Ramsey's Theorem

Tom Ridge

October 11, 2017

Abstract

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

Contents

1 Ramsey's Theorem

theory *Ramsey*
imports *Main HOL−Library.Infinite-Set*
begin

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

1.1 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

1.2 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

1.3 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

1.4 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