

# Ramsey's Theorem

Tom Ridge

December 12, 2009

## Abstract

The infinite form of Ramsey's Theorem is proved following Boolos and Jeffrey, Chapter 26.

## Contents

<b>1</b>	<b>Infinite Sets and Related Concepts</b>	<b>1</b>
1.1	Infinite Sets . . . . .	1
1.2	Infinitely Many and Almost All . . . . .	8
1.3	Enumeration of an Infinite Set . . . . .	10
1.4	Miscellaneous . . . . .	11
<b>2</b>	<b>Ramsey's Theorem</b>	<b>11</b>
2.1	Library lemmas . . . . .	11
2.2	Dependent Choice Variant . . . . .	12
2.3	Partitions . . . . .	13
2.4	Ramsey's theorem . . . . .	13

## 1 Infinite Sets and Related Concepts

```
theory Infinite-Set
imports Main
begin
```

### 1.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

#### abbreviation

```
infinite :: 'a set  $\Rightarrow$  bool where
infinite S ==  $\neg$  finite S
```

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

**lemma** *infinite-imp-nonempty*:  $\text{infinite } S \implies S \neq \{\}$   
**by** *auto*

**lemma** *infinite-remove*:  
 $\text{infinite } S \implies \text{infinite } (S - \{a\})$   
**by** *simp*

**lemma** *Diff-infinite-finite*:  
**assumes**  $T$ : *finite*  $T$  **and**  $S$ : *infinite*  $S$   
**shows** *infinite*  $(S - T)$   
**using**  $T$

**proof** *induct*  
**from**  $S$   
**show** *infinite*  $(S - \{\})$  **by** *auto*  
**next**  
**fix**  $T x$   
**assume** *ih*: *infinite*  $(S - T)$   
**have**  $S - (\text{insert } x T) = (S - T) - \{x\}$   
**by** (*rule* *Diff-insert*)  
**with** *ih*  
**show** *infinite*  $(S - (\text{insert } x T))$   
**by** (*simp* *add*: *infinite-remove*)  
**qed**

**lemma** *Un-infinite*:  $\text{infinite } S \implies \text{infinite } (S \cup T)$   
**by** *simp*

**lemma** *infinite-super*:  
**assumes**  $T$ :  $S \subseteq T$  **and**  $S$ : *infinite*  $S$   
**shows** *infinite*  $T$   
**proof**  
**assume** *finite*  $T$   
**with**  $T$  **have** *finite*  $S$  **by** (*simp* *add*: *finite-subset*)  
**with**  $S$  **show** *False* **by** *simp*  
**qed**

As a concrete example, we prove that the set of natural numbers is infinite.

**lemma** *finite-nat-bounded*:  
**assumes**  $S$ : *finite*  $(S::\text{nat set})$   
**shows**  $\exists k. S \subseteq \{..<k\}$  (**is**  $\exists k. ?\text{bounded } S k$ )  
**using**  $S$   
**proof** *induct*  
**have**  $?\text{bounded } \{\} 0$  **by** *simp*  
**then show**  $\exists k. ?\text{bounded } \{\} k ..$   
**next**  
**fix**  $S x$   
**assume**  $\exists k. ?\text{bounded } S k$

```

then obtain  $k$  where  $k: ?bounded\ S\ k\ ..$ 
show  $\exists k. ?bounded\ (insert\ x\ S)\ k$ 
proof (cases  $x < k$ )
  case True
    with  $k$  show ?thesis by auto
  next
    case False
      with  $k$  have  $?bounded\ S\ (Suc\ x)$  by auto
      then show ?thesis by auto
qed
qed

lemma finite-nat-iff-bounded:
   $finite\ (S::nat\ set) = (\exists k. S \subseteq \{..<k\})$  (is  $?lhs = ?rhs$ )
proof
  assume ?lhs
  then show ?rhs by (rule finite-nat-bounded)
next
  assume ?rhs
  then obtain  $k$  where  $S \subseteq \{..<k\}$  ..
  then show finite  $S$ 
    by (rule finite-subset) simp
qed

lemma finite-nat-iff-bounded-le:
   $finite\ (S::nat\ set) = (\exists k. S \subseteq \{..k\})$  (is  $?lhs = ?rhs$ )
proof
  assume ?lhs
  then obtain  $k$  where  $S \subseteq \{..<k\}$ 
    by (blast dest: finite-nat-bounded)
  then have  $S \subseteq \{..k\}$  by auto
  then show ?rhs ..
next
  assume ?rhs
  then obtain  $k$  where  $S \subseteq \{..k\}$  ..
  then show finite  $S$ 
    by (rule finite-subset) simp
qed

lemma infinite-nat-iff-unbounded:
   $infinite\ (S::nat\ set) = (\forall m. \exists n. m < n \wedge n \in S)$ 
  (is  $?lhs = ?rhs$ )
proof
  assume ?lhs
  show ?rhs
  proof (rule ccontr)
    assume  $\neg ?rhs$ 
    then obtain  $m$  where  $m: \forall n. m < n \longrightarrow n \notin S$  by blast
    then have  $S \subseteq \{..m\}$ 

```

```

    by (auto simp add: sym [OF linorder-not-less])
  with ⟨?lhs⟩ show False
    by (simp add: finite-nat-iff-bounded-le)
qed
next
assume ?rhs
show ?lhs
proof
  assume finite S
  then obtain m where  $S \subseteq \{..m\}$ 
    by (auto simp add: finite-nat-iff-bounded-le)
  then have  $\forall n. m < n \longrightarrow n \notin S$  by auto
  with ⟨?rhs⟩ show False by blast
qed
qed

```

**lemma** *infinite-nat-iff-unbounded-le*:  
 $infinite (S::nat\ set) = (\forall m. \exists n. m \leq n \wedge n \in S)$   
(is ?lhs = ?rhs)

```

proof
  assume ?lhs
  show ?rhs
  proof
    fix m
    from ⟨?lhs⟩ obtain n where  $m < n \wedge n \in S$ 
      by (auto simp add: infinite-nat-iff-unbounded)
    then have  $m \leq n \wedge n \in S$  by simp
    then show  $\exists n. m \leq n \wedge n \in S ..$ 
  qed
next
assume ?rhs
show ?lhs
proof (auto simp add: infinite-nat-iff-unbounded)
  fix m
  from ⟨?rhs⟩ obtain n where  $Suc\ m \leq n \wedge n \in S$ 
    by blast
  then have  $m < n \wedge n \in S$  by simp
  then show  $\exists n. m < n \wedge n \in S ..$ 
qed
qed

```

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some  $k$ , there is some larger number that is an element of the set.

**lemma** *unbounded-k-infinite*:  
**assumes**  $k: \forall m. k < m \longrightarrow (\exists n. m < n \wedge n \in S)$   
**shows**  $infinite (S::nat\ set)$   
**proof** –  
{

```

fix  $m$  have  $\exists n. m < n \wedge n \in S$ 
proof (cases  $k < m$ )
  case True
    with  $k$  show ?thesis by blast
  next
    case False
    from  $k$  obtain  $n$  where  $Suc\ k < n \wedge n \in S$  by auto
    with False have  $m < n \wedge n \in S$  by auto
    then show ?thesis ..
  qed
}
then show ?thesis
  by (auto simp add: infinite-nat-iff-unbounded)
qed

```

```

lemma nat-infinite [simp]: infinite (UNIV :: nat set)
  by (auto simp add: infinite-nat-iff-unbounded)

```

```

lemma nat-not-finite [elim]: finite (UNIV::nat set)  $\implies R$ 
  by simp

```

Every infinite set contains a countable subset. More precisely we show that a set  $S$  is infinite if and only if there exists an injective function from the naturals into  $S$ .

```

lemma range-inj-infinite:
  inj ( $f :: nat \Rightarrow 'a$ )  $\implies$  infinite (range  $f$ )
proof
  assume finite (range  $f$ ) and inj  $f$ 
  then have finite (UNIV::nat set)
    by (rule finite-imageD)
  then show False by simp
qed

```

```

lemma int-infinite [simp]:
  shows infinite (UNIV::int set)
proof –
  from inj-int have infinite (range int) by (rule range-inj-infinite)
  moreover
  have range int  $\subseteq$  (UNIV::int set) by simp
  ultimately show infinite (UNIV::int set) by (simp add: infinite-super)
qed

```

The “only if” direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set  $S$ . The idea is to construct a sequence of non-empty and infinite subsets of  $S$  obtained by successively removing elements of  $S$ .

```

lemma linorder-injI:
  assumes hyp:  $!!x\ y. x < (y :: 'a :: linorder) \implies f\ x \neq f\ y$ 

```

```

shows inj f
proof (rule inj-onI)
  fix x y
  assume f-eq:  $f\ x = f\ y$ 
  show  $x = y$ 
  proof (rule linorder-cases)
    assume  $x < y$ 
    with hyp have  $f\ x \neq f\ y$  by blast
    with f-eq show ?thesis by simp
  next
    assume  $x = y$ 
    then show ?thesis .
  next
    assume  $y < x$ 
    with hyp have  $f\ y \neq f\ x$  by blast
    with f-eq show ?thesis by simp
  qed
qed

lemma infinite-countable-subset:
  assumes inf: infinite (S::'a set)
  shows  $\exists f. \text{inj } (f::\text{nat} \Rightarrow 'a) \wedge \text{range } f \subseteq S$ 
proof –
  def Sseq  $\equiv \text{nat-rec } S (\lambda n\ T. T - \{\text{SOME } e. e \in T\})$ 
  def pick  $\equiv \lambda n. (\text{SOME } e. e \in \text{Sseq } n)$ 
  have Sseq-inf:  $\bigwedge n. \text{infinite } (\text{Sseq } n)$ 
  proof –
    fix n
    show infinite (Sseq n)
    proof (induct n)
      from inf show infinite (Sseq 0)
      by (simp add: Sseq-def)
    next
      fix n
      assume infinite (Sseq n) then show infinite (Sseq (Suc n))
      by (simp add: Sseq-def infinite-remove)
    qed
  qed
  have Sseq-S:  $\bigwedge n. \text{Sseq } n \subseteq S$ 
  proof –
    fix n
    show  $\text{Sseq } n \subseteq S$ 
    by (induct n) (auto simp add: Sseq-def)
  qed
  have Sseq-pick:  $\bigwedge n. \text{pick } n \in \text{Sseq } n$ 
  proof –
    fix n
    show  $\text{pick } n \in \text{Sseq } n$ 
    proof (unfold pick-def, rule someI-ex)

```

```

    from Sseq-inf have infinite (Sseq n) .
    then have Sseq n ≠ {} by auto
    then show ∃x. x ∈ Sseq n by auto
  qed
  qed
  with Sseq-S have rng: range pick ⊆ S
  by auto
  have pick-Sseq-gt: ∧n m. pick n ∉ Sseq (n + Suc m)
  proof -
    fix n m
    show pick n ∉ Sseq (n + Suc m)
    by (induct m) (auto simp add: Sseq-def pick-def)
  qed
  have pick-pick: ∧n m. pick n ≠ pick (n + Suc m)
  proof -
    fix n m
    from Sseq-pick have pick (n + Suc m) ∈ Sseq (n + Suc m) .
    moreover from pick-Sseq-gt
    have pick n ∉ Sseq (n + Suc m) .
    ultimately show pick n ≠ pick (n + Suc m)
    by auto
  qed
  have inj: inj pick
  proof (rule linorder-injI)
    fix i j :: nat
    assume i < j
    show pick i ≠ pick j
    proof
      assume eq: pick i = pick j
      from (i < j) obtain k where j = i + Suc k
      by (auto simp add: less-iff-Suc-add)
      with pick-pick have pick i ≠ pick j by simp
      with eq show False by simp
    qed
  qed
  from rng inj show ?thesis by auto
  qed

```

**lemma** *infinite-iff-countable-subset*:

$infinite\ S = (\exists f. inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S)$

**by** (auto simp add: infinite-countable-subset range-inj-infinite infinite-super)

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

**lemma** *inf-img-fin-dom*:

**assumes** *img: finite (f'A)* **and** *dom: infinite A*

**shows**  $\exists y \in f'A. infinite\ (f - \{y\})$

**proof** (*rule ccontr*)  
**assume**  $\neg ?thesis$   
**with** *img* **have** *finite*  $(UN\ y:f'A. f -' \{y\})$  **by** (*blast intro: finite-UN-I*)  
**moreover** **have**  $A \subseteq (UN\ y:f'A. f -' \{y\})$  **by** *auto*  
**moreover** **note** *dom*  
**ultimately** **show** *False* **by** (*simp add: infinite-super*)  
**qed**

**lemma** *inf-img-fin-domE*:  
**assumes** *finite*  $(f'A)$  **and** *infinite*  $A$   
**obtains**  $y$  **where**  $y \in f'A$  **and** *infinite*  $(f -' \{y\})$   
**using** *assms* **by** (*blast dest: inf-img-fin-dom*)

## 1.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

**definition**  
 $Inf\text{-}many :: ('a \Rightarrow bool) \Rightarrow bool$  (**binder** *INFM* 10) **where**  
 $Inf\text{-}many\ P = infinite\ \{x. P\ x\}$

**definition**  
 $Alm\text{-}all :: ('a \Rightarrow bool) \Rightarrow bool$  (**binder** *MOST* 10) **where**  
 $Alm\text{-}all\ P = (\neg (INFM\ x. \neg P\ x))$

**notation** (*xsymbols*)  
 $Inf\text{-}many$  (**binder**  $\exists_{\infty}$  10) **and**  
 $Alm\text{-}all$  (**binder**  $\forall_{\infty}$  10)

**notation** (*HTML output*)  
 $Inf\text{-}many$  (**binder**  $\exists_{\infty}$  10) **and**  
 $Alm\text{-}all$  (**binder**  $\forall_{\infty}$  10)

**lemma** *INFM-EX*:  
 $(\exists_{\infty} x. P\ x) \Longrightarrow (\exists x. P\ x)$   
**unfolding** *Inf-many-def*  
**proof** (*rule ccontr*)  
**assume** *inf: infinite*  $\{x. P\ x\}$   
**assume**  $\neg ?thesis$  **then** **have**  $\{x. P\ x\} = \{\}$  **by** *simp*  
**then** **have** *finite*  $\{x. P\ x\}$  **by** *simp*  
**with** *inf* **show** *False* **by** *simp*  
**qed**

**lemma** *MOST-iff-finiteNeg*:  $(\forall_{\infty} x. P\ x) = finite\ \{x. \neg P\ x\}$   
**by** (*simp add: Alm-all-def Inf-many-def*)

**lemma** *ALL-MOST*:  $\forall x. P\ x \Longrightarrow \forall_{\infty} x. P\ x$   
**by** (*simp add: MOST-iff-finiteNeg*)

**lemma** *INFM-mono*:

**assumes**  $\text{inf}: \exists_{\infty} x. P x$  **and**  $q: \bigwedge x. P x \implies Q x$   
**shows**  $\exists_{\infty} x. Q x$

**proof** –

**from**  $\text{inf}$  **have**  $\text{infinite } \{x. P x\}$  **unfolding** *Inf-many-def* .

**moreover from**  $q$  **have**  $\{x. P x\} \subseteq \{x. Q x\}$  **by** *auto*

**ultimately show** *?thesis*

**by** (*simp add: Inf-many-def infinite-super*)

**qed**

**lemma** *MOST-mono*:  $\forall_{\infty} x. P x \implies (\bigwedge x. P x \implies Q x) \implies \forall_{\infty} x. Q x$

**unfolding** *Alm-all-def* **by** (*blast intro: INFM-mono*)

**lemma** *INFM-disj-distrib*:

$(\exists_{\infty} x. P x \vee Q x) \longleftrightarrow (\exists_{\infty} x. P x) \vee (\exists_{\infty} x. Q x)$

**unfolding** *Inf-many-def* **by** (*simp add: Collect-disj-eq*)

**lemma** *MOST-conj-distrib*:

$(\forall_{\infty} x. P x \wedge Q x) \longleftrightarrow (\forall_{\infty} x. P x) \wedge (\forall_{\infty} x. Q x)$

**unfolding** *Alm-all-def* **by** (*simp add: INFM-disj-distrib del: disj-not1*)

**lemma** *MOST-rev-mp*:

**assumes**  $\forall_{\infty} x. P x$  **and**  $\forall_{\infty} x. P x \longrightarrow Q x$

**shows**  $\forall_{\infty} x. Q x$

**proof** –

**have**  $\forall_{\infty} x. P x \wedge (P x \longrightarrow Q x)$

**using** *prems* **by** (*simp add: MOST-conj-distrib*)

**thus** *?thesis* **by** (*rule MOST-mono*) *simp*

**qed**

**lemma** *not-INFM* [*simp*]:  $\neg (\text{INFM } x. P x) \longleftrightarrow (\text{MOST } x. \neg P x)$

**unfolding** *Alm-all-def not-not ..*

**lemma** *not-MOST* [*simp*]:  $\neg (\text{MOST } x. P x) \longleftrightarrow (\text{INFM } x. \neg P x)$

**unfolding** *Alm-all-def not-not ..*

**lemma** *INFM-const* [*simp*]:  $(\text{INFM } x::'a. P) \longleftrightarrow P \wedge \text{infinite } (\text{UNIV}::'a \text{ set})$

**unfolding** *Inf-many-def* **by** *simp*

**lemma** *MOST-const* [*simp*]:  $(\text{MOST } x::'a. P) \longleftrightarrow P \vee \text{finite } (\text{UNIV}::'a \text{ set})$

**unfolding** *Alm-all-def* **by** *simp*

**lemma** *INFM-nat*:  $(\exists_{\infty} n. P (n::\text{nat})) = (\forall m. \exists n. m < n \wedge P n)$

**by** (*simp add: Inf-many-def infinite-nat-iff-unbounded*)

**lemma** *INFM-nat-le*:  $(\exists_{\infty} n. P (n::\text{nat})) = (\forall m. \exists n. m \leq n \wedge P n)$

**by** (*simp add: Inf-many-def infinite-nat-iff-unbounded-le*)

**lemma** *MOST-nat*:  $(\forall_{\infty} n. P (n::nat)) = (\exists m. \forall n. m < n \longrightarrow P n)$   
**by** (*simp add: Alm-all-def INFM-nat*)

**lemma** *MOST-nat-le*:  $(\forall_{\infty} n. P (n::nat)) = (\exists m. \forall n. m \leq n \longrightarrow P n)$   
**by** (*simp add: Alm-all-def INFM-nat-le*)

### 1.3 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

**consts**

*enumerate* :: 'a::wellorder set => (nat => 'a::wellorder)

**primrec**

*enumerate-0*: *enumerate* *S* 0 = (*LEAST* *n*. *n* ∈ *S*)

*enumerate-Suc*: *enumerate* *S* (*Suc* *n*) = *enumerate* (*S* - {*LEAST* *n*. *n* ∈ *S*}) *n*

**lemma** *enumerate-Suc'*:

*enumerate* *S* (*Suc* *n*) = *enumerate* (*S* - {*enumerate* *S* 0}) *n*

**by** *simp*

**lemma** *enumerate-in-set*: *infinite* *S*  $\implies$  *enumerate* *S* *n* : *S*

**apply** (*induct n arbitrary: S*)

**apply** (*fastsimp intro: LeastI dest!: infinite-imp-nonempty*)

**apply** *simp*

**apply** (*metis Collect-def Collect-mem-eq DiffE infinite-remove*)

**done**

**declare** *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

**lemma** *enumerate-step*: *infinite* *S*  $\implies$  *enumerate* *S* *n* < *enumerate* *S* (*Suc* *n*)

**apply** (*induct n arbitrary: S*)

**apply** (*rule order-le-neq-trans*)

**apply** (*simp add: enumerate-0 Least-le enumerate-in-set*)

**apply** (*simp only: enumerate-Suc'*)

**apply** (*subgoal-tac* *enumerate* (*S* - {*enumerate* *S* 0}) 0 : *S* - {*enumerate* *S* 0})

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

**apply** (*simp add: enumerate-in-set del: Diff-iff*)

**apply** (*simp add: enumerate-Suc'*)

**done**

**lemma** *enumerate-mono*: *m* < *n*  $\implies$  *infinite* *S*  $\implies$  *enumerate* *S* *m* < *enumerate* *S* *n*

**apply** (*erule less-Suc-induct*)

**apply** (*auto intro: enumerate-step*)

**done**

## 1.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

**definition**

*atmost-one* :: 'a set  $\Rightarrow$  bool **where**  
*atmost-one* S = ( $\forall x y. x \in S \wedge y \in S \longrightarrow x=y$ )

**lemma** *atmost-one-empty*:  $S = \{\}$   $\Longrightarrow$  *atmost-one* S  
**by** (*simp add: atmost-one-def*)

**lemma** *atmost-one-singleton*:  $S = \{x\}$   $\Longrightarrow$  *atmost-one* S  
**by** (*simp add: atmost-one-def*)

**lemma** *atmost-one-unique* [*elim*]: *atmost-one* S  $\Longrightarrow$   $x \in S \Longrightarrow y \in S \Longrightarrow y = x$   
**by** (*simp add: atmost-one-def*)

**end**

## 2 Ramsey's Theorem

**theory** *Ramsey*  
**imports** *Main Infinite-Set*  
**begin**

**declare** [*simp-depth-limit = 5*]

### 2.1 Library lemmas

**lemma** *infinite-inj-infinite-image*: *infinite* Z  $\Longrightarrow$  *inj-on* f Z  $\Longrightarrow$  *infinite* (f ' Z)  
**apply**(*rule ccontr*)  
**apply**(*simp*)  
**apply**(*force dest: finite-imageD*)  
**done**

**lemma** *infinite-dom-finite-rng*: [*infinite* A; *finite* (f ' A)]  $\Longrightarrow$  ? b : f ' A.  
*infinite* {a : A. f a = b}  
**apply**(*rule ccontr*) **apply**(*simp*)  
**apply**(*subgoal-tac UNION A (% b. {a : A. f a = f b}) = A*) **prefer** 2 **apply**(*blast*)  
**apply**(*subgoal-tac (UN c : f ' A. {a : A. f a = c}) = (UN b:A. {a : A. f a = f b})*) **prefer** 2 **apply**(*blast*)  
**apply**(*subgoal-tac finite (UN c:f ' A. {a : A. f a = c})*) **apply**(*force*)  
**apply**(*rule finite-UN-I*)  
**apply**(*auto*)  
**done**

**lemma** *infinite-mem*: *infinite* X  $\Longrightarrow$  ? x. x : X

```

apply(rule ccontr)
apply(force)
done

```

```

lemma not-empty-least: (Y::nat set) ~ = {} ==> ? m. m : Y & (! m'. m' : Y
--> m <= m')
apply(rule-tac x=LEAST x. x : Y in exI)
apply(rule)
apply(rule LeastI-ex) apply(force)
apply(rule)
apply(rule)
apply(rule Least-le)
apply(assumption)
done

```

## 2.2 Dependent Choice Variant

```

—
consts choice :: ('a => bool) => ('a => 'a => bool) => nat => 'a
primrec
  choice P R 0 = (SOME x. P x)
  choice P R (Suc n) = (let x = choice P R n in SOME y. P y & R x y)
—

```

```

lemma dc:
  (! x y z. R x y & R y z --> R x z)
  & (? x0. P x0)
  & (! x. P x --> (? y. P y & R x y))
  --> (? f::nat=>'b. (! n. P (f n)) & (! n m. R (f n) (f (n+m+1))))

```

```

apply(intro allI impI, elim exE conjE)
apply(rule-tac x=choice P R in exI)
apply(subgoal-tac (ALL n. P (choice P R n))) prefer 2
apply(rule, induct-tac n)
apply(simp add: Let-def) apply(rule someI-ex) apply(blast)
apply(simp add: Let-def) apply(subgoal-tac P (SOME y. P y & R (choice P R
na) y) & R (choice P R na) (SOME y. P y & R (choice P R na) y)) apply(blast)
apply(rule someI-ex) apply(blast)
apply(rule) apply(assumption)

```

```

apply(subgoal-tac ! n. R (choice P R n) (choice P R (Suc n))) prefer 2
apply(rule)
apply(simp add: Let-def)
apply(subgoal-tac P (SOME y. P y & R (choice P R n) y) & R (choice P R n)
(SOME y. P y & R (choice P R n) y)) apply(blast)
apply(rule someI-ex) apply(force)

```

```

apply(rule) apply(rule)
apply(induct-tac m)

```

```

apply(force)
apply(drule-tac x=n+na+1 in spec) back
apply(force simp del: choice.simps)
done

```

## 2.3 Partitions

**definition**

```

part :: nat => nat => 'a set => ('a set => nat) => bool where
part r s Y f = (! X. X <= Y & finite X & card X = r --> f X < s)

```

```

lemma part: [| infinite YY; part (Suc n) s YY f; yy : YY |] ==> part n s (YY
- {yy}) (%u. f (insert yy u))
apply(simp add: part-def)
apply(intro allI impI, elim exE conjE)
apply(drule-tac x=insert yy X in spec)
apply(force simp: card-Diff-singleton-if)
done

```

```

lemma part-subset: part (Suc n) s YY f ==> Y <= YY ==> part (Suc n) s Y
f
apply(simp add: part-def)
apply(blast)
done

```

## 2.4 Ramsey's theorem

**lemma** ramsey:

```

! (s::nat) (r::nat) (YY::'a set) (f::'a set => nat).
infinite YY
& (! X. X <= YY & finite X & card X = r --> f X < s)
--> (? Y' t'.
Y' <= YY
& infinite Y'
& t' < s
& (! X. X <= Y' & finite X & card X = r --> f X = t'))
apply(simp add: part-def[symmetric] del: ex-simps)
apply(rule, rule, rule-tac nat.induct)

```

```

apply(intro allI impI)
apply(rule-tac x=YY in exI)
apply(rule-tac x=f {} in exI)
apply(force simp: part-def)

```

```

apply(intro allI impI) apply(elim exE conjE)

```

**apply**(subgoal-tac

```

? g.
(! m::nat. let (y,Y,t) = (g m) in

```

```

y : YY & y ~: Y
& Y <= YY & infinite Y
& t < s
& (! X. X <= Y & finite X & card X = nat --> (f o insert y) X = t)
& (! m m'.
let (y, Y, t) = (g m) in
let (y', Y', t') = (g (m+m'+1)) in
y' : Y
& Y' <= Y
)
)
)
prefer 2
apply(cut-tac

P = % gn.
let (y, Y, t) = (gn) in
y : YY & y ~: Y
& Y <= YY & infinite Y
& t < s
& (! X. X <= Y & finite X & card X = nat --> (f o insert y) X = t)

and R = % gn gn'.
let (y, Y, t) = (gn) in
let (y', Y', t') = (gn') in
y' : Y
& Y' <= Y

in dc)
apply(erule impE)

apply(intro conjI)
apply(intro allI impI, elim conjE) apply(simp add: Let-def split-beta) ap-
ply(blast)
apply(subgoal-tac ? yy. yy : YY) prefer 2 apply(rule infinite-mem) ap-
ply(assumption)
apply(elim exE conjE)
apply(drule-tac x=YY - {yy} in spec) apply(drule-tac x=f o insert yy in
spec)
apply(erule impE) apply(simp) apply(rule part) apply(assumption+)
apply(elim exE conjE)
apply(rule-tac x=(yy, Y', t') in exI) apply(simp) apply(blast)
apply(intro allI impI)
apply(case-tac x) apply(rename-tac yx b Yx tx)
apply(subgoal-tac ? yx'. yx' : Yx) prefer 2 apply(rule infinite-mem) ap-
ply(force)
apply(elim exE conjE)
apply(drule-tac x=Yx - {yx'} in spec)

```

**apply**(*drule-tac*  $x=f$  *o insert*  $yx'$  **in** *spec*)  
**apply**(*erule* *impE*) **apply**(*simp*) **apply**(*elim* *exE* *conjE*) **apply**(*rule* *part*)  
**apply**(*assumption+*)  
**apply**(*rule* *part-subset*) **apply**(*assumption*) **apply**(*assumption*) **apply**(*assumption*)  
**apply**(*elim* *exE* *conjE*)  
**apply**(*rule-tac*  $x=(yx', Y', t')$  **in** *exI*) **apply**(*simp*) **apply**(*blast*)  
**apply**(*assumption*)  
  
**apply**(*elim* *exE* *conjE*)  
  
**apply**(*subgoal-tac* ?  $s'$ .  $s' < s$  & *infinite* { $n$ . ( $\% n$ . *let* ( $y, Y, t$ ) =  $g$  *n* *in*  $t$ )  $n = s'$ }) **prefer** 2  
  
**apply**(*subgoal-tac* ?  $s'$ : (( $\% n$ . *let* ( $Y, y, t$ ) =  $g$  *n* *in*  $t$ ) ' *UNIV*). *infinite* { $n$  : *UNIV*. (*let* ( $Y, y, t$ ) =  $g$  *n* *in*  $t$ ) =  $s'$ }) **prefer** 2  
**apply**(*rule* *infinite-dom-finite-rng*) **apply**(*simp*)  
**apply**(*simp* (*no-asm*) *add: finite-nat-iff-bounded*)  
**apply**(*rule-tac*  $x=s$  **in** *exI*)  
**apply**(*rule*)  
**apply**(*simp* *add: image-iff*) **apply**(*elim* *exE* *conjE*)  
**apply**(*drule-tac*  $x=xa$  **in** *spec*) **apply**(*force* *simp* *add: Let-def split-beta*)  
**apply**(*elim* *bexE* *conjE*) **apply**(*rule-tac*  $x=s'$  **in** *exI*) **apply**(*simp*)  
**apply**(*simp* *add: image-iff*) **apply**(*elim* *exE* *conjE*) **apply**(*drule-tac*  $x=x$  **in** *spec*) **apply**(*force* *simp: Let-def split-beta*)  
  
**apply**(*elim* *exE* *conjE*)  
**apply**(*rule-tac*  $x=(\% n$ . *let* ( $y, Y, t$ ) =  $g$  *n* *in*  $y$ ) ' { $n$ . ( $\% n$ . *let* ( $y, Y, t$ ) =  $g$  *n* *in*  $t$ )  $n = s'$ }) **in** *exI*)  
**apply**(*rule-tac*  $x=s'$  **in** *exI*)  
  
**apply**(*subgoal-tac* *inj* ( $\% n$ . *let* ( $y, Y, t$ ) =  $g$  *n* *in*  $y$ )) **prefer** 2  
**apply**(*simp* *add: inj-on-def*)  
  
**apply**(*subgoal-tac* *ALL*  $x y$ .  $x < y$  & (*let* ( $y, Y, t$ ) =  $g$  *x* *in*  $y$ ) = (*let* ( $y, Y, t$ ) =  $g$  *y* *in*  $y$ )  $\longrightarrow x = y$ )  
**apply**(*intro* *allI* *impI*)  
**apply**(*subgoal-tac*  $x < y$  |  $x = y$  |  $y < x$ ) **prefer** 2 **apply**(*arith*)  
**apply**(*elim* *disjE*)  
**apply**(*drule-tac*  $x=x$  **in** *spec*) **back** **back** **apply**(*drule-tac*  $x=y$  **in** *spec*) **back**  
**back** **apply**(*force* *simp: Let-def*)  
**apply**(*assumption*)  
**apply**(*drule-tac*  $x=y$  **in** *spec*) **back** **back** **apply**(*drule-tac*  $x=x$  **in** *spec*) **back**  
**back** **apply**(*force* *simp: Let-def*)  
**apply**(*intro* *allI* *impI*)  
**apply**(*drule-tac*  $x=x$  **in** *spec*) **apply**(*drule-tac*  $x=x$  **in** *spec*) **apply**(*drule-tac*  $x=y-(\text{Suc } x)$  **in** *spec*) **apply**(*force* *simp: Let-def*)  
  
**apply**(*intro* *allI* *conjI*)

```

apply(drule-tac  $x=xa$  in spec)
apply(force simp add: Let-def split-beta)

apply(rule infinite-inj-infinite-image) apply(assumption)
apply(simp add: inj-on-def)

apply(simp)

apply(intro allI impI, elim exE conjE)
apply(simp add: subset-image-iff) apply(elim exE conjE)
apply(subgoal-tac ? a. a : AA & (! a'. a' : AA --> a <= a')) prefer 2
apply(rule not-empty-least) apply(force)
apply(elim exE conjE)
apply(case-tac g a rule: prod.exhaust) apply(case-tac b rule: prod.exhaust) ap-
ply(rename-tac ya b Ya ta)
apply(subgoal-tac ya : X) prefer 2 apply(force intro!: rev-image-eqI simp:
Let-def)
apply(drule-tac s=X in sym)
apply(subgoal-tac f X = (f o insert ya) (X - {ya})) apply(simp)
apply(drule-tac x=a in spec)
apply(simp add: Let-def) apply(elim exE conjE)
apply(drule-tac x=X-{ya} in spec) back apply(erule impE)
apply(simp)
apply(rule)
apply(rule)
apply(drule-tac t=X in sym) apply(simp)
apply(simp add: image-iff) apply(elim bexE exE conjE) apply(rename-tac a')
apply(subgoal-tac a' ~ = a) prefer 2 apply(force)
apply(drule-tac x=a in spec)
apply(drule-tac x=a'-Suc a in spec) back
apply(simp add: Let-def split-beta) apply(case-tac g a' rule: prod.exhaust) ap-
ply(case-tac ba rule: prod.exhaust) apply(rename-tac ya' ba Ya' ta') apply(simp
add: Let-def split-beta)
apply(drule-tac x=a' in spec) apply(erule impE) apply(force)
apply(force)
apply(force simp add: card-Diff-singleton-if)
apply(subgoal-tac ta = s') apply(simp) apply(force)
apply(simp) apply(rule-tac f=f in arg-cong) apply(force)
done

end

```