Ramsey's Theorem

Tom Ridge

December 17, 2016

Abstract

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

Contents

1 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

The set of natural numbers is infinite.

lemma *infinite-nat-iff-unbounded-le*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*≥*m*. *n* ∈ *S*)
using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-sequentially*)

lemma *infinite-nat-iff-unbounded*: *infinite* (*S*::*nat set*) ⟷ (∀ *m*. ∃ *n*>*m*. *n* ∈ *S*)
using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-at-top-dense*)

lemma *finite-nat-iff-bounded*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {..<*k*})
using *infinite-nat-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-iff-bounded-le*: *finite* (*S*::*nat set*) ⟷ (∃ *k*. *S* ⊆ {.. *k*})
using *infinite-nat-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-bounded*: *finite* (*S*::*nat set*) ⟹ ∃ *k*. *S* ⊆ {..<*k*}
by (*simp add*: *finite-nat-iff-bounded*)

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*: ∀ *m*>*k*. ∃ *n*>*m*. *n* ∈ *S* ⟹ *infinite* (*S*::*nat set*)
apply (*clarsimp simp add*: *finite-nat-set-iff-bounded*)
apply (*drule-tac x*=*Suc* (*max m k*) **in** *spec*)
using *less-Suc-eq* **by** *fastforce*

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) ⟹ *R*
by *simp*

lemma *range-inj-infinite*:
inj (*f*::*nat* ⇒ *'a*) ⟹ *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

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* (*S*::*int set*) ⟷ *infinite* ((*nat o abs*) ‘ *S*)
by (*auto simp*: *transfer-nat-int-set-relations o-def image-comp dest*: *finite-image-absD*)

proposition *infinite-int-iff-unbounded-le*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| ≥ *m* ∧ *n* ∈ *S*)
apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded-le o-def image-def*)
apply (*metis abs-ge-zero nat-le-eq-zle le-nat-iff*)
done

proposition *infinite-int-iff-unbounded*: *infinite* (*S*::*int set*) ⟷ (∀ *m*. ∃ *n*. |*n*| > *m* ∧ *n* ∈ *S*)
apply (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded o-def image-def*)
apply (*metis* (*full-types*) *nat-le-iff nat-mono not-le*)
done

proposition *finite-int-iff-bounded*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {..<*k*})
using *infinite-int-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

proposition *finite-int-iff-bounded-le*: *finite* (*S*::*int set*) ⟷ (∃ *k*. *abs* ‘ *S* ⊆ {.. *k*})

using *infinite-int-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

1.1 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.

lemma *not-INFM* [*simp*]: $\neg$ (*INFM x. P x*) $\longleftrightarrow$ (*MOST x. $\neg$ P x*) **by** (*fact not-frequently*)
lemma *not-MOST* [*simp*]: $\neg$ (*MOST x. P x*) $\longleftrightarrow$ (*INFM x. $\neg$ P x*) **by** (*fact not-eventually*)

lemma *INFM-const* [*simp*]: (*INFM x::'a. P*) $\longleftrightarrow$ *P* $\wedge$ *infinite* (*UNIV::'a set*)
by (*simp add*: *frequently-const-iff*)

lemma *MOST-const* [*simp*]: (*MOST x::'a. P*) $\longleftrightarrow$ *P* $\vee$ *finite* (*UNIV::'a set*)
by (*simp add*: *eventually-const-iff*)

lemma *INFM-imp-distrib*: (*INFM x. P x* $\longrightarrow$ *Q x*) $\longleftrightarrow$ ((*MOST x. P x*) $\longrightarrow$ (*INFM x. Q x*))
by (*simp only*: *imp-conv-disj frequently-disj-iff not-eventually*)

lemma *MOST-imp-iff*: *MOST x. P x* $\Longrightarrow$ (*MOST x. P x* $\longrightarrow$ *Q x*) $\longleftrightarrow$ (*MOST x. Q x*)
by (*auto intro*: *eventually-rev-mp eventually-mono*)

lemma *INFM-conjI*: *INFM x. P x* $\Longrightarrow$ *MOST x. Q x* $\Longrightarrow$ *INFM x. P x* $\wedge$ *Q x*
by (*rule frequently-rev-mp*[*of P*]) (*auto elim*: *eventually-mono*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM x. P* (*f x*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *INFM x. P x*
using *finite-vimageI*[*of* {*x. P x*} *f*] **by** (*auto simp*: *frequently-cofinite*)

lemma *MOST-inj*: *MOST x. P x* $\Longrightarrow$ *inj f* $\Longrightarrow$ *MOST x. P* (*f x*)
using *finite-vimageI*[*of* {*x. $\neg$ P x*} *f*] **by** (*auto simp*: *eventually-cofinite*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg$ (*INFM x. x = a*)
$\neg$ (*INFM x. a = x*)
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
MOST x. x $\neq$ *a*
MOST x. a $\neq$ *x*
unfolding *eventually-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:

$(INFM\ x{::}'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
$(INFM\ x{::}'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
$(MOST\ x{::}'a.\ x = a) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
$(MOST\ x{::}'a.\ a = x) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
unfolding *eventually-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$
unfolding *eventually-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{>}m.\ P\ n)$
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded-le subset-eq not-le*[*symmetric*])

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{\geq}m.\ P\ n)$
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded subset-eq not-le*[*symmetric*])

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ P\ n)$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ P\ n)$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded-le*)

lemma *MOST-INFM*: $infinite\ (UNIV{::}'a\ set) \Longrightarrow MOST\ x{::}'a.\ P\ x \Longrightarrow INFM\ x{::}'a.\ P\ x$
by (*simp add*: *eventually-frequently*)

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
by (*simp add*: *cofinite-eq-sequentially eventually-sequentially-Suc*)

lemma
shows *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
by (*simp-all add*: *MOST-Suc-iff*)

lemma *MOST-ge-nat*: $MOST\ n{::}nat.\ m \leq n$
by (*simp add*: *cofinite-eq-sequentially eventually-ge-at-top*)

lemma *Inf-many-def*: *Inf-many* $P \longleftrightarrow infinite\ \{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *Alm-all-def*: *Alm-all* $P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ **by** *simp*
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow infinite\ \{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow finite\ \{x.\ \neg\ P\ x\}$ **by** (*fact eventually-cofinite*)
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$ **by** (*fact frequently-ex*)
lemma *ALL-MOST*: $\forall\, x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ **by** (*fact always-eventually*)

lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ **by** (*fact frequently-elim1*)
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-mono*)
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ **by** (*fact frequently-disj-iff*)
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-rev-mp*)
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ **by** (*fact eventually-conj-iff*)
lemma *MOST-conjI*: *MOST* $x.\ P\ x \Longrightarrow$ *MOST* $x.\ Q\ x \Longrightarrow$ *MOST* $x.\ P\ x \wedge Q\ x$ **by** (*fact eventually-conj*)
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow (INFM\ y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.\ INFM\ y.\ P\ x\ y)$ **by** (*fact frequently-bex-finite-distrib*)
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow (MOST\ y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.\ MOST\ y.\ P\ x\ y)$ **by** (*fact eventually-ball-finite-distrib*)
lemma *INFM-E*: *INFM* $x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$ **by** (*fact frequentlyE*)
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow$ *MOST* $x.\ P\ x$ **by** (*rule eventuallyI*)
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

1.2 Enumeration of an Infinite Set

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

Could be generalized to $enumerate'\ S\ n = (SOME\ t.\ t \in s \wedge finite\ \{s \in S.\ s < t\} \wedge card\ \{s \in S.\ s < t\} = n)$.

primrec (**in** *wellorder*) *enumerate* :: *'a set* ⇒ *nat* ⇒ *'a*
where
  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* ⟹ *enumerate S n* ∈ *S*
  apply (*induct n arbitrary*: *S*)
   apply (*fastforce intro*: *LeastI dest*!: *infinite-imp-nonempty*)
  apply *simp*
  apply (*metis DiffE infinite-remove*)
  done

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

lemma *enumerate-step*: *infinite S* ⟹ *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* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

lemma *le-enumerate*:
assumes *S*: *infinite S*
shows *n* ≤ *enumerate S n*
using *S*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have *n* ≤ *enumerate S n* **by** *simp*
also note *enumerate-mono*[*of n Suc n*, *OF* - ‹*infinite S*›]
finally show *?case* **by** *simp*
qed

lemma *enumerate-Suc″*:
fixes *S* :: *′a*::*wellorder set*
assumes *infinite S*
shows *enumerate S* (*Suc n*) = (*LEAST s*. *s* ∈ *S* ∧ *enumerate S n* < *s*)
using *assms*
proof (*induct n arbitrary*: *S*)
case *0*
then have ∀ *s* ∈ *S*. *enumerate S 0* ≤ *s*
by (*auto simp*: *enumerate.simps intro*: *Least-le*)
then show *?case*
unfolding *enumerate-Suc′ enumerate-0*[*of S* − {*enumerate S 0*}]
by (*intro arg-cong*[**where** *f* = *Least*] *ext*) *auto*
next
case (*Suc n S*)
show *?case*
using *enumerate-mono*[*OF zero-less-Suc* ‹*infinite S*›, *of n*] ‹*infinite S*›
apply (*subst* (*1 2*) *enumerate-Suc′*)
apply (*subst Suc*)
using ‹*infinite S*›
apply *simp*

```
    apply (intro arg-cong[where f = Least] ext)
    apply (auto simp: enumerate-Suc'[symmetric])
    done
qed

lemma enumerate-Ex:
  assumes S: infinite (S::nat set)
  shows s ∈ S ⟹ ∃ n. enumerate S n = s
proof (induct s rule: less-induct)
  case (less s)
  show ?case
  proof cases
    let ?y = Max {s'∈S. s' < s}
    assume ∃ y∈S. y < s
    then have y: ⋀x. ?y < x ⟷ (∀ s'∈S. s' < s ⟶ s' < x)
      by (subst Max-less-iff) auto
    then have y-in: ?y ∈ {s'∈S. s' < s}
      by (intro Max-in) auto
    with less.hyps[of ?y] obtain n where enumerate S n = ?y
      by auto
    with S have enumerate S (Suc n) = s
      by (auto simp: y less enumerate-Suc'' intro!: Least-equality)
    then show ?case by auto
  next
    assume *: ¬ (∃ y∈S. y < s)
    then have ∀ t∈S. s ≤ t by auto
    with ‹s ∈ S› show ?thesis
      by (auto intro!: exI[of - 0] Least-equality simp: enumerate-0)
  qed
qed

lemma bij-enumerate:
  fixes S :: nat set
  assumes S: infinite S
  shows bij-betw (enumerate S) UNIV S
proof −
  have ⋀n m. n ≠ m ⟹ enumerate S n ≠ enumerate S m
    using enumerate-mono[OF - ‹infinite S›] by (auto simp: neq-iff)
  then have inj (enumerate S)
    by (auto simp: inj-on-def)
  moreover have ∀ s ∈ S. ∃ i. enumerate S i = s
    using enumerate-Ex[OF S] by auto
  moreover note ‹infinite S›
  ultimately show ?thesis
    unfolding bij-betw-def by (auto intro: enumerate-in-set)
qed
```

A pair of weird and wonderful lemmas from HOL Light

```
lemma finite-transitivity-chain:
```

assumes *finite A*
and *R*: $\bigwedge x.\ \sim R\ x\ x\ \bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$
and *A*: $\bigwedge x.\ x \in A \Longrightarrow \exists\, y.\ y \in A \wedge R\ x\ y$
shows $A = \{\}$
using ‹*finite A*› *A*
proof (*induction A*)
case (*insert a A*)
with *R* **show** *?case*
by (*metis empty-iff insert-iff*)
qed *simp*

corollary *Union-maximal-sets*:
assumes *finite* $\mathcal{F}$
shows $\bigcup\{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} = \bigcup \mathcal{F}$
(**is** *?lhs* = *?rhs*)
proof
show *?rhs* $\subseteq$ *?lhs*
proof (*rule Union-subsetI*)
fix *S*
assume $S \in \mathcal{F}$
have $\{T \in \mathcal{F}.\ S \subseteq T\} = \{\}$ **if** $\sim (\exists\, y.\ y \in \{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} \wedge S \subseteq y)$
apply (*rule finite-transitivity-chain* [*of* - $\lambda T\ U.\ S \subseteq T \wedge T \subset U$])
using *assms that* **apply** *auto*
by (*blast intro*: *dual-order.trans psubset-imp-subset*)
then show $\exists\, y.\ y \in \{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} \wedge S \subseteq y$
using ‹$S \in \mathcal{F}$› **by** *blast*
qed
qed *force*

end

2 Ramsey's Theorem

theory *Ramsey*
imports *Main* ~~/*src*/*HOL*/*Library*/*Infinite-Set*
begin

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

2.1 Library lemmas

lemma *infinite-inj-infinite-image*: *infinite Z* ==> *inj-on f Z* ==> *infinite* (*f* ‘ *Z*)
apply(*rule ccontr*)
apply(*simp*)
apply(*force dest*: *finite-imageD*)
done

lemma *infinite-dom-finite-rng*: [| *infinite A*; *finite* (*f* ‘ *A*) |] ==> *?* *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* ==> *?* *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

—
primrec *choice* :: ('*a* => *bool*) => ('*a* => '*a* => *bool*) => *nat* => '*a* **where**
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*) **apply**(*blast*)
apply(*subgoal-tac ? yy. yy* : *YY*) **prefer** *2* **apply**(*rule infinite-mem*) **apply**(*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*) **apply**(*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) −−> 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−(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(*rename-tac b*) **apply**(*case-tac b rule*: *prod.exhaust*) **apply**(*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*) **apply**(*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