec
  qed
  from S1 S2 show ?thesis by auto
qed

end
```

7 Computably enumerable sets of natural numbers

```
theory RecEnSet
imports PRecList PRecFun2 PRecFinSet PRecUnGr
begin
```

7.1 Basic definitions

```
definition
  fn-to-set :: (nat ⇒ nat ⇒ nat) ⇒ nat set where
  fn-to-set f = { x. ∃ y. f x y = 0 }

definition
  ce-sets :: (nat set) set where
  ce-sets = { (fn-to-set p) | p. p ∈ PrimRec2 }
```

7.2 Basic properties of computably enumerable sets

```
lemma ce-set-lm-1: p ∈ PrimRec2 ⟹ fn-to-set p ∈ ce-sets by (auto simp add:
ce-sets-def)

lemma ce-set-lm-2: ⟦ p ∈ PrimRec2; ∀ x. (x ∈ A) = (∃ y. p x y = 0)⟧ ⟹ A ∈
ce-sets
proof −
  assume p-is-pr: p ∈ PrimRec2
  assume ∀ x. (x ∈ A) = (∃ y. p x y = 0)
  then have A = fn-to-set p by (unfold fn-to-set-def, auto)
  with p-is-pr show A ∈ ce-sets by (simp add: ce-set-lm-1)
qed
```

```
lemma ce-set-lm-3: A ∈ ce-sets ⟹ ∃ p ∈ PrimRec2. A = fn-to-set p
proof −
  assume A ∈ ce-sets
  then have A ∈ { (fn-to-set p) | p. p ∈ PrimRec2 } by (simp add: ce-sets-def)
  thus ?thesis by auto
qed

lemma ce-set-lm-4: A ∈ ce-sets ⟹ ∃ p ∈ PrimRec2. ∀ x. (x ∈ A) = (∃ y. p x
y = 0)
proof −
  assume A ∈ ce-sets
  then have ∃ p ∈ PrimRec2. A = fn-to-set p by (rule ce-set-lm-3)
  then obtain p where p-is-pr: p ∈ PrimRec2 and L1: A = fn-to-set p ..
  from p-is-pr L1 show ?thesis by (unfold fn-to-set-def, auto)
qed

lemma ce-set-lm-5: ⟦ A ∈ ce-sets; p ∈ PrimRec1 ⟧ ⟹ { x . p x ∈ A } ∈ ce-sets
proof −
  assume A1: A ∈ ce-sets
  assume A2: p ∈ PrimRec1
  from A1 have ∃ pA ∈ PrimRec2. A = fn-to-set pA by (rule ce-set-lm-3)
  then obtain pA where pA-is-pr: pA ∈ PrimRec2 and S1: A = fn-to-set pA ..
  from S1 have S2: A = { x . ∃ y. pA x y = 0 } by (simp add: fn-to-set-def)
  define q where q x y = pA (p x) y for x y
  from pA-is-pr A2 have q-is-pr: q ∈ PrimRec2 unfolding q-def by prec
  have ⋀ x. (p x ∈ A) = (∃ y. q x y = 0)
  proof −
    fix x show (p x ∈ A) = (∃ y. q x y = 0)
    proof
      assume A: p x ∈ A
      with S2 obtain y where L1: pA (p x) y = 0 by auto
      then have q x y = 0 by (simp add: q-def)
      thus ∃ y. q x y = 0 ..
    next
      assume A: ∃y. q x y = 0
      then obtain y where L1: q x y = 0 ..
      then have pA (p x) y = 0 by (simp add: q-def)
      with S2 show p x ∈ A by auto
    qed
  qed
  then have { x . p x ∈ A } = { x. ∃ y. q x y = 0 } by auto
  then have { x . p x ∈ A } = fn-to-set q by (simp add: fn-to-set-def)
  moreover from q-is-pr have fn-to-set q ∈ ce-sets by (rule ce-set-lm-1)
  ultimately show ?thesis by auto
qed

lemma ce-set-lm-6: ⟦ A ∈ ce-sets; A ≠ {}⟧ ⟹ ∃ q ∈ PrimRec1. A = { q x | x.
x ∈ UNIV }
proof −
```

```
  assume A1: A ∈ ce-sets
  assume A2: A ≠ {}
  from A1 have ∃ pA ∈ PrimRec2. A = fn-to-set pA by (rule ce-set-lm-3)
  then obtain pA where pA-is-pr: pA ∈ PrimRec2 and S1: A = fn-to-set pA ..
  from S1 have S2: A = { x. ∃ y. pA x y = 0 } by (simp add: fn-to-set-def)
  from A2 obtain a where a-in: a ∈ A by auto
  define q where q z = (if pA (c-fst z) (c-snd z) = 0 then c-fst z else a) for z
  from pA-is-pr have q-is-pr: q ∈ PrimRec1 unfolding q-def by prec
  have S3: ∀ z. q z ∈ A
  proof
    fix z show q z ∈ A
    proof cases
      assume A: pA (c-fst z) (c-snd z) = 0
      with S2 have c-fst z ∈ A by auto
      moreover from A q-def have q z = c-fst z by simp
      ultimately show q z ∈ A by auto
    next
      assume A: pA (c-fst z) (c-snd z) ≠ 0
      with q-def have q z = a by simp
      with a-in show q z ∈ A by auto
    qed
  qed
  then have S4: { q x | x. x ∈ UNIV } ⊆ A by auto
  have S5: A ⊆ { q x | x. x ∈ UNIV }
  proof
    fix x assume A: x ∈ A show x ∈ {q x |x. x ∈ UNIV}
    proof
      from A S2 obtain y where L1: pA x y = 0 by auto
      let ?z = c-pair x y
      from L1 have q ?z = x by (simp add: q-def)
      then have ∃ u. q u = x by blast
      then show ∃u. x = q u ∧ u ∈ UNIV by auto
    qed
  qed
  from S4 S5 have S6: A = { q x | x. x ∈ UNIV } by auto
  with q-is-pr show ?thesis by blast
qed

lemma ce-set-lm-7: ⟦ A ∈ ce-sets; p ∈ PrimRec1⟧ ⟹ { p x | x. x ∈ A } ∈ ce-sets
proof −
  assume A1: A ∈ ce-sets
  assume A2: p ∈ PrimRec1
  let ?B = { p x | x. x ∈ A }
  fix y have S1: (y ∈ ?B) = (∃ x. x ∈ A ∧ (y = p x)) by auto
  from A1 have ∃ pA ∈ PrimRec2. A = fn-to-set pA by (rule ce-set-lm-3)
  then obtain pA where pA-is-pr: pA ∈ PrimRec2 and S2: A = fn-to-set pA ..
  from S2 have S3: A = { x. ∃ y. pA x y = 0 } by (simp add: fn-to-set-def)
  define q where q y t = (if y = p (c-snd t) then pA (c-snd t) (c-fst t) else 1)
for y t
```

```
  from pA-is-pr A2 have q-is-pr: q ∈ PrimRec2 unfolding q-def by prec
  have L1: ⋀ y. (y ∈ ?B) = (∃ z. q y z = 0)
  proof − fix y show (y ∈ ?B) = (∃ z. q y z = 0)
  proof
    assume AA1: y ∈ ?B
    then obtain x0 where LL-2: x0 ∈ A and LL-3: y = p x0 by auto
    from S3 have LL-4: (x0 ∈ A) = (∃ z. pA x0 z = 0) by auto
    from LL-2 LL-4 obtain z0 where LL-5: pA x0 z0 = 0 by auto
    define t where t = c-pair z0 x0
    from t-def q-def LL-3 LL-5 have q y t = 0 by simp
    then show ∃ z. q y z = 0 by auto
  next
    assume A1: ∃ z. q y z = 0
    then obtain z0 where LL-1: q y z0 = 0 ..
    have LL2: y = p (c-snd z0)
    proof (rule ccontr)
      assume y ≠ p (c-snd z0)
      with q-def LL-1 have q y z0 = 1 by auto
      with LL-1 show False by auto
    qed
    from LL2 LL-1 q-def have LL3: pA (c-snd z0) (c-fst z0) = 0 by auto
    with S3 have LL4: c-snd z0 ∈ A by auto
    with LL2 show y ∈ {p x | x. x ∈ A} by auto
  qed
  qed
  then have L2: ?B = { y | y. ∃ z. q y z = 0} by auto
  with fn-to-set-def have ?B = fn-to-set q by auto
  with q-is-pr ce-set-lm-1 show ?thesis by auto
qed

theorem ce-empty: {} ∈ ce-sets
proof −
  let ?f = (λ x a. (1::nat))
  have S1: ?f ∈ PrimRec2 by (rule const-is-pr-2)
  then have ∀ x a. ?f x a ≠ 0 by simp
  then have {x. ∃ a. ?f x a = 0 }={} by auto
  also have fn-to-set ?f = . . . by (simp add: fn-to-set-def)
  with S1 show ?thesis by (auto simp add: ce-sets-def)
qed

theorem ce-univ: UNIV ∈ ce-sets
proof −
  let ?f = (λ x a. (0::nat))
  have S1: ?f ∈ PrimRec2 by (rule const-is-pr-2)
  then have ∀ x a. ?f x a = 0 by simp
  then have {x. ∃ a. ?f x a = 0 }=UNIV by auto
  also have fn-to-set ?f = . . . by (simp add: fn-to-set-def)
  with S1 show ?thesis by (auto simp add: ce-sets-def)
qed
```

```
theorem ce-singleton: {a} ∈ ce-sets
proof −
  let ?f = λ x y. (abs-of-diff x a) + y
  have S1: ?f ∈ PrimRec2 using const-is-pr-2 [where ?n=a] by prec
  then have ∀ x y. (?f x y = 0) = (x=a ∧ y=0) by (simp add: abs-of-diff-eq)
  then have S2: {x. ∃ y. ?f x y = 0 }={a} by auto
  have fn-to-set ?f = {x. ∃ y. ?f x y = 0 } by (simp add: fn-to-set-def)
  with S2 have fn-to-set ?f = {a} by simp
  with S1 show ?thesis by (auto simp add: ce-sets-def)
qed

theorem ce-union: ⟦ A ∈ ce-sets; B ∈ ce-sets ⟧ ⟹ A ∪ B ∈ ce-sets
proof −
  assume A1: A ∈ ce-sets
  then obtain p-a where S2: p-a ∈ PrimRec2 and S3: A = fn-to-set p-a
    by (auto simp add: ce-sets-def)
  assume A2: B ∈ ce-sets
  then obtain p-b where S5: p-b ∈ PrimRec2 and S6: B = fn-to-set p-b
    by (auto simp add: ce-sets-def)
  let ?p = (λ x y. (p-a x y) ∗ (p-b x y))
  from S2 S5 have S7: ?p ∈ PrimRec2 by prec
  have S8: ∀ x y. (?p x y = 0) = ((p-a x y = 0) ∨ (p-b x y = 0)) by simp
  let ?C = fn-to-set ?p
  have S9: ?C = {x. ∃ y. ?p x y = 0} by (simp add: fn-to-set-def)
  from S3 have S10: A = {x. ∃ y. p-a x y = 0} by (simp add: fn-to-set-def)
  from S6 have S11: B = {x. ∃ y. p-b x y = 0} by (simp add: fn-to-set-def)
  from S10 S11 S9 S8 have S12: ?C = A ∪ B by auto
  from S7 have ?C ∈ ce-sets by (auto simp add: ce-sets-def)
  with S12 show ?thesis by simp
qed

theorem ce-intersect: ⟦ A ∈ ce-sets; B ∈ ce-sets ⟧ ⟹ A ∩ B ∈ ce-sets
proof −
  assume A1: A ∈ ce-sets
  then obtain p-a where S2: p-a ∈ PrimRec2 and S3: A = fn-to-set p-a
    by (auto simp add: ce-sets-def)
  assume A2: B ∈ ce-sets
  then obtain p-b where S5: p-b ∈ PrimRec2 and S6: B = fn-to-set p-b
    by (auto simp add: ce-sets-def)
  let ?p = (λ x y. (p-a x (c-fst y)) + (p-b x (c-snd y)))
  from S2 S5 have S7: ?p ∈ PrimRec2 by prec
  have S8: ∀ x. (∃ y. ?p x y = 0) = ((∃ z. p-a x z = 0) ∧ (∃ z. p-b x z = 0))
  proof
    fix x show (∃ y. ?p x y = 0) = ((∃ z. p-a x z = 0) ∧ (∃ z. p-b x z = 0))
    proof −
    have 1: (∃ y. ?p x y = 0) ⟹ ((∃ z. p-a x z = 0) ∧ (∃ z. p-b x z = 0))
    by blast
    have 2: ((∃ z. p-a x z = 0) ∧ (∃ z. p-b x z = 0)) ⟹ (∃ y. ?p x y = 0)
```

proof −
assume ((∃ *z*. *p-a x z* = *0*) ∧ (∃ *z*. *p-b x z* = *0*))
then obtain *z1 z2* **where** *s-23*: *p-a x z1* = *0* **and** *s-24*: *p-b x z2* = *0* **by** *auto*
let *?y1* = *c-pair z1 z2*
from *s-23* **have** *s-25*: *p-a x* (*c-fst ?y1*) = *0* **by** *simp*
from *s-24* **have** *s-26*: *p-b x* (*c-snd ?y1*) = *0* **by** *simp*
from *s-25 s-26* **have** *s-27*: *p-a x* (*c-fst ?y1*) + *p-b x* (*c-snd ?y1*) = *0* **by** *simp*
then show *?thesis* **..**
qed
from *1 2* **have** (∃ *y*. *?p x y* = *0*) = ((∃ *z*. *p-a x z* = *0*) ∧ (∃ *z*. *p-b x z* = *0*)) **by** (*rule iffI*)
then show *?thesis* **by** *auto*
qed
qed
let *?C* = *fn-to-set ?p*
have *S9*: *?C* = {*x*. ∃ *y*. *?p x y* = *0*} **by** (*simp add*: *fn-to-set-def*)
from *S3* **have** *S10*: *A* = {*x*. ∃ *y*. *p-a x y* = *0*} **by** (*simp add*: *fn-to-set-def*)
from *S6* **have** *S11*: *B* = {*x*. ∃ *y*. *p-b x y* = *0*} **by** (*simp add*: *fn-to-set-def*)
from *S10 S11 S9 S8* **have** *S12*: *?C* = *A* ∩ *B* **by** *auto*
from *S7* **have** *?C* ∈ *ce-sets* **by** (*auto simp add*: *ce-sets-def*)
with *S12* **show** *?thesis* **by** *simp*
qed

7.3 Enumeration of computably enumerable sets

definition
nat-to-ce-set :: *nat* ⇒ (*nat set*) **where**
nat-to-ce-set = (λ *n*. *fn-to-set* (*pr-conv-1-to-2* (*nat-to-pr n*)))

lemma *nat-to-ce-set-lm-1*: *nat-to-ce-set n* = { *x* . ∃ *y*. (*nat-to-pr n*) (*c-pair x y*) = *0* }
proof −
have *S1*: *nat-to-ce-set n* = *fn-to-set* (*pr-conv-1-to-2* (*nat-to-pr n*)) **by** (*simp add*: *nat-to-ce-set-def*)
then have *S2*: *nat-to-ce-set n* = { *x* . ∃ *y*. (*pr-conv-1-to-2* (*nat-to-pr n*)) *x y* = *0*} **by** (*simp add*: *fn-to-set-def*)
have *S3*: ⋀ *x y*. (*pr-conv-1-to-2* (*nat-to-pr n*)) *x y* = (*nat-to-pr n*) (*c-pair x y*) **by** (*simp add*: *pr-conv-1-to-2-def*)
from *S2 S3* **show** *?thesis* **by** *auto*
qed

lemma *nat-to-ce-set-into-ce*: *nat-to-ce-set n* ∈ *ce-sets*
proof −
have *S1*: *nat-to-ce-set n* = *fn-to-set* (*pr-conv-1-to-2* (*nat-to-pr n*)) **by** (*simp add*: *nat-to-ce-set-def*)
have (*nat-to-pr n*) ∈ *PrimRec1* **by** (*rule nat-to-pr-into-pr*)
then have *S2*: (*pr-conv-1-to-2* (*nat-to-pr n*)) ∈ *PrimRec2* **by** (*rule pr-conv-1-to-2-lm*)
from *S2 S1* **show** *?thesis* **by** (*simp add*: *ce-set-lm-1*)

qed

lemma *nat-to-ce-set-srj*: *A* ∈ *ce-sets* ⟹ ∃ *n*. *A* = *nat-to-ce-set n*
proof −
assume *A*: *A* ∈ *ce-sets*
then have ∃ *p* ∈ *PrimRec2*. *A* = *fn-to-set p* **by** (*rule ce-set-lm-3*)
then obtain *p* **where** *p-is-pr*: *p* ∈ *PrimRec2* **and** *S1*: *A* = *fn-to-set p* **..**
define *q* **where** *q* = *pr-conv-2-to-1 p*
from *p-is-pr* **have** *q-is-pr*: *q* ∈ *PrimRec1* **by** (*unfold q-def*, *rule pr-conv-2-to-1-lm*)
from *q-def* **have** *S2*: *pr-conv-1-to-2 q* = *p* **by** *simp*
let *?n* = *index-of-pr q*
from *q-is-pr* **have** *nat-to-pr ?n* = *q* **by** (*rule index-of-pr-is-real*)
with *S2 S1* **have** *A* = *fn-to-set* (*pr-conv-1-to-2* (*nat-to-pr ?n*)) **by** *auto*
then have *A* = *nat-to-ce-set ?n* **by** (*simp add*: *nat-to-ce-set-def*)
thus *?thesis* **..**
qed

7.4 Characteristic functions

definition
chf :: *nat set* ⇒ (*nat* ⇒ *nat*) — Characteristic function **where**
chf = (λ *A x*. *if x* ∈ *A then 0 else 1*)

definition
zero-set :: (*nat* ⇒ *nat*) ⇒ *nat set* **where**
zero-set = (λ *f*. { *x*. *f x* = *0*})

lemma *chf-lm-1* [*simp*]: *zero-set* (*chf A*) = *A* **by** (*unfold chf-def*, *unfold zero-set-def*, *simp*)

lemma *chf-lm-2*: (*x* ∈ *A*) = (*chf A x* = *0*) **by** (*unfold chf-def*, *simp*)

lemma *chf-lm-3*: (*x* ∉ *A*) = (*chf A x* = *1*) **by** (*unfold chf-def*, *simp*)

lemma *chf-lm-4*: *chf A* ∈ *PrimRec1* ⟹ *A* ∈ *ce-sets*
proof −
assume *A*: *chf A* ∈ *PrimRec1*
define *p* **where** *p* = *chf A*
from *A p-def* **have** *p-is-pr*: *p* ∈ *PrimRec1* **by** *auto*
define *q* **where** *q x y* = *p x* **for** *x y* :: *nat*
from *p-is-pr* **have** *q-is-pr*: *q* ∈ *PrimRec2* **unfolding** *q-def* **by** *prec*
have *S1*: *A* = {*x*. *p*(*x*) = *0*}
proof −
have *zero-set p* = *A* **by** (*unfold p-def*, *simp*)
thus *?thesis* **by** (*simp add*: *zero-set-def*)
qed
have *S2*: *fn-to-set q* = {*x*. ∃ *y*. *q x y* = *0*} **by** (*simp add*: *fn-to-set-def*)
have *S3*: ⋀ *x*. (*p x* = *0*) = (∃ *y*. *q x y* = *0*) **by** (*unfold q-def*, *auto*)
then have *S4*: {*x*. *p x* = *0*} = {*x*. ∃ *y*. *q x y* = *0*} **by** *auto*

```
  with S1 S2 have S5: fn-to-set q = A by auto
  from q-is-pr have fn-to-set q ∈ ce-sets by (rule ce-set-lm-1)
  with S5 show ?thesis by auto
qed

lemma chf-lm-5: finite A ⟹ chf A ∈ PrimRec1
proof −
  assume A: finite A
  define u where u = set-to-nat A
  from A have S1: nat-to-set u = A by (unfold u-def, rule nat-to-set-srj)
  have chf A = (λ x. sgn2 (c-in x u))
  proof
    fix x show chf A x = sgn2 (c-in x u)
    proof cases
      assume A: x ∈ A
      then have S1-1: chf A x = 0 by (simp add: chf-lm-2)
      from A S1 have x ∈ nat-to-set u by auto
      then have c-in x u = 1 by (simp add: x-in-u-eq)
      with S1-1 show ?thesis by simp
    next
      assume A: x ∉ A
      then have S1-1: chf A x = 1 by (simp add: chf-def)
      from A S1 have x ∉ nat-to-set u by auto
      then have c-in x u = 0 by (simp add: x-in-u-eq c-in-def)
      with S1-1 show ?thesis by simp
    qed
  qed
  moreover from c-in-is-pr have (λ x. sgn2 (c-in x u)) ∈ PrimRec1 by prec
  ultimately show ?thesis by auto
qed

theorem ce-finite: finite A ⟹ A ∈ ce-sets
proof −
  assume A: finite A
  then have chf A ∈ PrimRec1 by (rule chf-lm-5)
  then show ?thesis by (rule chf-lm-4)
qed
```

7.5 Computably enumerable relations

```
definition
  ce-set-to-rel :: nat set ⇒ (nat ∗ nat) set where
  ce-set-to-rel = (λ A. { (c-fst x, c-snd x) | x. x ∈ A})

definition
  ce-rel-to-set :: (nat ∗ nat) set ⇒ nat set where
  ce-rel-to-set = (λ R. { c-pair x y | x y. (x,y) ∈ R})

definition
```

```
  ce-rels :: ((nat ∗ nat) set) set where
  ce-rels = { R | R. ce-rel-to-set R ∈ ce-sets }

lemma ce-rel-lm-1 [simp]: ce-set-to-rel (ce-rel-to-set r) = r
proof
  show  ce-set-to-rel (ce-rel-to-set r) ⊆ r
  proof fix z
    assume A: z ∈ ce-set-to-rel (ce-rel-to-set r)
    then obtain u where L1: u ∈ (ce-rel-to-set r) and L2: z = (c-fst u, c-snd u)
      unfolding ce-set-to-rel-def by auto
    from L1 obtain x y where L3: (x,y) ∈ r and L4: u = c-pair x y
      unfolding ce-rel-to-set-def by auto
    from L4 have L5: c-fst u = x by simp
    from L4 have L6: c-snd u = y by simp
    from L5 L6 L2 have z = (x,y) by simp
    with L3 show z ∈ r by auto
  qed
next
  show r ⊆ ce-set-to-rel (ce-rel-to-set r)
  proof fix z show z ∈ r ⟹ z ∈ ce-set-to-rel (ce-rel-to-set r)
    proof −
      assume A: z ∈ r
      define x where x = fst z
      define y where y = snd z
      from x-def y-def have L1: z = (x,y) by simp
      define u where u = c-pair x y
       from A L1 u-def have L2: u ∈ ce-rel-to-set r by (unfold ce-rel-to-set-def,
auto)
      from L1 u-def have L3: z = (c-fst u, c-snd u) by simp
     from L2 L3 show z ∈ ce-set-to-rel (ce-rel-to-set r) by (unfold ce-set-to-rel-def,
auto)
    qed
  qed
qed

lemma ce-rel-lm-2 [simp]: ce-rel-to-set (ce-set-to-rel A) = A
proof
  show ce-rel-to-set (ce-set-to-rel A) ⊆ A
  proof fix z show z ∈ ce-rel-to-set (ce-set-to-rel A) ⟹ z ∈ A
    proof −
      assume A: z ∈ ce-rel-to-set (ce-set-to-rel A)
      then obtain x y where L1: z = c-pair x y and L2: (x,y) ∈ ce-set-to-rel A
        unfolding ce-rel-to-set-def by auto
      from L2 obtain u where L3: (x,y) = (c-fst u, c-snd u) and L4: u ∈ A
        unfolding ce-set-to-rel-def by auto
      from L3 L1 have L5: z = u by simp
      with L4 show z ∈ A by auto
    qed
  qed
```

```
next
  show A ⊆ ce-rel-to-set (ce-set-to-rel A)
  proof fix z show z ∈ A ⟹ z ∈ ce-rel-to-set (ce-set-to-rel A)
    proof −
      assume A: z ∈ A
     then have L1: (c-fst z, c-snd z) ∈ ce-set-to-rel A by (unfold ce-set-to-rel-def,
auto)
      define x where x = c-fst z
      define y where y = c-snd z
      from L1 x-def y-def have L2: (x,y) ∈ ce-set-to-rel A by simp
    then have L3: c-pair x y ∈ ce-rel-to-set (ce-set-to-rel A) by (unfold ce-rel-to-set-def,
auto)
      with x-def y-def show z ∈ ce-rel-to-set (ce-set-to-rel A) by simp
    qed
  qed
qed

lemma ce-rels-def1: ce-rels = { ce-set-to-rel A | A. A ∈ ce-sets}
proof
  show ce-rels ⊆ {ce-set-to-rel A |A. A ∈ ce-sets}
  proof fix r show  r ∈ ce-rels ⟹ r ∈ {ce-set-to-rel A |A. A ∈ ce-sets}
    proof −
      assume A: r ∈ ce-rels
      then have L1: ce-rel-to-set r ∈ ce-sets by (unfold ce-rels-def, auto)
      define A where A = ce-rel-to-set r
      from A-def L1 have L2: A ∈ ce-sets by auto
      from A-def have L3: ce-set-to-rel A = r by simp
      with L2 show r ∈ {ce-set-to-rel A |A. A ∈ ce-sets} by auto
    qed
  qed
next
  show {ce-set-to-rel A |A. A ∈ ce-sets} ⊆ ce-rels
  proof fix r show r ∈ {ce-set-to-rel A |A. A ∈ ce-sets} ⟹ r ∈ ce-rels
    proof −
      assume A: r ∈ {ce-set-to-rel A |A. A ∈ ce-sets}
      then obtain A where L1: r = ce-set-to-rel A and L2: A ∈ ce-sets by auto
      from L1 have ce-rel-to-set r = A by simp
      with L2 show r ∈ ce-rels unfolding ce-rels-def by auto
    qed
  qed
qed

lemma ce-rel-to-set-inj: inj ce-rel-to-set
proof (rule inj-on-inverseI)
  fix x assume A: (x::(nat×nat) set) ∈ UNIV show ce-set-to-rel (ce-rel-to-set x)
= x by (rule ce-rel-lm-1)
qed

lemma ce-rel-to-set-srj: surj ce-rel-to-set
```

```
proof (rule surjI [where ?f=ce-set-to-rel])
  fix x show ce-rel-to-set (ce-set-to-rel x) = x by (rule ce-rel-lm-2)
qed

lemma ce-rel-to-set-bij: bij ce-rel-to-set
proof (rule bijI)
  show inj ce-rel-to-set by (rule ce-rel-to-set-inj)
next
  show surj ce-rel-to-set by (rule ce-rel-to-set-srj)
qed

lemma ce-set-to-rel-inj: inj ce-set-to-rel
proof (rule inj-on-inverseI)
  fix x assume A: (x::nat set) ∈ UNIV show ce-rel-to-set (ce-set-to-rel x) = x by
(rule ce-rel-lm-2)
qed

lemma ce-set-to-rel-srj: surj ce-set-to-rel
proof (rule surjI [where ?f=ce-rel-to-set])
  fix x show ce-set-to-rel (ce-rel-to-set x) = x by (rule ce-rel-lm-1)
qed

lemma ce-set-to-rel-bij: bij ce-set-to-rel
proof (rule bijI)
  show inj ce-set-to-rel by (rule ce-set-to-rel-inj)
next
  show surj ce-set-to-rel by (rule ce-set-to-rel-srj)
qed

lemma ce-rel-lm-3: A ∈ ce-sets ⟹ ce-set-to-rel A ∈ ce-rels
proof –
  assume A: A ∈ ce-sets
  from A ce-rels-def1 show ?thesis by auto
qed

lemma ce-rel-lm-4: ce-set-to-rel A ∈ ce-rels ⟹ A ∈ ce-sets
proof –
  assume A: ce-set-to-rel A ∈ ce-rels
  from A show ?thesis by (unfold ce-rels-def, auto)
qed

lemma ce-rel-lm-5: (A ∈ ce-sets) = (ce-set-to-rel A ∈ ce-rels)
proof
  assume A ∈ ce-sets then show ce-set-to-rel A ∈ ce-rels by (rule ce-rel-lm-3)
next
  assume ce-set-to-rel A ∈ ce-rels then show A ∈ ce-sets by (rule ce-rel-lm-4)
qed

lemma ce-rel-lm-6: r ∈ ce-rels ⟹ ce-rel-to-set r ∈ ce-sets
```

proof –
  assume *A*: *r* ∈ *ce-rels*
  then show *?thesis* **by** (*unfold ce-rels-def*, *auto*)
qed

lemma *ce-rel-lm-7*: *ce-rel-to-set r* ∈ *ce-sets* ⟹ *r* ∈ *ce-rels*
proof –
  assume *ce-rel-to-set r* ∈ *ce-sets*
  then show *?thesis* **by** (*unfold ce-rels-def*, *auto*)
qed

lemma *ce-rel-lm-8*: (*r* ∈ *ce-rels*) = (*ce-rel-to-set r* ∈ *ce-sets*) **by** (*unfold ce-rels-def*, *auto*)

lemma *ce-rel-lm-9*: (*x*,*y*) ∈ *r* ⟹ *c-pair x y* ∈ *ce-rel-to-set r* **by** (*unfold ce-rel-to-set-def*, *auto*)

lemma *ce-rel-lm-10*: *x* ∈ *A* ⟹ (*c-fst x*, *c-snd x*) ∈ *ce-set-to-rel A* **by** (*unfold ce-set-to-rel-def*, *auto*)

lemma *ce-rel-lm-11*: *c-pair x y* ∈ *ce-rel-to-set r* ⟹ (*x*,*y*) ∈ *r*
proof –
  assume *A*: *c-pair x y* ∈ *ce-rel-to-set r*
  let *?z* = *c-pair x y*
  from *A* **have** (*c-fst ?z*, *c-snd ?z*) ∈ *ce-set-to-rel* (*ce-rel-to-set r*) **by** (*rule ce-rel-lm-10*)
  then show (*x*,*y*) ∈ *r* **by** *simp*
qed

lemma *ce-rel-lm-12*: (*c-pair x y* ∈ *ce-rel-to-set r*) = ((*x*,*y*) ∈ *r*)
proof
  assume *c-pair x y* ∈ *ce-rel-to-set r* **then show** (*x*, *y*) ∈ *r* **by** (*rule ce-rel-lm-11*)
next
  assume (*x*, *y*) ∈ *r* **then show** *c-pair x y* ∈ *ce-rel-to-set r* **by** (*rule ce-rel-lm-9*)
qed

lemma *ce-rel-lm-13*: (*x*,*y*) ∈ *ce-set-to-rel A* ⟹ *c-pair x y* ∈ *A*
proof –
  assume (*x*,*y*) ∈ *ce-set-to-rel A*
  then have *c-pair x y* ∈ *ce-rel-to-set* (*ce-set-to-rel A*) **by** (*rule ce-rel-lm-9*)
  then show *?thesis* **by** *simp*
qed

lemma *ce-rel-lm-14*: *c-pair x y* ∈ *A* ⟹ (*x*,*y*) ∈ *ce-set-to-rel A*
proof –
  assume *c-pair x y* ∈ *A*
  then have *c-pair x y* ∈ *ce-rel-to-set* (*ce-set-to-rel A*) **by** *simp*
  then show *?thesis* **by** (*rule ce-rel-lm-11*)
qed

```
lemma ce-rel-lm-15: ((x,y) ∈ ce-set-to-rel A) = (c-pair x y ∈ A)
proof
 assume (x, y) ∈ ce-set-to-rel A then show c-pair x y ∈ A by (rule ce-rel-lm-13)
next
 assume c-pair x y ∈ A then show (x, y) ∈ ce-set-to-rel A by (rule ce-rel-lm-14)
qed

lemma ce-rel-lm-16: x ∈ ce-rel-to-set r ⟹ (c-fst x, c-snd x) ∈ r
proof −
  assume x ∈ ce-rel-to-set r
 then have (c-fst x, c-snd x) ∈ ce-set-to-rel (ce-rel-to-set r) by (rule ce-rel-lm-10)
  then show ?thesis by simp
qed

lemma ce-rel-lm-17: (c-fst x, c-snd x) ∈ ce-set-to-rel A ⟹ x ∈ A
proof −
  assume (c-fst x, c-snd x) ∈ ce-set-to-rel A
  then have c-pair (c-fst x) (c-snd x) ∈ A by (rule ce-rel-lm-13)
  then show ?thesis by simp
qed

lemma ce-rel-lm-18: ((c-fst x, c-snd x) ∈ ce-set-to-rel A) = (x ∈ A)
proof
 assume (c-fst x, c-snd x) ∈ ce-set-to-rel A then show x ∈ A by (rule ce-rel-lm-17)
next
 assume x ∈ A then show (c-fst x, c-snd x) ∈ ce-set-to-rel A by (rule ce-rel-lm-10)
qed

lemma ce-rel-lm-19: (c-fst x, c-snd x) ∈ r ⟹ x ∈ ce-rel-to-set r
proof −
  assume (c-fst x, c-snd x) ∈ r
  then have (c-fst x, c-snd x) ∈ ce-set-to-rel (ce-rel-to-set r) by simp
  then show ?thesis by (rule ce-rel-lm-17)
qed

lemma ce-rel-lm-20: ((c-fst x, c-snd x) ∈ r) = (x ∈ ce-rel-to-set r)
proof
 assume (c-fst x, c-snd x) ∈ r then show x ∈ ce-rel-to-set r by (rule ce-rel-lm-19)
next
 assume x ∈ ce-rel-to-set r then show (c-fst x, c-snd x) ∈ r by (rule ce-rel-lm-16)
qed

lemma ce-rel-lm-21: r ∈ ce-rels ⟹ ∃ p ∈ PrimRec3. ∀ x y. ((x,y) ∈ r) = (∃ u.
p x y u = 0)
proof −
  assume r-ce: r ∈ ce-rels
  define A where A = ce-rel-to-set r
  from r-ce have A-ce: A ∈ ce-sets by (unfold A-def, rule ce-rel-lm-6)
  then have ∃ p ∈ PrimRec2. A = fn-to-set p by (rule ce-set-lm-3)
```

```
  then obtain q where q-is-pr: q ∈ PrimRec2 and A-def1: A = fn-to-set q ..
  from A-def1 have A-def2: A = { x. ∃ y. q x y = 0} by (unfold fn-to-set-def)
  define p where p x y u = q (c-pair x y) u for x y u
  from q-is-pr have p-is-pr: p ∈ PrimRec3 unfolding p-def by prec
  have ⋀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0)
  proof − fix x y show ((x,y) ∈ r) = (∃ u. p x y u = 0)
    proof
      assume A: (x,y) ∈ r
      define z where z = c-pair x y
      with A-def A have z-in-A: z ∈ A by (unfold ce-rel-to-set-def, auto)
      with A-def2 have z ∈ { x. ∃ y. q x y = 0} by auto
      then obtain u where q z u = 0 by auto
      with z-def have p x y u = 0 by (simp add: z-def p-def)
      then show ∃ u. p x y u = 0 by auto
    next
      assume A: ∃ u. p x y u = 0
      define z where z = c-pair x y
      from A obtain u where p x y u = 0 by auto
      then have q-z: q z u = 0 by (simp add: z-def p-def)
      with A-def2 have z-in-A: z ∈ A by auto
      then have c-pair x y ∈ A by (unfold z-def)
      then have c-pair x y ∈ ce-rel-to-set r by (unfold A-def)
      then show (x,y) ∈ r by (rule ce-rel-lm-11)
    qed
  qed
  with p-is-pr show ?thesis by auto
qed

lemma ce-rel-lm-22: r ∈ ce-rels ⟹ ∃ p ∈ PrimRec3. r = { (x,y). ∃ u. p x y u
= 0 }
proof −
  assume r-ce: r ∈ ce-rels
  then have ∃ p ∈ PrimRec3. ∀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0) by (rule
ce-rel-lm-21)
  then obtain p where p-is-pr: p ∈ PrimRec3 and L1: ∀ x y. ((x,y) ∈ r) = (∃
u. p x y u = 0) by auto
  from p-is-pr L1 show ?thesis by blast
qed

lemma ce-rel-lm-23: ⟦ p ∈ PrimRec3; ∀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0) ⟧
⟹ r ∈ ce-rels
proof −
  assume p-is-pr: p ∈ PrimRec3
  assume A: ∀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0)
  define q where q z u = p (c-fst z) (c-snd z) u for z u
  from p-is-pr have q-is-pr: q ∈ PrimRec2 unfolding q-def by prec
  define A where A = { x. ∃ y. q x y = 0}
  then have A-def1: A = fn-to-set q by (unfold fn-to-set-def, auto)
  from q-is-pr A-def1 have A-ce: A ∈ ce-sets by (simp add: ce-set-lm-1)
```

```
  have main: A = ce-rel-to-set r
  proof
    show A ⊆ ce-rel-to-set r
    proof
      fix z assume z-in-A: z ∈ A
      show z ∈ ce-rel-to-set r
      proof −
        define x where x = c-fst z
        define y where y = c-snd z
        from z-in-A A-def obtain u where L2: q z u = 0 by auto
        with x-def y-def q-def have L3: p x y u = 0 by simp
        then have ∃ u. p x y u = 0 by auto
        with A have (x,y) ∈ r by auto
        then have c-pair x y ∈ ce-rel-to-set r by (rule ce-rel-lm-9)
        with x-def y-def show ?thesis by simp
      qed
    qed
  next
    show ce-rel-to-set r ⊆ A
    proof
      fix z assume z-in-r: z ∈ ce-rel-to-set r
      show z ∈ A
      proof −
        define x where x = c-fst z
        define y where y = c-snd z
        from z-in-r have (c-fst z, c-snd z) ∈ r by (rule ce-rel-lm-16)
        with x-def y-def have (x,y) ∈ r by simp
        with A obtain u where L1: p x y u = 0 by auto
        with x-def y-def q-def have q z u = 0 by simp
        with A-def show z ∈ A by auto
      qed
    qed
  qed
  with A-ce have ce-rel-to-set r ∈ ce-sets by auto
  then show r ∈ ce-rels by (rule ce-rel-lm-7)
qed

lemma ce-rel-lm-24: ⟦ r ∈ ce-rels; s ∈ ce-rels ⟧ ⟹ s O r ∈ ce-rels
proof −
  assume r-ce: r ∈ ce-rels
  assume s-ce: s ∈ ce-rels
  from r-ce have ∃ p ∈ PrimRec3. ∀ x y. ((x,y) ∈ r)=(∃ u. p x y u = 0) by
(rule ce-rel-lm-21)
  then obtain p-r where p-r-is-pr: p-r ∈ PrimRec3 and R1: ∀ x y. ((x,y) ∈
r)=(∃ u. p-r x y u = 0)
    by auto
  from s-ce have ∃ p ∈ PrimRec3. ∀ x y. ((x,y) ∈ s)=(∃ u. p x y u = 0) by
(rule ce-rel-lm-21)
  then obtain p-s where p-s-is-pr: p-s ∈ PrimRec3 and S1: ∀ x y. ((x,y) ∈
```

```
s)=(∃ u. p-s x y u = 0)
    by auto
  define p where p x z u = (p-s x (c-fst u) (c-fst (c-snd u))) + (p-r (c-fst u) z
(c-snd (c-snd u)))
    for x z u
  from p-r-is-pr p-s-is-pr have p-is-pr: p ∈ PrimRec3 unfolding p-def by prec
  define sr where sr = s O r
  have main: ∀ x z. ((x,z) ∈ sr) = (∃ u. p x z u = 0)
  proof (rule allI, rule allI)
    fix x z
    show ((x, z) ∈ sr) = (∃u. p x z u = 0)
    proof
      assume A: (x, z) ∈ sr
      show ∃u. p x z u = 0
      proof −
        from A sr-def obtain y where L1: (x,y) ∈ s and L2: (y,z) ∈ r by auto
        from L1 S1 obtain u-s where L3: p-s x y u-s = 0 by auto
        from L2 R1 obtain u-r where L4: p-r y z u-r = 0 by auto
        define u where u = c-pair y (c-pair u-s u-r)
        from L3 L4 have p x z u = 0 by (unfold p-def, unfold u-def, simp)
        then show ?thesis by auto
      qed
    next
      assume A: ∃u. p x z u = 0
      show (x, z) ∈ sr
      proof −
        from A obtain u where L1: p x z u = 0 by auto
         then have L2: (p-s x (c-fst u) (c-fst (c-snd u))) + (p-r (c-fst u) z (c-snd
(c-snd u))) = 0 by (unfold p-def)
        from L2 have L3: p-s x (c-fst u) (c-fst (c-snd u)) = 0 by auto
        from L2 have L4: p-r (c-fst u) z (c-snd (c-snd u)) = 0 by auto
        from L3 S1 have L5: (x,(c-fst u)) ∈ s by auto
        from L4 R1 have L6: ((c-fst u),z) ∈ r by auto
        from L5 L6 have (x,z) ∈ s O r by auto
        with sr-def show ?thesis by auto
      qed
    qed
  qed
  from p-is-pr main have sr ∈ ce-rels by (rule ce-rel-lm-23)
  then show ?thesis by (unfold sr-def)
qed

lemma ce-rel-lm-25: r ∈ ce-rels =⇒ r^-1 ∈ ce-rels
proof −
  assume r-ce: r ∈ ce-rels
  have r^-1 = {(y,x). (x,y) ∈ r} by auto
  then have L1: ∀ x y. ((x,y) ∈ r) = ((y,x) ∈ r^-1) by auto
  from r-ce have ∃ p ∈ PrimRec3. ∀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0) by
(rule ce-rel-lm-21)
```

```
  then obtain p where p-is-pr: p ∈ PrimRec3 and R1: ∀ x y. ((x,y) ∈ r) = (∃ u. p x y u = 0) by auto
  define q where q x y u = p y x u for x y u
  from p-is-pr have q-is-pr: q ∈ PrimRec3 unfolding q-def by prec
  from L1 R1 have L2: ∀ x y. ((x,y) ∈ r^−1) = (∃ u. p y x u = 0) by auto
  with q-def have L3: ∀ x y. ((x,y) ∈ r^−1) = (∃ u. q x y u = 0) by auto
  with q-is-pr show ?thesis by (rule ce-rel-lm-23)
qed

lemma ce-rel-lm-26: r ∈ ce-rels ⟹ Domain r ∈ ce-sets
proof −
  assume r-ce: r ∈ ce-rels
  have L1: ∀ x. (x ∈ Domain r) = (∃ y. (x,y) ∈ r) by auto
  define A where A = ce-rel-to-set r
  from r-ce have ce-rel-to-set r ∈ ce-sets by (rule ce-rel-lm-6)
  then have A-ce: A ∈ ce-sets by (unfold A-def)
  have ∀ x y. ((x,y) ∈ r) = (c-pair x y ∈ ce-rel-to-set r) by (simp add: ce-rel-lm-12)
  then have L2: ∀ x y. ((x,y) ∈ r) = (c-pair x y ∈ A) by (unfold A-def)
  from A-ce c-fst-is-pr have L3: { c-fst z |z. z ∈ A } ∈ ce-sets by (rule ce-set-lm-7)
  have L4: ∀ x. (x ∈ { c-fst z |z. z ∈ A }) =(∃ y. c-pair x y ∈ A)
  proof fix x show (x ∈ { c-fst z |z. z ∈ A }) =(∃ y. c-pair x y ∈ A)
    proof
      assume A: x ∈ {c-fst z |z. z ∈ A}
      then obtain z where z-in-A: z ∈ A and x-z: x = c-fst z by auto
      from x-z have z = c-pair x (c-snd z) by simp
      with z-in-A have c-pair x (c-snd z) ∈ A by auto
      then show ∃ y. c-pair x y ∈ A by auto
    next
      assume A: ∃ y. c-pair x y ∈ A
      then obtain y where y-1: c-pair x y ∈ A by auto
      define z where z = c-pair x y
      from y-1 have z-in-A: z ∈ A by (unfold z-def)
      from z-def have x-z: x = c-fst z by (unfold z-def, simp)
      from z-in-A x-z show x ∈ {c-fst z |z. z ∈ A} by auto
    qed
  qed
  from L1 L2 have L5: ∀ x. (x ∈ Domain r) = (∃ y. c-pair x y ∈ A) by auto
  from L4 L5 have L6: ∀ x. (x ∈ Domain r) = (x ∈ { c-fst z |z. z ∈ A }) by auto
  then have Domain r = { c-fst z |z. z ∈ A } by auto
  with L3 show Domain r ∈ ce-sets by auto
qed

lemma ce-rel-lm-27: r ∈ ce-rels ⟹ Range r ∈ ce-sets
proof −
  assume r-ce: r ∈ ce-rels
  then have r^−1 ∈ ce-rels by (rule ce-rel-lm-25)
  then have Domain (r^−1) ∈ ce-sets by (rule ce-rel-lm-26)
  then show ?thesis by (unfold Domain-converse [symmetric])
```

qed

lemma *ce-rel-lm-28*: *r* ∈ *ce-rels* ⟹ *Field r* ∈ *ce-sets*
proof −
  assume *r-ce*: *r* ∈ *ce-rels*
  from *r-ce* **have** *L1*: *Domain r* ∈ *ce-sets* **by** (*rule ce-rel-lm-26*)
  from *r-ce* **have** *L2*: *Range r* ∈ *ce-sets* **by** (*rule ce-rel-lm-27*)
  from *L1 L2* **have** *L3*: *Domain r* ∪ *Range r* ∈ *ce-sets* **by** (*rule ce-union*)
  then show *?thesis* **by** (*unfold Field-def*)
qed

lemma *ce-rel-lm-29*: ⟦ *A* ∈ *ce-sets*; *B* ∈ *ce-sets* ⟧ ⟹ *A* × *B* ∈ *ce-rels*
proof −
  assume *A-ce*: *A* ∈ *ce-sets*
  assume *B-ce*: *B* ∈ *ce-sets*
  define *r-a* **where** *r-a* = {(*x*,(*0*::*nat*)) | *x*. *x* ∈ *A*}
  define *r-b* **where** *r-b* = {((*0*::*nat*),*z*) | *z*. *z* ∈ *B*}
  have *L1*: *r-a O r-b* = *A* × *B* **by** (*unfold r-a-def*, *unfold r-b-def*, *auto*)
  have *r-a-ce*: *r-a* ∈ *ce-rels*
  proof −
   have *loc1*: *ce-rel-to-set r-a* = { *c-pair x 0* | *x*. *x* ∈ *A*} **by** (*unfold r-a-def*, *unfold ce-rel-to-set-def*, *auto*)
    define *p* **where** *p x* = *c-pair x 0* **for** *x*
    have *p-is-pr*: *p* ∈ *PrimRec1* **unfolding** *p-def* **by** *prec*
    from *A-ce p-is-pr* **have** { *c-pair x 0* | *x*. *x* ∈ *A*} ∈ *ce-sets*
      unfolding *p-def* **by** (*simp add*: *ce-set-lm-7*)
    with *loc1* **have** *ce-rel-to-set r-a* ∈ *ce-sets* **by** *auto*
    then show *?thesis* **by** (*rule ce-rel-lm-7*)
  qed
  have *r-b-ce*: *r-b* ∈ *ce-rels*
  proof −
    have *loc1*: *ce-rel-to-set r-b* = { *c-pair 0 z* | *z*. *z* ∈ *B*}
      by (*unfold r-b-def*, *unfold ce-rel-to-set-def*, *auto*)
    define *p* **where** *p z* = *c-pair 0 z* **for** *z*
    have *p-is-pr*: *p* ∈ *PrimRec1* **unfolding** *p-def* **by** *prec*
    from *B-ce p-is-pr* **have** { *c-pair 0 z* | *z*. *z* ∈ *B*} ∈ *ce-sets*
      unfolding *p-def* **by** (*simp add*: *ce-set-lm-7*)
    with *loc1* **have** *ce-rel-to-set r-b* ∈ *ce-sets* **by** *auto*
    then show *?thesis* **by** (*rule ce-rel-lm-7*)
  qed
  from *r-b-ce r-a-ce* **have** *r-a O r-b* ∈ *ce-rels* **by** (*rule ce-rel-lm-24*)
  with *L1* **show** *?thesis* **by** *auto*
qed

lemma *ce-rel-lm-30*: {} ∈ *ce-rels*
proof −
  have *ce-rel-to-set* {} = {} **by** (*unfold ce-rel-to-set-def*, *auto*)
  with *ce-empty* **have** *ce-rel-to-set* {} ∈ *ce-sets* **by** *auto*
  then show *?thesis* **by** (*rule ce-rel-lm-7*)

```
qed

lemma ce-rel-lm-31: UNIV ∈ ce-rels
proof −
  from ce-univ ce-univ have UNIV × UNIV ∈ ce-rels by (rule ce-rel-lm-29)
  then show ?thesis by auto
qed

lemma ce-rel-lm-32: ce-rel-to-set (r ∪ s) = (ce-rel-to-set r) ∪ (ce-rel-to-set s) by
(unfold ce-rel-to-set-def, auto)

lemma ce-rel-lm-33: ⟦ r ∈ ce-rels; s ∈ ce-rels ⟧ ⟹ r ∪ s ∈ ce-rels
proof −
  assume r ∈ ce-rels
  then have r-ce: ce-rel-to-set r ∈ ce-sets by (rule ce-rel-lm-6)
  assume s ∈ ce-rels
  then have s-ce: ce-rel-to-set s ∈ ce-sets by (rule ce-rel-lm-6)
 have ce-rel-to-set (r ∪ s) = (ce-rel-to-set r) ∪ (ce-rel-to-set s) by (unfold ce-rel-to-set-def,
auto)
  moreover from r-ce s-ce have (ce-rel-to-set r) ∪ (ce-rel-to-set s) ∈ ce-sets by
(rule ce-union)
  ultimately have ce-rel-to-set (r ∪ s) ∈ ce-sets by auto
  then show ?thesis by (rule ce-rel-lm-7)
qed

lemma ce-rel-lm-34: ce-rel-to-set (r ∩ s) = (ce-rel-to-set r) ∩ (ce-rel-to-set s)
proof
 show ce-rel-to-set (r ∩ s) ⊆ ce-rel-to-set r ∩ ce-rel-to-set s by (unfold ce-rel-to-set-def,
auto)
next
  show ce-rel-to-set r ∩ ce-rel-to-set s ⊆ ce-rel-to-set (r ∩ s)
  proof fix x assume A: x ∈ ce-rel-to-set r ∩ ce-rel-to-set s
    from A have L1: x ∈ ce-rel-to-set r by auto
    from A have L2: x ∈ ce-rel-to-set s by auto
    from L1 obtain u v where L3: (u,v) ∈ r and L4: x = c-pair u v
      unfolding ce-rel-to-set-def by auto
    from L2 obtain u1 v1 where L5: (u1,v1) ∈ s and L6: x = c-pair u1 v1
      unfolding ce-rel-to-set-def by auto
    from L4 L6 have L7: c-pair u1 v1 = c-pair u v by auto
    then have u1=u by (rule c-pair-inj1)
    moreover from L7 have v1=v by (rule c-pair-inj2)
    ultimately have (u,v)=(u1,v1) by auto
    with L3 L5 have (u,v) ∈ r ∩ s by auto
    with L4 show x ∈ ce-rel-to-set (r ∩ s) by (unfold ce-rel-to-set-def, auto)
  qed
qed

lemma ce-rel-lm-35: ⟦ r ∈ ce-rels; s ∈ ce-rels ⟧ ⟹ r ∩ s ∈ ce-rels
proof −
```

```
  assume r ∈ ce-rels
  then have r-ce: ce-rel-to-set r ∈ ce-sets by (rule ce-rel-lm-6)
  assume s ∈ ce-rels
  then have s-ce: ce-rel-to-set s ∈ ce-sets by (rule ce-rel-lm-6)
 have ce-rel-to-set (r ∩ s) = (ce-rel-to-set r) ∩ (ce-rel-to-set s) by (rule ce-rel-lm-34)
  moreover from r-ce s-ce have (ce-rel-to-set r) ∩ (ce-rel-to-set s) ∈ ce-sets by
(rule ce-intersect)
  ultimately have ce-rel-to-set (r ∩ s) ∈ ce-sets by auto
  then show ?thesis by (rule ce-rel-lm-7)
qed

lemma ce-rel-lm-36: ce-set-to-rel (A ∪ B) = (ce-set-to-rel A) ∪ (ce-set-to-rel B)
  by (unfold ce-set-to-rel-def, auto)

lemma ce-rel-lm-37: ce-set-to-rel (A ∩ B) = (ce-set-to-rel A) ∩ (ce-set-to-rel B)
proof −
  define f where f x = (c-fst x, c-snd x) for x
  have f-inj: inj f
  proof (unfold f-def, rule inj-on-inverseI [where ?g=λ (u,v). c-pair u v])
    fix x :: nat
    assume x ∈ UNIV
    show case-prod c-pair (c-fst x, c-snd x) = x by simp
  qed
  from f-inj have f ‘ (A ∩ B) = f ‘ A ∩ f ‘ B by (rule image-Int)
  then show ?thesis by (unfold f-def, unfold ce-set-to-rel-def, auto)
qed

lemma ce-rel-lm-38: ⟦ r ∈ ce-rels; A ∈ ce-sets ⟧ ⟹ r‘‘A ∈ ce-sets
proof −
  assume r-ce: r ∈ ce-rels
  assume A-ce: A ∈ ce-sets
  have L1: r‘‘A = Range (r ∩ A × UNIV) by blast
  have L2: Range (r ∩ A × UNIV) ∈ ce-sets
  proof (rule ce-rel-lm-27)
    show r ∩ A × UNIV ∈ ce-rels
    proof (rule ce-rel-lm-35)
      show r ∈ ce-rels by (rule r-ce)
    next
      show A × UNIV ∈ ce-rels
      proof (rule ce-rel-lm-29)
        show A ∈ ce-sets by (rule A-ce)
      next
        show UNIV ∈ ce-sets by (rule ce-univ)
      qed
    qed
  qed
  from L1 L2 show ?thesis by auto
qed
```

7.6 Total computable functions

definition
graph :: (*nat* ⇒ *nat*) ⇒ (*nat* × *nat*) *set* **where**
graph = (λ *f*. { (*x*, *f x*) | *x*. *x* ∈ *UNIV*})

lemma *graph-lm-1*: (*x*,*y*) ∈ *graph f* ⟹ *y* = *f x* **by** (*unfold graph-def*, *auto*)

lemma *graph-lm-2*: *y* = *f x* ⟹ (*x*,*y*) ∈ *graph f* **by** (*unfold graph-def*, *auto*)

lemma *graph-lm-3*: ((*x*,*y*) ∈ *graph f*) = (*y* = *f x*) **by** (*unfold graph-def*, *auto*)

lemma *graph-lm-4*: *graph* (*f o g*) = (*graph g*) *O* (*graph f*) **by** (*unfold graph-def*, *auto*)

definition
c-graph :: (*nat* ⇒ *nat*) ⇒ *nat set* **where**
c-graph = (λ *f*. { *c-pair x* (*f x*) | *x*. *x* ∈ *UNIV*})

lemma *c-graph-lm-1*: *c-pair x y* ∈ *c-graph f* ⟹ *y* = *f x*
proof −
  assume *A*: *c-pair x y* ∈ *c-graph f*
  have *S1*: *c-graph f* = {*c-pair x* (*f x*) | *x*. *x* ∈ *UNIV*} **by** (*simp add*: *c-graph-def*)
  from *A S1* **obtain** *z* **where** *S2*: *c-pair x y* = *c-pair z* (*f z*) **by** *auto*
  then have *x* = *z* **by** (*rule c-pair-inj1*)
  moreover from *S2* **have** *y* = *f z* **by** (*rule c-pair-inj2*)
  ultimately show *?thesis* **by** *auto*
qed

lemma *c-graph-lm-2*: *y* = *f x* ⟹ *c-pair x y* ∈ *c-graph f* **by** (*unfold c-graph-def*, *auto*)

lemma *c-graph-lm-3*: (*c-pair x y* ∈ *c-graph f*) = (*y* = *f x*)
proof
  assume *c-pair x y* ∈ *c-graph f* **then show** *y* = *f x* **by** (*rule c-graph-lm-1*)
next
  assume *y* = *f x* **then show** *c-pair x y* ∈ *c-graph f* **by** (*rule c-graph-lm-2*)
qed

lemma *c-graph-lm-4*: *c-graph f* = *ce-rel-to-set* (*graph f*) **by** (*unfold c-graph-def ce-rel-to-set-def graph-def*, *auto*)

lemma *c-graph-lm-5*: *graph f* = *ce-set-to-rel* (*c-graph f*) **by** (*simp add*: *c-graph-lm-4*)

definition
total-recursive :: (*nat* ⇒ *nat*) ⇒ *bool* **where**
total-recursive = (λ *f*. *graph f* ∈ *ce-rels*)

lemma *total-recursive-def1*: *total-recursive* = (λ *f*. *c-graph f* ∈ *ce-sets*)
proof (*rule ext*) **fix** *f* **show** *total-recursive f* = (*c-graph f* ∈ *ce-sets*)

```
  proof
    assume A: total-recursive f
    then have graph f ∈ ce-rels by (unfold total-recursive-def)
    then have ce-rel-to-set (graph f) ∈ ce-sets by (rule ce-rel-lm-6)
    then show c-graph f ∈ ce-sets by (simp add: c-graph-lm-4)
  next
    assume c-graph f ∈ ce-sets
    then have ce-rel-to-set (graph f) ∈ ce-sets by (simp add: c-graph-lm-4)
    then have graph f ∈ ce-rels by (rule ce-rel-lm-7)
    then show total-recursive f by (unfold total-recursive-def)
  qed
qed

theorem pr-is-total-rec: f ∈ PrimRec1 ⟹ total-recursive f
proof −
  assume A: f ∈ PrimRec1
  define p where p x = c-pair x (f x) for x
  from A have p-is-pr: p ∈ PrimRec1 unfolding p-def by prec
  let ?U = { p x | x. x ∈ UNIV }
  from ce-univ p-is-pr have U-ce: ?U ∈ ce-sets by (rule ce-set-lm-7)
  have U-1: ?U = { c-pair x (f x) | x. x ∈ UNIV} by (simp add: p-def)
  with U-ce have S1: { c-pair x (f x) | x. x ∈ UNIV} ∈ ce-sets by simp
 with c-graph-def have c-graph-f-is-ce: c-graph f ∈ ce-sets by (unfold c-graph-def,
auto)
  then show ?thesis by (unfold total-recursive-def1, auto)
qed

theorem comp-tot-rec: ⟦ total-recursive f; total-recursive g ⟧ ⟹ total-recursive (f
o g)
proof −
  assume total-recursive f
  then have f-ce: graph f ∈ ce-rels by (unfold total-recursive-def)
  assume total-recursive g
  then have g-ce: graph g ∈ ce-rels by (unfold total-recursive-def)
  from f-ce g-ce have graph g O graph f ∈ ce-rels by (rule ce-rel-lm-24)
  then have graph (f o g) ∈ ce-rels by (simp add: graph-lm-4)
  then show ?thesis by (unfold total-recursive-def)
qed

lemma univ-for-pr-tot-rec-lm: c-graph univ-for-pr ∈ ce-sets
proof −
  define A where A = c-graph univ-for-pr
  from A-def have S1: A = { c-pair x (univ-for-pr x) | x. x ∈ UNIV }
    by (simp add: c-graph-def)
  from S1 have S2: A = { z . ∃ x. z = c-pair x (univ-for-pr x) } by auto
  have S3: ⋀ z. (∃ x. (z = c-pair x (univ-for-pr x))) = (univ-for-pr (c-fst z) =
c-snd z)
  proof −
    fix z show (∃ x. (z = c-pair x (univ-for-pr x))) = (univ-for-pr (c-fst z) =
```

```
c-snd z)
    proof
      assume A: ∃ x. z = c-pair x (univ-for-pr x)
      then obtain x where S3-1: z = c-pair x (univ-for-pr x) ..
      then show univ-for-pr (c-fst z) = c-snd z by simp
    next
      assume A: univ-for-pr (c-fst z) = c-snd z
      from A have z = c-pair (c-fst z) (univ-for-pr (c-fst z)) by simp
      thus ∃ x. z = c-pair x (univ-for-pr x) ..
    qed
  qed
  with S2 have S4: A = { z . univ-for-pr (c-fst z) = c-snd z } by auto
  define p where p x y =
    (if c-assoc-have-key (pr-gr y) (c-fst x) = 0 then
      (if c-assoc-value (pr-gr y) (c-fst x) = c-snd x then (0::nat) else 1)
    else 1) for x y
   from c-assoc-have-key-is-pr c-assoc-value-is-pr pr-gr-is-pr have p-is-pr: p ∈
PrimRec2
    unfolding p-def by prec
  have S5: ⋀ z. (univ-for-pr (c-fst z) = c-snd z) = (∃ y. p z y = 0)
  proof -
    fix z show (univ-for-pr (c-fst z) = c-snd z) = (∃ y. p z y = 0)
    proof
      assume A: univ-for-pr (c-fst z) = c-snd z
      let ?n = c-fst (c-fst z)
      let ?x = c-snd (c-fst z)
      let ?y = loc-upb ?n ?x
    have S5-1: c-assoc-have-key (pr-gr ?y) (c-pair ?n ?x) = 0 by (rule loc-upb-main)
      have S5-2: c-assoc-value (pr-gr ?y) (c-pair ?n ?x) = univ-for-pr (c-pair ?n
?x) by (rule pr-gr-value)
      from S5-1 have S5-3: c-assoc-have-key (pr-gr ?y) (c-fst z) = 0 by simp
      from S5-2 A have S5-4: c-assoc-value (pr-gr ?y) (c-fst z) = c-snd z by simp
      from S5-3 S5-4 have p z ?y = 0 by (simp add: p-def)
      thus ∃ y. p z y = 0 ..
    next
      assume A: ∃ y. p z y = 0
      then obtain y where S5-1: p z y = 0 ..
      have S5-2: c-assoc-have-key (pr-gr y) (c-fst z) = 0
      proof (rule ccontr)
        assume A-1: c-assoc-have-key (pr-gr y) (c-fst z) ≠ 0
        then have p z y = 1 by (simp add: p-def)
        with S5-1 show False by auto
      qed
      then have S5-3: p z y = (if c-assoc-value (pr-gr y) (c-fst z) = c-snd z then
(0::nat) else 1) by (simp add: p-def)
      have S5-4: c-assoc-value (pr-gr y) (c-fst z) = c-snd z
      proof (rule ccontr)
        assume A-2: c-assoc-value (pr-gr y) (c-fst z) ≠ c-snd z
        then have p z y = 1 by (simp add: p-def)
```

```
        with S5-1 show False by auto
      qed
      have S5-5: c-is-sub-fun (pr-gr y) univ-for-pr by (rule pr-gr-1)
      from S5-5 S5-2 have S5-6: c-assoc-value (pr-gr y) (c-fst z) = univ-for-pr (c-fst z) by (rule c-is-sub-fun-lm-1)
      with S5-4 show univ-for-pr (c-fst z) = c-snd z by auto
    qed
  qed
  from S5 S4 have A = {z. ∃ y. p z y = 0} by auto
  then have A = fn-to-set p by (simp add: fn-to-set-def)
  moreover from p-is-pr have fn-to-set p ∈ ce-sets by (rule ce-set-lm-1)
  ultimately have A ∈ ce-sets by auto
  with A-def show ?thesis by auto
qed

theorem univ-for-pr-tot-rec: total-recursive univ-for-pr
proof –
  have c-graph univ-for-pr ∈ ce-sets by (rule univ-for-pr-tot-rec-lm)
  then show ?thesis by (unfold total-recursive-def1, auto)
qed
```

7.7 Computable sets, Post's theorem

```
definition
  computable :: nat set ⇒ bool where
  computable = (λ A. A ∈ ce-sets ∧ −A ∈ ce-sets)

lemma computable-complement-1: computable A ⟹ computable (− A)
proof –
  assume computable A
  then show ?thesis by (unfold computable-def, auto)
qed

lemma computable-complement-2: computable (− A) ⟹ computable A
proof –
  assume computable (− A)
  then show ?thesis by (unfold computable-def, auto)
qed

lemma computable-complement-3: (computable A) = (computable (− A)) by (unfold computable-def, auto)

theorem comp-impl-tot-rec: computable A ⟹ total-recursive (chf A)
proof –
  assume A: computable A
  from A have A1: A ∈ ce-sets by (unfold computable-def, simp)
  from A have A2: −A ∈ ce-sets by (unfold computable-def, simp)
  define p where p x = c-pair x 0 for x
  define q where q x = c-pair x 1 for x
```

from *p-def* **have** *p-is-pr*: *p* ∈ *PrimRec1* **unfolding** *p-def* **by** *prec*
from *q-def* **have** *q-is-pr*: *q* ∈ *PrimRec1* **unfolding** *q-def* **by** *prec*
define *U0* **where** *U0* = {*p x* | *x*. *x* ∈ *A*}
define *U1* **where** *U1* = {*q x* | *x*. *x* ∈ − *A*}
from *A1 p-is-pr* **have** *U0-ce*: *U0* ∈ *ce-sets* **by**(*unfold U0-def*, *rule ce-set-lm-7*)
from *A2 q-is-pr* **have** *U1-ce*: *U1* ∈ *ce-sets* **by**(*unfold U1-def*, *rule ce-set-lm-7*)
define *U* **where** *U* = *U0* ∪ *U1*
from *U0-ce U1-ce* **have** *U-ce*: *U* ∈ *ce-sets* **by** (*unfold U-def*, *rule ce-union*)
define *V* **where** *V* = *c-graph* (*chf A*)
have *V-1*: *V* = { *c-pair x* (*chf A x*) | *x*. *x* ∈ *UNIV*} **by** (*simp add*: *V-def c-graph-def*)
from *U0-def p-def* **have** *U0-1*: *U0* = { *c-pair x y* | *x y*. *x* ∈ *A* ∧ *y*=*0*} **by** *auto*
from *U1-def q-def* **have** *U1-1*: *U1* = { *c-pair x y* | *x y*. *x* ∉ *A* ∧ *y*=*1*} **by** *auto*
from *U0-1 U1-1 U-def* **have** *U-1*: *U* = { *c-pair x y* | *x y*. (*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)} **by** *auto*
from *V-1* **have** *V-2*: *V* = { *c-pair x y* | *x y*. *y* = *chf A x*} **by** *auto*
have *L1*: ⋀ *x y*. ((*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)) = (*y* = *chf A x*)
proof −
fix *x y*
show ((*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)) = (*y* = *chf A x*) **by**(*unfold chf-def*, *auto*)
qed
from *V-2 U-1 L1* **have** *U*=*V* **by** *simp*
with *U-ce* **have** *V-ce*: *V* ∈ *ce-sets* **by** *auto*
with *V-def* **have** *c-graph* (*chf A*) ∈ *ce-sets* **by** *auto*
then show *?thesis* **by** (*unfold total-recursive-def1*)
qed

theorem *tot-rec-impl-comp*: *total-recursive* (*chf A*) ⟹ *computable A*
proof −
assume *A*: *total-recursive* (*chf A*)
then have *A1*: *c-graph* (*chf A*) ∈ *ce-sets* **by** (*unfold total-recursive-def1*)
let *?U* = *c-graph* (*chf A*)
have *L1*: *?U* = { *c-pair x* (*chf A x*) | *x*. *x* ∈ *UNIV*} **by** (*simp add*: *c-graph-def*)
have *L2*: ⋀ *x y*. ((*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)) = (*y* = *chf A x*)
proof − **fix** *x y* **show** ((*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)) = (*y* = *chf A x*) **by**(*unfold chf-def*, *auto*)
qed
from *L1 L2* **have** *L3*: *?U* = { *c-pair x y* | *x y*. (*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)} **by** *auto*
define *p* **where** *p x* = *c-pair x 0* **for** *x*
define *q* **where** *q x* = *c-pair x 1* **for** *x*
have *p-is-pr*: *p* ∈ *PrimRec1* **unfolding** *p-def* **by** *prec*
have *q-is-pr*: *q* ∈ *PrimRec1* **unfolding** *q-def* **by** *prec*
define *V* **where** *V* = { *c-pair x y* | *x y*. (*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*)}
from *V-def L3 A1* **have** *V-ce*: *V* ∈ *ce-sets* **by** *auto*
from *V-def* **have** *L4*: ∀ *z*. (*z* ∈ *V*) = (∃ *x y*. *z* = *c-pair x y* ∧ ((*x* ∈ *A* ∧ *y*=*0*) ∨ (*x* ∉ *A* ∧ *y*=*1*))) **by** *blast*
have *L5*: ⋀ *x*. (*p x* ∈ *V*) = (*x* ∈ *A*)

```
  proof − fix x show (p x ∈ V) = (x ∈ A)
    proof
      assume A: p x ∈ V
      then have c-pair x 0 ∈ V by (unfold p-def)
      with V-def obtain x1 y1 where L5-2: c-pair x 0 = c-pair x1 y1
        and L5-3: ((x1 ∈ A ∧ y1=0) ∨ (x1 ∉ A ∧ y1=1)) by auto
      from L5-2 have X-eq-X1: x=x1 by (rule c-pair-inj1)
      from L5-2 have Y1-eq-0: 0=y1 by (rule c-pair-inj2)
      from L5-3 X-eq-X1 Y1-eq-0 show x ∈ A by auto
    next
      assume A: x ∈ A
      let ?z = c-pair x 0
      from A have L5-1: ∃ x1 y1. c-pair x 0 = c-pair x1 y1 ∧ ((x1 ∈ A ∧ y1=0) ∨ (x1 ∉ A ∧ y1=1)) by auto
      with V-def have c-pair x 0 ∈ V by auto
      with p-def show p x ∈ V by simp
    qed
  qed
  then have A-eq: A = { x. p x ∈ V} by auto
  from V-ce p-is-pr have { x. p x ∈ V} ∈ ce-sets by (rule ce-set-lm-5)
  with A-eq have A-ce: A ∈ ce-sets by simp
  have CA-eq: − A = {x. q x ∈ V}
  proof −
    have ⋀ x. (q x ∈ V) = (x ∉ A)
    proof − fix x show (q x ∈ V) = (x ∉ A)
      proof
        assume A: q x ∈ V
        then have c-pair x 1 ∈ V by (unfold q-def)
        with V-def obtain x1 y1 where L5-2: c-pair x 1 = c-pair x1 y1
          and L5-3: ((x1 ∈ A ∧ y1=0) ∨ (x1 ∉ A ∧ y1=1)) by auto
        from L5-2 have X-eq-X1: x=x1 by (rule c-pair-inj1)
        from L5-2 have Y1-eq-1: 1=y1 by (rule c-pair-inj2)
        from L5-3 X-eq-X1 Y1-eq-1 show x ∉ A by auto
      next
        assume A: x ∉ A
        from A have L5-1: ∃ x1 y1. c-pair x 1 = c-pair x1 y1 ∧ ((x1 ∈ A ∧ y1=0) ∨ (x1 ∉ A ∧ y1=1)) by auto
        with V-def have c-pair x 1 ∈ V by auto
        with q-def show q x ∈ V by simp
      qed
    qed
    then show ?thesis by auto
  qed
  from V-ce q-is-pr have { x. q x ∈ V} ∈ ce-sets by (rule ce-set-lm-5)
  with CA-eq have CA-ce: − A ∈ ce-sets by simp
  from A-ce CA-ce show ?thesis by (simp add: computable-def)
qed

theorem post-th-0: (computable A) = (total-recursive (chf A))
```

```
proof
 assume computable A then show total-recursive (chf A) by (rule comp-impl-tot-rec)
next
 assume total-recursive (chf A) then show computable A by (rule tot-rec-impl-comp)
qed
```

7.8 Universal computably enumerable set

```
definition
  univ-ce :: nat set where
  univ-ce = { c-pair n x | n x. x ∈ nat-to-ce-set n }

lemma univ-for-pr-lm: univ-for-pr (c-pair n x) = (nat-to-pr n) x
  by (simp add: univ-for-pr-def pr-conv-2-to-1-def)

theorem univ-is-ce: univ-ce ∈ ce-sets
proof −
  define A where A = c-graph univ-for-pr
  then have A ∈ ce-sets by (simp add: univ-for-pr-tot-rec-lm)
  then have ∃ pA ∈ PrimRec2. A = fn-to-set pA by (rule ce-set-lm-3)
   then obtain pA where pA-is-pr: pA ∈ PrimRec2 and S1: A = fn-to-set pA
by auto
  from S1 have S2: A = { x. ∃ y. pA x y = 0 } by (simp add: fn-to-set-def)
  define p where p z y = pA (c-pair (c-pair (c-fst z) (c-pair (c-snd z) (c-fst y)))
0) (c-snd y)
    for z y
  from pA-is-pr have p-is-pr: p ∈ PrimRec2 unfolding p-def by prec
  have ⋀ z. (∃ n x. z = c-pair n x ∧ x ∈ nat-to-ce-set n) = (c-snd z ∈ nat-to-ce-set
(c-fst z))
  proof −
     fix z show (∃ n x. z = c-pair n x ∧ x ∈ nat-to-ce-set n) = (c-snd z ∈
nat-to-ce-set (c-fst z))
    proof
      assume A: ∃n x. z = c-pair n x ∧ x ∈ nat-to-ce-set n
      then obtain n x where L1: z = c-pair n x ∧ x ∈ nat-to-ce-set n by auto
      from L1 have L2: z = c-pair n x by auto
      from L1 have L3: x ∈ nat-to-ce-set n by auto
      from L1 have L4: c-fst z = n by simp
      from L1 have L5: c-snd z = x by simp
      from L3 L4 L5 show c-snd z ∈ nat-to-ce-set (c-fst z) by auto
    next
      assume A: c-snd z ∈ nat-to-ce-set (c-fst z)
      let ?n = c-fst z
      let ?x = c-snd z
      have L1: z = c-pair ?n ?x by simp
      from L1 A have z = c-pair ?n ?x ∧ ?x ∈ nat-to-ce-set ?n by auto
      thus ∃n x. z = c-pair n x ∧ x ∈ nat-to-ce-set n by blast
    qed
  qed
```

```
  then have { c-pair n x | n x. x ∈ nat-to-ce-set n } = { z. c-snd z ∈ nat-to-ce-set
(c-fst z)} by auto
  then have S3: univ-ce  = { z. c-snd z ∈ nat-to-ce-set (c-fst z)} by (simp add:
univ-ce-def)
  have S4: ⋀ z. (c-snd z ∈ nat-to-ce-set (c-fst z)) = (∃ y. p z y = 0)
  proof −
    fix z show (c-snd z ∈ nat-to-ce-set (c-fst z)) = (∃ y. p z y = 0)
    proof
      assume A: c-snd z ∈ nat-to-ce-set (c-fst z)
      have nat-to-ce-set (c-fst z) = { x . ∃ y. (nat-to-pr (c-fst z)) (c-pair x y) = 0
} by (simp add: nat-to-ce-set-lm-1)
      with A obtain u where S4-1: (nat-to-pr (c-fst z)) (c-pair (c-snd z) u) = 0
by auto
       then have S4-2: univ-for-pr (c-pair (c-fst z) (c-pair (c-snd z) u)) = 0 by
(simp add: univ-for-pr-lm)
      from A-def have S4-3: A = { c-pair x (univ-for-pr x) | x. x ∈ UNIV } by
(simp add: c-graph-def)
      then have S4-4: ⋀ x. c-pair x (univ-for-pr x) ∈ A by auto
       then have c-pair (c-pair (c-fst z) (c-pair (c-snd z) u)) (univ-for-pr (c-pair
(c-fst z) (c-pair (c-snd z) u))) ∈ A by auto
      with S4-2 have S4-5: c-pair (c-pair (c-fst z) (c-pair (c-snd z) u)) 0 ∈ A by
auto
      with S2 obtain v where S4-6: pA (c-pair (c-pair (c-fst z) (c-pair (c-snd z)
u)) 0) v = 0
        by auto
      define y where y = c-pair u v
      from y-def have S4-7: u = c-fst y by simp
      from y-def have S4-8: v = c-snd y by simp
      from S4-6 S4-7 S4-8 p-def have p z y = 0 by simp
      thus ∃ y. p z y = 0 ..
    next
      assume A: ∃ y. p z y = 0
      then obtain y where S4-1: p z y = 0 ..
       from S4-1 p-def have S4-2: pA (c-pair (c-pair (c-fst z) (c-pair (c-snd z)
(c-fst y))) 0) (c-snd y) = 0 by simp
      with S2 have S4-3: c-pair (c-pair (c-fst z) (c-pair (c-snd z) (c-fst y))) 0 ∈
A by auto
       with A-def have c-pair (c-pair (c-fst z) (c-pair (c-snd z) (c-fst y))) 0 ∈
c-graph univ-for-pr by simp
     then have S4-4: 0 = univ-for-pr (c-pair (c-fst z) (c-pair (c-snd z) (c-fst y)))
by (rule c-graph-lm-1)
      then have S4-5: univ-for-pr (c-pair (c-fst z) (c-pair (c-snd z) (c-fst y))) =
0 by auto
       then have S4-6: (nat-to-pr (c-fst z)) (c-pair (c-snd z) (c-fst y)) = 0 by
(simp add: univ-for-pr-lm)
      then have S4-7: ∃ y. (nat-to-pr (c-fst z)) (c-pair (c-snd z) y) = 0 ..
      have S4-8: nat-to-ce-set (c-fst z) = { x . ∃ y. (nat-to-pr (c-fst z)) (c-pair x
y) = 0 } by (simp add: nat-to-ce-set-lm-1)
      from S4-7 have S4-9: c-snd z ∈ { x . ∃ y. (nat-to-pr (c-fst z)) (c-pair x y)
```

```
= 0 } by auto
      with S4-8 show c-snd z ∈ nat-to-ce-set (c-fst z) by auto
    qed
  qed
  with S3 have univ-ce = {z. ∃ y. p z y = 0} by auto
  then have univ-ce = fn-to-set p by (simp add: fn-to-set-def)
  moreover from p-is-pr have fn-to-set p ∈ ce-sets by (rule ce-set-lm-1)
  ultimately show univ-ce ∈ ce-sets by auto
qed

lemma univ-ce-lm-1: (c-pair n x ∈ univ-ce) = (x ∈ nat-to-ce-set n)
proof −
  from univ-ce-def have S1: univ-ce = {z . ∃ n x. z = c-pair n x ∧ x ∈ nat-to-ce-set
n} by auto
  have S2: (∃ n1 x1. c-pair n x = c-pair n1 x1 ∧ x1 ∈ nat-to-ce-set n1) = (x ∈
nat-to-ce-set n)
  proof
    assume ∃ n1 x1. c-pair n x = c-pair n1 x1 ∧ x1 ∈ nat-to-ce-set n1
     then obtain n1 x1 where L1: c-pair n x = c-pair n1 x1 and L2: x1 ∈
nat-to-ce-set n1 by auto
    from L1 have L3: n = n1 by (rule c-pair-inj1)
    from L1 have L4: x = x1 by (rule c-pair-inj2)
    from L2 L3 L4 show x ∈ nat-to-ce-set n by auto
  next
    assume A: x ∈ nat-to-ce-set n
    then have c-pair n x = c-pair n x ∧ x ∈ nat-to-ce-set n by auto
    thus ∃ n1 x1. c-pair n x = c-pair n1 x1 ∧ x1 ∈ nat-to-ce-set n1 by blast
  qed
  with S1 show ?thesis by auto
qed

theorem univ-ce-is-not-comp1: − univ-ce ∉ ce-sets
proof (rule ccontr)
  assume ¬ − univ-ce ∉ ce-sets
  then have A: − univ-ce ∈ ce-sets by auto
  define p where p x = c-pair x x for x
  have p-is-pr: p ∈ PrimRec1 unfolding p-def by prec
  define A where A = { x. p x ∈ − univ-ce }
  from A p-is-pr have { x. p x ∈ − univ-ce } ∈ ce-sets by (rule ce-set-lm-5)
  with A-def have S1: A ∈ ce-sets by auto
  then have ∃ n. A = nat-to-ce-set n by (rule nat-to-ce-set-srj)
  then obtain n where S2: A = nat-to-ce-set n ..
  from A-def have (n ∈ A) = (p n ∈ − univ-ce) by auto
  with p-def have (n ∈ A) = (c-pair n n ∉ univ-ce) by auto
  with univ-ce-def univ-ce-lm-1 have (n ∈ A) = (n ∉ nat-to-ce-set n) by auto
  with S2 have (n ∈ A) = (n ∉ A) by auto
  thus False by auto
qed
```

```
theorem univ-ce-is-not-comp2: ¬ total-recursive (chf univ-ce)
proof
  assume total-recursive (chf univ-ce)
  then have computable univ-ce by (rule tot-rec-impl-comp)
  then have − univ-ce ∈ ce-sets by (unfold computable-def, auto)
  with univ-ce-is-not-comp1 show False by auto
qed

theorem univ-ce-is-not-comp3: ¬ computable univ-ce
proof (rule ccontr)
  assume ¬ ¬ computable univ-ce
  then have computable univ-ce by auto
  then have total-recursive (chf univ-ce) by (rule comp-impl-tot-rec)
  with univ-ce-is-not-comp2 show False by auto
qed
```

7.9 s-1-1 theorem, one-one and many-one reducibilities

```
definition
  index-of-r-to-l :: nat where
  index-of-r-to-l =
  pair-by-index
    (pair-by-index index-of-c-fst (comp-by-index index-of-c-fst index-of-c-snd))
    (comp-by-index index-of-c-snd index-of-c-snd)

lemma index-of-r-to-l-lm: nat-to-pr index-of-r-to-l (c-pair x (c-pair y z)) = c-pair
(c-pair x y) z
  apply(unfold index-of-r-to-l-def)
  apply(simp add: pair-by-index-main)
  apply(unfold c-f-pair-def)
  apply(simp add: index-of-c-fst-main)
  apply(simp add: comp-by-index-main)
  apply(simp add: index-of-c-fst-main)
  apply(simp add: index-of-c-snd-main)
done

definition
  s-ce :: nat ⇒ nat ⇒ nat where
  s-ce == (λ e x. s1-1 (comp-by-index e index-of-r-to-l) x)

lemma s-ce-is-pr: s-ce ∈ PrimRec2
  unfolding s-ce-def using comp-by-index-is-pr s1-1-is-pr by prec

lemma s-ce-inj: s-ce e1 x1 = s-ce e2 x2 ⟹ e1=e2 ∧ x1=x2
proof −
  let ?n1 = index-of-r-to-l
  assume s-ce e1 x1 = s-ce e2 x2
  then have s1-1 (comp-by-index e1 ?n1) x1 = s1-1 (comp-by-index e2 ?n1) x2
by (unfold s-ce-def)
```

```
 then have L1: comp-by-index e1 ?n1 = comp-by-index e2 ?n1 ∧ x1=x2 by (rule
s1-1-inj)
  from L1 have comp-by-index e1 ?n1 = comp-by-index e2 ?n1 ..
  then have e1=e2 by (rule comp-by-index-inj1)
  moreover from L1 have x1=x2 by auto
  ultimately show ?thesis by auto
qed

lemma s-ce-inj1: s-ce e1 x = s-ce e2 x ⟹ e1=e2
proof −
  assume s-ce e1 x = s-ce e2 x
  then have e1=e2 ∧ x=x by (rule s-ce-inj)
  then show e1=e2 by auto
qed

lemma s-ce-inj2: s-ce e x1 = s-ce e x2 ⟹ x1=x2
proof −
  assume s-ce e x1 = s-ce e x2
  then have e=e ∧ x1=x2 by (rule s-ce-inj)
  then show x1=x2 by auto
qed

theorem s1-1-th1: ∀ n x y. ((nat-to-pr n) (c-pair x y)) = (nat-to-pr (s1-1 n x)) y
proof (rule allI, rule allI, rule allI)
  fix n x y show nat-to-pr n (c-pair x y) = nat-to-pr (s1-1 n x) y
  proof −
    have (λ y. (nat-to-pr n) (c-pair x y)) = nat-to-pr (s1-1 n x) by (rule s1-1-th)
    then show ?thesis by (simp add: fun-eq-iff)
  qed
qed

lemma s-lm: (nat-to-pr (s-ce e x)) (c-pair y z) = (nat-to-pr e) (c-pair (c-pair x
y) z)
proof −
  let ?n1 = index-of-r-to-l
  have (nat-to-pr (s-ce e x)) (c-pair y z) = nat-to-pr (s1-1 (comp-by-index e ?n1)
x) (c-pair y z) by (unfold s-ce-def, simp)
   also have ... = (nat-to-pr (comp-by-index e ?n1)) (c-pair x (c-pair y z)) by
(simp add: s1-1-th1)
  also have ... = (nat-to-pr e) ((nat-to-pr ?n1) (c-pair x (c-pair y z))) by (simp
add: comp-by-index-main)
  finally show ?thesis by (simp add: index-of-r-to-l-lm)
qed

theorem s-ce-1-1-th: (c-pair x y ∈ nat-to-ce-set e) = (y ∈ nat-to-ce-set (s-ce e x))
proof
  assume A: c-pair x y ∈ nat-to-ce-set e
  then obtain z where L1: (nat-to-pr e) (c-pair (c-pair x y) z) = 0
    by (auto simp add: nat-to-ce-set-lm-1)
```

have (*nat-to-pr* (*s-ce e x*)) (*c-pair y z*) = *0* **by** (*simp add*: *s-lm L1*)
with *nat-to-ce-set-lm-1* **show** *y* ∈ *nat-to-ce-set* (*s-ce e x*) **by** *auto*
next
assume *A*: *y* ∈ *nat-to-ce-set* (*s-ce e x*)
then obtain *z* **where** *L1*: (*nat-to-pr* (*s-ce e x*)) (*c-pair y z*) = *0*
by (*auto simp add*: *nat-to-ce-set-lm-1*)
then have (*nat-to-pr e*) (*c-pair* (*c-pair x y*) *z*) = *0* **by** (*simp add*: *s-lm*)
with *nat-to-ce-set-lm-1* **show** *c-pair x y* ∈ *nat-to-ce-set e* **by** *auto*
qed

definition
one-reducible-to-via :: (*nat set*) ⇒ (*nat set*) ⇒ (*nat* ⇒ *nat*) ⇒ *bool* **where**
one-reducible-to-via = (λ *A B f*. *total-recursive f* ∧ *inj f* ∧ (∀ *x*. (*x* ∈ *A*) = (*f x* ∈ *B*)))

definition
one-reducible-to :: (*nat set*) ⇒ (*nat set*) ⇒ *bool* **where**
one-reducible-to = (λ *A B*. ∃ *f*. *one-reducible-to-via A B f*)

definition
many-reducible-to-via :: (*nat set*) ⇒ (*nat set*) ⇒ (*nat* ⇒ *nat*) ⇒ *bool* **where**
many-reducible-to-via = (λ *A B f*. *total-recursive f* ∧ (∀ *x*. (*x* ∈ *A*) = (*f x* ∈ *B*)))

definition
many-reducible-to :: (*nat set*) ⇒ (*nat set*) ⇒ *bool* **where**
many-reducible-to = (λ *A B*. ∃ *f*. *many-reducible-to-via A B f*)

lemma *one-reducible-to-via-trans*: ⟦ *one-reducible-to-via A B f*; *one-reducible-to-via B C g* ⟧ ⟹ *one-reducible-to-via A C* (*g o f*)
proof −
assume *A1*: *one-reducible-to-via A B f*
assume *A2*: *one-reducible-to-via B C g*
from *A1* **have** *f-tr*: *total-recursive f* **by** (*unfold one-reducible-to-via-def*, *auto*)
from *A1* **have** *f-inj*: *inj f* **by** (*unfold one-reducible-to-via-def*, *auto*)
from *A1* **have** *L1*: ∀ *x*. (*x* ∈ *A*) = (*f x* ∈ *B*) **by** (*unfold one-reducible-to-via-def*, *auto*)
from *A2* **have** *g-tr*: *total-recursive g* **by** (*unfold one-reducible-to-via-def*, *auto*)
from *A2* **have** *g-inj*: *inj g* **by** (*unfold one-reducible-to-via-def*, *auto*)
from *A2* **have** *L2*: ∀ *x*. (*x* ∈ *B*) = (*g x* ∈ *C*) **by** (*unfold one-reducible-to-via-def*, *auto*)
from *g-tr f-tr* **have** *fg-tr*: *total-recursive* (*g o f*) **by** (*rule comp-tot-rec*)
from *g-inj f-inj* **have** *fg-inj*: *inj* (*g o f*) **by** (*rule inj-compose*)
from *L1 L2* **have** *L3*: (∀ *x*. (*x* ∈ *A*) = ((*g o f*) *x* ∈ *C*)) **by** *auto*
with *fg-tr fg-inj* **show** *?thesis* **by** (*unfold one-reducible-to-via-def*, *auto*)
qed

lemma *one-reducible-to-trans*: ⟦ *one-reducible-to A B*; *one-reducible-to B C* ⟧ ⟹ *one-reducible-to A C*

```
proof −
  assume one-reducible-to A B
 then obtain f where A1: one-reducible-to-via A B f unfolding one-reducible-to-def
by auto
  assume one-reducible-to B C
 then obtain g where A2: one-reducible-to-via B C g unfolding one-reducible-to-def
by auto
 from A1 A2 have one-reducible-to-via A C (g o f) by (rule one-reducible-to-via-trans)
  then show ?thesis unfolding one-reducible-to-def by auto
qed

lemma one-reducible-to-via-refl: one-reducible-to-via A A (λ x. x)
proof −
  have is-pr: (λ x. x) ∈ PrimRec1 by (rule pr-id1-1)
  then have is-tr: total-recursive (λ x. x) by (rule pr-is-total-rec)
  have is-inj: inj (λ x. x) by simp
  have L1: ∀ x. (x ∈ A) = (((λ x. x) x) ∈ A) by simp
  with is-tr is-inj show ?thesis by (unfold one-reducible-to-via-def, auto)
qed

lemma one-reducible-to-refl: one-reducible-to A A
proof −
  have one-reducible-to-via A A (λ x. x) by (rule one-reducible-to-via-refl)
  then show ?thesis by (unfold one-reducible-to-def, auto)
qed

lemma many-reducible-to-via-trans: ⟦ many-reducible-to-via A B f; many-reducible-to-via
B C g ⟧ ⟹ many-reducible-to-via A C (g o f)
proof −
  assume A1: many-reducible-to-via A B f
  assume A2: many-reducible-to-via B C g
  from A1 have f-tr: total-recursive f by (unfold many-reducible-to-via-def, auto)
 from A1 have L1: ∀ x. (x ∈ A) = (f x ∈ B) by (unfold many-reducible-to-via-def,
auto)
  from A2 have g-tr: total-recursive g by (unfold many-reducible-to-via-def, auto)
 from A2 have L2: ∀ x. (x ∈ B) = (g x ∈ C) by (unfold many-reducible-to-via-def,
auto)
  from g-tr f-tr have fg-tr: total-recursive (g o f) by (rule comp-tot-rec)
  from L1 L2 have L3: (∀ x. (x ∈ A) = ((g o f) x ∈ C)) by auto
  with fg-tr show ?thesis by (unfold many-reducible-to-via-def, auto)
qed

lemma many-reducible-to-trans: ⟦ many-reducible-to A B; many-reducible-to B C
⟧ ⟹ many-reducible-to A C
proof −
  assume many-reducible-to A B
  then obtain f where A1: many-reducible-to-via A B f
    unfolding many-reducible-to-def by auto
  assume many-reducible-to B C
```

```
  then obtain g where A2: many-reducible-to-via B C g
    unfolding many-reducible-to-def by auto
  from A1 A2 have many-reducible-to-via A C (g o f) by (rule many-reducible-to-via-trans)
  then show ?thesis unfolding many-reducible-to-def by auto
qed

lemma one-reducibility-via-is-many: one-reducible-to-via A B f ⟹ many-reducible-to-via
A B f
proof –
  assume A: one-reducible-to-via A B f
  from A have f-tr: total-recursive f by (unfold one-reducible-to-via-def, auto)
  from A have ∀ x. (x ∈ A) = (f x ∈ B) by (unfold one-reducible-to-via-def, auto)
  with f-tr show ?thesis by (unfold many-reducible-to-via-def, auto)
qed

lemma one-reducibility-is-many: one-reducible-to A B ⟹ many-reducible-to A B
proof –
  assume one-reducible-to A B
  then obtain f where A: one-reducible-to-via A B f
    unfolding one-reducible-to-def by auto
  then have many-reducible-to-via A B f by (rule one-reducibility-via-is-many)
  then show ?thesis unfolding many-reducible-to-def by auto
qed

lemma many-reducible-to-via-refl: many-reducible-to-via A A (λ x. x)
proof –
  have one-reducible-to-via A A (λ x. x) by (rule one-reducible-to-via-refl)
  then show ?thesis by (rule one-reducibility-via-is-many)
qed

lemma many-reducible-to-refl: many-reducible-to A A
proof –
  have one-reducible-to A A by (rule one-reducible-to-refl)
  then show ?thesis by (rule one-reducibility-is-many)
qed

theorem m-red-to-comp: ⟦ many-reducible-to A B; computable B ⟧ ⟹ computable
A
proof –
  assume many-reducible-to A B
  then obtain f where A1: many-reducible-to-via A B f
    unfolding many-reducible-to-def by auto
  from A1 have f-tr: total-recursive f by (unfold many-reducible-to-via-def, auto)
  from A1 have L1: ∀ x. (x ∈ A) = (f x ∈ B) by (unfold many-reducible-to-via-def,
auto)
  assume computable B
  then have L2: total-recursive (chf B) by (rule comp-impl-tot-rec)
  have L3: chf A = (chf B) o f
  proof fix x
```

```
    have chf A x = (chf B) (f x)
    proof cases
      assume A: x ∈ A
      then have L3-1: chf A x = 0 by (simp add: chf-lm-2)
      from A L1 have f x ∈ B by auto
      then have L3-2: (chf B) (f x) = 0 by (simp add: chf-lm-2)
      from L3-1 L3-2 show chf A x = (chf B) (f x) by auto
    next
      assume A: x ∉ A
      then have L3-1: chf A x = 1 by (simp add: chf-lm-3)
      from A L1 have f x ∉ B by auto
      then have L3-2: (chf B) (f x) = 1 by (simp add: chf-lm-3)
      from L3-1 L3-2 show chf A x = (chf B) (f x) by auto
    qed
    then show chf A x = (chf B ∘ f) x by auto
  qed
  from L2 f-tr have total-recursive (chf B ∘ f) by (rule comp-tot-rec)
  with L3 have total-recursive (chf A) by auto
  then show ?thesis by (rule tot-rec-impl-comp)
qed

lemma many-reducible-lm-1: many-reducible-to univ-ce A ⟹ ¬ computable A
proof (rule ccontr)
  assume A1: many-reducible-to univ-ce A
  assume ¬ ¬ computable A
  then have A2: computable A by auto
  from A1 A2 have computable univ-ce by (rule m-red-to-comp)
  with univ-ce-is-not-comp3 show False by auto
qed

lemma one-reducible-lm-1: one-reducible-to univ-ce A ⟹ ¬ computable A
proof −
  assume one-reducible-to univ-ce A
  then have many-reducible-to univ-ce A by (rule one-reducibility-is-many)
  then show ?thesis by (rule many-reducible-lm-1)
qed

lemma one-reducible-lm-2: one-reducible-to-via (nat-to-ce-set n) univ-ce (λ x. c-pair
n x)
proof −
  define f where f x = c-pair n x for x
  have f-is-pr: f ∈ PrimRec1 unfolding f-def by prec
  then have f-tr: total-recursive f by (rule pr-is-total-rec)
  have f-inj: inj f
  proof (rule injI)
    fix x y assume A: f x = f y
    then have c-pair n x = c-pair n y by (unfold f-def)
    then show x = y by (rule c-pair-inj2)
  qed
```

```
  have ∀ x. (x ∈ (nat-to-ce-set n)) = (f x ∈ univ-ce)
  proof fix x show (x ∈ nat-to-ce-set n) = (f x ∈ univ-ce) by (unfold f-def, simp add: univ-ce-lm-1)
  qed
   with f-tr f-inj show ?thesis by (unfold f-def, unfold one-reducible-to-via-def, auto)
qed
```

```
lemma one-reducible-lm-3: one-reducible-to (nat-to-ce-set n) univ-ce
proof −
   have one-reducible-to-via (nat-to-ce-set n) univ-ce (λ x. c-pair n x) by (rule one-reducible-lm-2)
  then show ?thesis by (unfold one-reducible-to-def, auto)
qed
```

```
lemma one-reducible-lm-4: A ∈ ce-sets ⟹ one-reducible-to A univ-ce
proof −
  assume A ∈ ce-sets
  then have ∃ n. A = nat-to-ce-set n by (rule nat-to-ce-set-srj)
  then obtain n where A = nat-to-ce-set n by auto
  with one-reducible-lm-3 show ?thesis by auto
qed
```

7.10 One-complete sets

```
definition
  one-complete :: nat set ⇒ bool where
   one-complete = (λ A. A ∈ ce-sets ∧ (∀ B. B ∈ ce-sets ⟶ one-reducible-to B A))
```

```
theorem univ-is-complete: one-complete univ-ce
proof (unfold one-complete-def)
  show univ-ce ∈ ce-sets ∧ (∀B. B ∈ ce-sets ⟶ one-reducible-to B univ-ce)
  proof
    show univ-ce ∈ ce-sets by (rule univ-is-ce)
  next
    show ∀B. B ∈ ce-sets ⟶ one-reducible-to B univ-ce
    proof (rule allI, rule impI)
       fix B assume B ∈ ce-sets then show one-reducible-to B univ-ce by (rule one-reducible-lm-4)
    qed
  qed
qed
```

7.11 Index sets, Rice's theorem

```
definition
  index-set :: nat set ⇒ bool where
  index-set = (λ A. ∀ n m. n ∈ A ∧ (nat-to-ce-set n = nat-to-ce-set m) ⟶ m ∈ A)
```

```
lemma index-set-lm-1: ⟦ index-set A; n∈ A; nat-to-ce-set n = nat-to-ce-set m ⟧
⟹ m ∈ A
proof −
  assume A1: index-set A
  assume A2: n ∈ A
  assume A3: nat-to-ce-set n = nat-to-ce-set m
  from A2 A3 have L1: n ∈ A ∧ (nat-to-ce-set n = nat-to-ce-set m) by auto
  from A1 have L2: ∀ n m. n ∈ A ∧ (nat-to-ce-set n = nat-to-ce-set m) ⟶ m
∈ A by (unfold index-set-def)
  from L1 L2 show ?thesis by auto
qed

lemma index-set-lm-2: index-set A ⟹ index-set (−A)
proof −
  assume A: index-set A
  show index-set (−A)
  proof (unfold index-set-def)
    show ∀ n m. n ∈ − A ∧ nat-to-ce-set n = nat-to-ce-set m ⟶ m ∈ − A
    proof (rule allI, rule allI, rule impI)
      fix n m assume A1: n ∈ − A ∧ nat-to-ce-set n = nat-to-ce-set m
      from A1 have A2: n ∈ −A by auto
      from A1 have A3: nat-to-ce-set m = nat-to-ce-set n by auto
      show m ∈ − A
      proof
        assume m ∈ A
        from A this A3 have n ∈ A by (rule index-set-lm-1)
        with A2 show False by auto
      qed
    qed
  qed
qed

lemma Rice-lm-1: ⟦ index-set A; A ≠ {}; A ≠ UNIV; ∃ n ∈ A. nat-to-ce-set n =
{} ⟧ ⟹ one-reducible-to univ-ce (− A)
proof −
  assume A1: index-set A
  assume A2: A ≠ {}
  assume A3: A ≠ UNIV
  assume ∃ n ∈ A. nat-to-ce-set n = {}
  then obtain e-0 where e-0-in-A: e-0 ∈ A and e-0-empty: nat-to-ce-set e-0 =
{} by auto
  from e-0-in-A A3 obtain e-1 where e-1-not-in-A: e-1 ∈ (− A) by auto
  with e-0-in-A have e-0-neq-e-1: e-0 ≠ e-1 by auto
  have nat-to-ce-set e-0 ≠ nat-to-ce-set e-1
  proof
    assume nat-to-ce-set e-0 = nat-to-ce-set e-1
    with A1 e-0-in-A have e-1 ∈ A by (rule index-set-lm-1)
    with e-1-not-in-A show False by auto
```

```
  qed
  with e-0-empty have e1-not-empty: nat-to-ce-set e-1 ≠ {} by auto
  define we-1 where we-1 = nat-to-ce-set e-1
  from e1-not-empty have we-1-not-empty: we-1 ≠ {} by (unfold we-1-def)
  define r where r = univ-ce × we-1
  have loc-lm-1: ⋀ x. x ∈ univ-ce ⟹ ∀ y. (y ∈ we-1) = ((x,y) ∈ r) by (unfold
r-def, auto)
  have loc-lm-2: ⋀ x. x ∉ univ-ce ⟹ ∀ y. (y ∈ {}) = ((x,y) ∈ r) by (unfold
r-def, auto)
  have r-ce: r ∈ ce-rels
  proof (unfold r-def, rule ce-rel-lm-29)
    show univ-ce ∈ ce-sets by (rule univ-is-ce)
    show we-1 ∈ ce-sets by (unfold we-1-def, rule nat-to-ce-set-into-ce)
  qed
  define we-n where we-n = ce-rel-to-set r
  from r-ce have we-n-ce: we-n ∈ ce-sets by (unfold we-n-def, rule ce-rel-lm-6)
  then have ∃ n. we-n = nat-to-ce-set n by (rule nat-to-ce-set-srj)
  then obtain n where we-n-df1: we-n = nat-to-ce-set n by auto
  define f where f x = s-ce n x for x
  from s-ce-is-pr have f-is-pr: f ∈ PrimRec1 unfolding f-def by prec
  then have f-tr: total-recursive f by (rule pr-is-total-rec)
  have f-inj: inj f
  proof (rule injI)
    fix x y
    assume f x = f y
    then have s-ce n x = s-ce n y by (unfold f-def)
    then show x = y by (rule s-ce-inj2)
  qed
  have loc-lm-3: ∀ x y. (c-pair x y ∈ we-n) = (y ∈ nat-to-ce-set (f x))
  proof (rule allI, rule allI)
    fix x y show (c-pair x y ∈ we-n) = (y ∈ nat-to-ce-set (f x)) by (unfold f-def,
unfold we-n-df1, simp add: s-ce-1-1-th)
  qed
  from A1 have loc-lm-4: index-set (− A) by (rule index-set-lm-2)
  have loc-lm-5: ∀ x. (x ∈ univ-ce) = (f x ∈ −A)
  proof fix x show (x ∈ univ-ce) = (f x ∈ −A)
    proof
      assume A: x ∈ univ-ce
      then have S1: ∀ y. (y ∈ we-1) = ((x,y) ∈ r) by (rule loc-lm-1)
      from ce-rel-lm-12 have ∀ y. (c-pair x y ∈ ce-rel-to-set r) = ((x,y) ∈ r) by
auto
      then have ∀ y. ((x,y) ∈ r) = (c-pair x y ∈ we-n) by (unfold we-n-def, auto)
      with S1 have ∀ y. (y ∈ we-1) = (c-pair x y ∈ we-n) by auto
      with loc-lm-3 have ∀ y. (y ∈ we-1) = (y ∈ nat-to-ce-set (f x)) by auto
      then have S2: we-1 = nat-to-ce-set (f x) by auto
      then have nat-to-ce-set e-1 = nat-to-ce-set (f x) by (unfold we-1-def)
      with loc-lm-4 e-1-not-in-A show f x ∈ −A by (rule index-set-lm-1)
    next
      show f x ∈ − A ⟹ x ∈ univ-ce
```

```
   proof (rule ccontr)
    assume fx-in-A: f x ∈ − A
    assume x-not-in-univ: x ∉ univ-ce
    then have S1: ∀ y. (y ∈ {}) = ((x,y) ∈ r) by (rule loc-lm-2)
    from ce-rel-lm-12 have ∀ y. (c-pair x y ∈ ce-rel-to-set r) = ((x,y) ∈ r) by auto
     then have ∀ y. ((x,y) ∈ r) = (c-pair x y ∈ we-n) by (unfold we-n-def, auto)
    with S1 have ∀ y. (y ∈ {}) = (c-pair x y ∈ we-n) by auto
    with loc-lm-3 have ∀ y. (y ∈ {}) = (y ∈ nat-to-ce-set (f x)) by auto
    then have S2: {} = nat-to-ce-set (f x) by auto
    then have nat-to-ce-set e-0 = nat-to-ce-set (f x) by (unfold e-0-empty)
    with A1 e-0-in-A have f x ∈ A by (rule index-set-lm-1)
    with fx-in-A show False by auto
   qed
  qed
 qed
 with f-tr f-inj have one-reducible-to-via univ-ce (−A) f by (unfold one-reducible-to-via-def, auto)
 then show ?thesis by (unfold one-reducible-to-def, auto)
qed

lemma Rice-lm-2: ⟦ index-set A; A ≠ {}; A ≠ UNIV; n ∈ A; nat-to-ce-set n = {} ⟧ ⟹ one-reducible-to univ-ce (− A)
proof −
 assume A1: index-set A
 assume A2: A ≠ {}
 assume A3: A ≠ UNIV
 assume A4: n ∈ A
 assume A5: nat-to-ce-set n = {}
 from A4 A5 have S1: ∃ n ∈ A. nat-to-ce-set n = {} by auto
 from A1 A2 A3 S1 show ?thesis by (rule Rice-lm-1)
qed

theorem Rice-1: ⟦ index-set A; A ≠ {}; A ≠ UNIV ⟧ ⟹ one-reducible-to univ-ce A ∨ one-reducible-to univ-ce (− A)
proof −
 assume A1: index-set A
 assume A2: A ≠ {}
 assume A3: A ≠ UNIV
 from ce-empty have ∃ n. {} = nat-to-ce-set n by (rule nat-to-ce-set-srj)
 then obtain n where n-empty: nat-to-ce-set n = {} by auto
 show ?thesis
 proof cases
  assume A: n ∈ A
   from A1 A2 A3 A n-empty have one-reducible-to univ-ce (− A) by (rule Rice-lm-2)
  then show ?thesis by auto
 next
```

```
    assume n ∉ A then have A: n ∈ − A by auto
    from A1 have S1: index-set (− A) by (rule index-set-lm-2)
    from A3 have S2: − A ≠ {} by auto
    from A2 have S3: − A ≠ UNIV by auto
    from S1 S2 S3 A n-empty have one-reducible-to univ-ce (− (− A)) by (rule Rice-lm-2)
    then have one-reducible-to univ-ce A by simp
    then show ?thesis by auto
  qed
qed

theorem Rice-2: ⟦ index-set A; A ≠ {}; A ≠ UNIV ⟧ ⟹ ¬ computable A
proof −
  assume A1: index-set A
  assume A2: A ≠ {}
  assume A3: A ≠ UNIV
  from A1 A2 A3 have one-reducible-to univ-ce A ∨ one-reducible-to univ-ce (− A) by (rule Rice-1)
  then have S1: ¬ one-reducible-to univ-ce A ⟶ one-reducible-to univ-ce (− A) by auto
  show ?thesis
  proof cases
    assume one-reducible-to univ-ce A
    then show ¬ computable A by (rule one-reducible-lm-1)
  next
    assume ¬ one-reducible-to univ-ce A
    with S1 have one-reducible-to univ-ce (− A) by auto
    then have ¬ computable (− A) by (rule one-reducible-lm-1)
    with computable-complement-3 show ¬ computable A by auto
  qed
qed

theorem Rice-3: ⟦ C ⊆ ce-sets; computable { n. nat-to-ce-set n ∈ C} ⟧ ⟹ C = {} ∨ C = ce-sets
proof (rule ccontr)
  assume A1: C ⊆ ce-sets
  assume A2: computable { n. nat-to-ce-set n ∈ C}
  assume A3: ¬ (C = {} ∨ C = ce-sets)
  from A3 have A4: C ≠ {} by auto
  from A3 have A5: C ≠ ce-sets by auto
  define A where A = { n. nat-to-ce-set n ∈ C}
  have S1: index-set A
  proof (unfold index-set-def)
    show ∀ n m. n ∈ A ∧ nat-to-ce-set n = nat-to-ce-set m ⟶ m ∈ A
    proof (rule allI, rule allI, rule impI)
      fix n m assume A1-1: n ∈ A ∧ nat-to-ce-set n = nat-to-ce-set m
      from A1-1 have n ∈ A by auto
      then have S1-1: nat-to-ce-set n ∈ C by (unfold A-def, auto)
      from A1-1 have nat-to-ce-set n = nat-to-ce-set m by auto
```

```
      with S1-1 have nat-to-ce-set m ∈ C by auto
      then show m ∈ A by (unfold A-def, auto)
    qed
  qed
  have S2: A ≠ {}
  proof −
    from A4 obtain B where S2-1: B ∈ C by auto
    with A1 have B ∈ ce-sets by auto
    then have ∃ n. B = nat-to-ce-set n by (rule nat-to-ce-set-srj)
    then obtain n where B = nat-to-ce-set n ..
    with S2-1 have nat-to-ce-set n ∈ C by auto
    then show ?thesis by (unfold A-def, auto)
  qed
  have S3: A ≠ UNIV
  proof −
    from A1 A5 obtain B where S2-1: B ∉ C and S2-2: B ∈ ce-sets by auto
    from S2-2 have ∃ n. B = nat-to-ce-set n by (rule nat-to-ce-set-srj)
    then obtain n where B = nat-to-ce-set n ..
    with S2-1 have nat-to-ce-set n ∉ C by auto
    then show ?thesis by (unfold A-def, auto)
  qed
  from S1 S2 S3 have ¬ computable A by (rule Rice-2)
  with A2 show False unfolding A-def by auto
qed

end
```

References

[1] Rogers. *Theory of recursive functions and effective computatibility.* 1967.

[2] Soare. *Recursively enumerable sets and degrees.* 1987.