Compactness Theorem for Propositional Logic and Combinatorial Applications

Fabián Fernando Serrano Suárez[†], Thaynara Arielly de Lima[⋆], Mauricio Ayala-Rincón[‡]

[†] Universidad Nacional de Colombia - Sede Manizales, Colombia
[⋆]Universidade Federal de Goiás, Goiânia, Brazil
[‡]Universidade de Brasília, Brasília D.F., Brazil

August 30, 2024

Abstract

This theory formalises the compactness theorem for propositional logic based on the model existence theorem approach. It also presents applications of the compactness theorem to formalize combinatorial theorems over countable structures: the de Bruijn-Erdős Graph coloring theorem for countable graphs, König's Lemma, and set- and graph-theoretical versions of Hall's Theorem for countable families of sets and graphs.

Contents

theory *Background-on-graphs*

imports *Main*

begin

1 Special Graph Theoretical Notions

This theory provides a background on specialized graph notions and properties. We follow the approach by L. Noschinski available in the AFPs. Since not all elements of Noschinski theory are required, we prefer not to import it.

The proof are desiccated in several steps since the focus is clarity instead proof automation.

record ('a,'b) *pre-digraph* =
verts :: 'a *set*
arcs :: 'b *set*
tail :: 'b ⇒ 'a
head :: 'b ⇒ 'a

definition *tails*:: ('a,'b) *pre-digraph* ⇒ 'a *set* **where**
tails G ≡ {*tail G e* |*e*. *e* ∈ *arcs G* }

definition *tails-set* :: ('a,'b) *pre-digraph* ⇒ 'b *set* ⇒ 'a *set* **where**
tails-set G E ≡ { *tail G e* |*e*. *e* ∈ *E* ∧ *E* ⊆ *arcs G* }

definition *heads*:: ('a,'b) *pre-digraph* ⇒ 'a *set* **where**
heads G ≡ { *head G e* |*e*. *e* ∈ *arcs G* }

definition *heads-set*:: ('a,'b) *pre-digraph* ⇒ 'b *set* ⇒ 'a *set* **where**
heads-set G E ≡ { *head G e* |*e*. *e* ∈ *E* ∧ *E* ⊆ *arcs G* }

definition *neighbour*:: ('a,'b) *pre-digraph* ⇒ 'a ⇒ 'a ⇒ *bool* **where**
neighbour G v u ≡
∃ *e*. *e*∈ (*arcs G*) ∧ ((*head G e* = *v* ∧ *tail G e* = *u*) ∨
(*head G e* = *u* ∧ *tail G e* = *v*))

definition *neighbourhood*:: ('a,'b) *pre-digraph* ⇒ 'a ⇒ 'a *set* **where**
neighbourhood G v ≡ {*u* |*u*. *neighbour G u v*}

definition *bipartite-digraph*:: ('a,'b) *pre-digraph* ⇒ 'a *set* ⇒ 'a *set* ⇒ *bool* **where**
bipartite-digraph G X Y ≡

$(X \cup Y = (verts\ G)) \wedge\ X \cap Y = \{\} \wedge$
$(\forall e \in (arcs\ G).(tail\ G\ e) \in X \longleftrightarrow (head\ G\ e) \in Y)$

definition *dir-bipartite-digraph*:: $('a,'b)$ *pre-digraph* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
where
dir-bipartite-digraph $G\ X\ Y \equiv (bipartite\text{-}digraph\ G\ X\ Y) \wedge$
$((tails\ G = X) \wedge (\forall\ e1 \in arcs\ G.\ \forall\ e2 \in arcs\ G.\ e1 = e2 \longleftrightarrow head\ G\ e1 = head\ G\ e2 \wedge tail\ G\ e1 = tail\ G\ e2))$

definition *K-E-bipartite-digraph*:: $('a,'b)$ *pre-digraph* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
where
K-E-bipartite-digraph $G\ X\ Y \equiv$
$(dir\text{-}bipartite\text{-}digraph\ G\ X\ Y) \wedge (\forall x{\in}X.\ finite\ (neighbourhood\ G\ x))$

definition *dirBD-matching*:: $('a,'b)$ *pre-digraph* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ *bool*
where
dirBD-matching $G\ X\ Y\ E \equiv$
$dir\text{-}bipartite\text{-}digraph\ G\ X\ Y \wedge (E \subseteq (arcs\ G)) \wedge$
$(\forall\ e1{\in}E.\ (\forall\ e2{\in}\ E.\ e1 \neq e2 \longrightarrow$
$((head\ G\ e1) \neq (head\ G\ e2)) \wedge$
$((tail\ G\ e1) \neq (tail\ G\ e2))))$

lemma *tail-head*:
assumes *dir-bipartite-digraph* $G\ X\ Y$ **and** $e \in arcs\ G$
shows $(tail\ G\ e) \in X \wedge (head\ G\ e) \in Y$
using *assms*
by (*unfold dir-bipartite-digraph-def*, *unfold bipartite-digraph-def*, *unfold tails-def*, *auto*)

lemma *tail-head1*:
assumes *dirBD-matching* $G\ X\ Y\ E$ **and** $e \in E$
shows $(tail\ G\ e) \in X \wedge (head\ G\ e) \in Y$
using *assms tail-head*[*of* $G\ X\ Y\ e$] **by**(*unfold dirBD-matching-def*, *auto*)

lemma *dirBD-matching-tail-edge-unicity*:
dirBD-matching $G\ X\ Y\ E \longrightarrow$
$(\forall e1 \in E.\ (\forall\ e2{\in}\ E.\ (tail\ G\ e1 = tail\ G\ e2) \longrightarrow e1 = e2))$

proof
assume *dirBD-matching* $G\ X\ Y\ E$
thus $\forall e1{\in}E.\ \forall e2{\in}E.\ tail\ G\ e1 = tail\ G\ e2 \longrightarrow e1 = e2$
by (*unfold dirBD-matching-def*, *auto*)
qed

lemma *dirBD-matching-head-edge-unicity*:
dirBD-matching $G\ X\ Y\ E \longrightarrow$

(∀ e1 ∈ E. (∀ e2∈ E. (head G e1 = head G e2) ⟶ e1 = e2))

proof
assume *dirBD-matching G X Y E*
thus ∀ *e1*∈*E*. ∀ *e2*∈*E*. *head G e1* = *head G e2* ⟶ *e1* = *e2*
by(*unfold dirBD-matching-def*, *auto*)
qed

definition *dirBD-perfect-matching*::
('a,'b) *pre-digraph* ⇒ 'a *set* ⇒ 'a *set* ⇒ 'b *set* ⇒ *bool*
where
dirBD-perfect-matching G X Y E ≡
dirBD-matching G X Y E ∧ (*tails-set G E* = *X*)

lemma *Tail-covering-edge-in-Pef-matching*:
∀ *x*∈*X*. *dirBD-perfect-matching G X Y E* ⟶ (∃ *e* ∈ *E*. *tail G e* = *x*)
proof
fix *x*
assume *Hip1*: *x* ∈ *X*
show *dirBD-perfect-matching G X Y E* ⟶ (∃ *e*∈*E*. *tail G e* = *x*)
proof
assume *dirBD-perfect-matching G X Y E*
hence *x* ∈ *tails-set G E* **using** *Hip1*
by (*unfold dirBD-perfect-matching-def*, *auto*)
thus ∃ *e*∈*E*. *tail G e* = *x* **by** (*unfold tails-set-def*, *auto*)
qed
qed

lemma *Edge-unicity-in-dirBD-P-matching*:
∀ *x*∈*X*. *dirBD-perfect-matching G X Y E* ⟶ (∃!*e* ∈ *E*. *tail G e* = *x*)

proof
fix *x*
assume *Hip1*: *x* ∈ *X*
show *dirBD-perfect-matching G X Y E* ⟶ (∃!*e* ∈ *E*. *tail G e* = *x*)
proof
assume *Hip2*: *dirBD-perfect-matching G X Y E*
then obtain ∃ *e*. *e* ∈ *E* ∧ *tail G e* = *x*
using *Hip1 Tail-covering-edge-in-Pef-matching*[*of X G Y E*] **by** *auto*
then obtain *e* **where** *e*: *e* ∈ *E* ∧ *tail G e* = *x* **by** *auto*
hence *a*: *e* ∈ *E* ∧ *tail G e* = *x* **by** *auto*
show ∃!*e*. *e* ∈ *E* ∧ *tail G e* = *x*
proof
show *e* ∈ *E* ∧ *tail G e* = *x* **using** *a* **by** *auto*
next
fix *e1*
assume *Hip3*: *e1* ∈ *E* ∧ *tail G e1* = *x*
hence *tail G e* = *tail G e1* ∧ *e* ∈ *E* ∧ *e1* ∈ *E* **using** *a* **by** *auto*

```
      moreover
      have dirBD-matching G X Y E
        using Hip2  by(unfold dirBD-perfect-matching-def, auto)
      ultimately
      show e1 = e
        using Hip2 dirBD-matching-tail-edge-unicity[of G X Y E]
        by auto
    qed
  qed
qed

definition E-head :: ('a,'b) pre-digraph ⇒ 'b set  ⇒ ('a  ⇒ 'a)
  where
  E-head G E = (λx. (THE y. ∃  e. e ∈ E ∧ tail G e = x ∧  head G e = y))

lemma unicity-E-head1:
   assumes dirBD-matching G X Y E ∧ e ∈ E ∧ tail G e = x ∧ head G e = y
   shows (∀ z.(∃  e. e ∈ E ∧ tail G e = x ∧ head G e = z)⟶ z = y)
using assms dirBD-matching-tail-edge-unicity by blast

lemma unicity-E-head2:
   assumes dirBD-matching G X Y E ∧ e ∈ E ∧ tail G e = x ∧ head G e = y
   shows  (THE a. ∃  e. e ∈ E ∧ tail G e = x ∧ head G e = a) = y
using assms dirBD-matching-tail-edge-unicity by blast

lemma  unicity-E-head:
  assumes dirBD-matching G X Y E ∧ e ∈ E ∧ tail G e = x ∧ head G e = y
  shows (E-head G E) x = y
  using assms unicity-E-head2[of G X Y E e x y] by(unfold E-head-def, auto)

lemma E-head-image :
  dirBD-perfect-matching G X Y E ⟶
   (e ∈ E ∧ tail G e = x ⟶ (E-head G E) x = head G e)

proof
  assume dirBD-perfect-matching G X Y E
  thus e ∈ E ∧ tail G e = x ⟶ (E-head G E) x = head G e
    using dirBD-matching-tail-edge-unicity [of G X Y E]
    by (unfold E-head-def, unfold dirBD-perfect-matching-def, blast)
qed

lemma E-head-in-neighbourhood:
  dirBD-matching G X Y E ⟶ e ∈ E ⟶ tail G e = x ⟶
   (E-head G E) x ∈ neighbourhood G x

proof (rule impI)+
  assume
  dir-BDm: dirBD-matching G X Y E and ed: e ∈ E and hd: tail G e = x
  show E-head G E x ∈ neighbourhood G x
```

```
  proof-
    have (∃ y. y = head G e) using hd by auto
    then obtain y where y: y = head G e by auto
    hence (E-head G E) x = y
      using dir-BDm ed hd unicity-E-head[of G X Y E e x y]
      by auto
    moreover
    have e ∈ (arcs G) using dir-BDm ed by(unfold dirBD-matching-def, auto)
    hence neighbour G y x using ed hd y by(unfold neighbour-def, auto)
    ultimately
    show ?thesis using hd ed by(unfold neighbourhood-def, auto)
  qed
qed

lemma dirBD-matching-inj-on:
   dirBD-perfect-matching G X Y E ⟶ inj-on (E-head G E) X

proof(rule impI)
  assume dirBD-pm : dirBD-perfect-matching G X Y E
  show inj-on (E-head G E) X
  proof(unfold inj-on-def)
    show ∀ x∈X. ∀ y∈X. E-head G E x = E-head G E y ⟶ x = y
    proof
      fix x
      assume 1: x∈ X
      show ∀ y∈X. E-head G E x = E-head G E y ⟶ x = y
      proof
        fix y
        assume 2: y∈ X
        show E-head G E x = E-head G E y ⟶ x = y
        proof(rule impI)
          assume same-eheads: E-head G E x = E-head G E y
          show x=y
          proof-
            have hex: (∃!e ∈ E. tail G e = x)
              using dirBD-pm 1 Edge-unicity-in-dirBD-P-matching[of X G Y E]
              by auto
            then obtain ex where hex1: ex ∈ E ∧ tail G ex = x by auto
            have ey: (∃!e ∈ E. tail G e = y)
              using dirBD-pm 2 Edge-unicity-in-dirBD-P-matching[of X G Y E]
              by auto
            then obtain ey where hey1: ey ∈ E ∧ tail G ey = y by auto
            have ettx: E-head G E x = head G ex
              using dirBD-pm hex1 E-head-image[of G X Y E ex x] by auto
            have etty: E-head G E y = head G ey
              using dirBD-pm hey1 E-head-image[of G X Y E ey y] by auto
            have same-heads: head G ex = head G ey
              using same-eheads ettx etty by auto
            hence same-edges: ex = ey
```

```
            using dirBD-pm 1 2 hex1 hey1
                  dirBD-matching-head-edge-unicity[of G X Y E]
           by(unfold dirBD-perfect-matching-def,unfold dirBD-matching-def, blast)
           thus ?thesis using same-edges hex1 hey1 by auto
          qed
        qed
      qed
    qed
  qed
qed


end


datatype 'b formula =
    FF
  | TT
  | atom 'b
  | Negation 'b formula                    (¬.(-) [110] 110)
  | Conjunction 'b formula 'b formula      (infixl ∧. 109)
  | Disjunction 'b formula 'b formula      (infixl ∨. 108)
  | Implication 'b formula 'b formula      (infixl →. 100)


lemma (¬.¬. Atom P →. Atom Q →. Atom R) =
       (((¬. (¬. Atom P)) →. Atom Q) →. Atom R)
by simp


datatype v-truth = Ttrue | Ffalse


definition v-negation :: v-truth ⇒ v-truth where
 v-negation x ≡ (if x = Ttrue then Ffalse else Ttrue)

definition v-conjunction :: v-truth ⇒ v-truth ⇒ v-truth where
 v-conjunction x y ≡ (if x = Ffalse then Ffalse else y)

definition v-disjunction :: v-truth ⇒ v-truth ⇒ v-truth where
 v-disjunction x y ≡ (if x = Ttrue then Ttrue else y)

definition v-implication :: v-truth ⇒ v-truth ⇒ v-truth where
 v-implication x y ≡ (if x = Ffalse then Ttrue else y)


primrec t-v-evaluation :: ('b ⇒ v-truth) ⇒ 'b formula ⇒ v-truth
where
   t-v-evaluation I FF = Ffalse
```

| *t-v-evaluation I TT = Ttrue*
| *t-v-evaluation I (atom p) = I p*
| *t-v-evaluation I (¬. F) = (v-negation (t-v-evaluation I F))*
| *t-v-evaluation I (F ∧. G) = (v-conjunction (t-v-evaluation I F) (t-v-evaluation I G))*
| *t-v-evaluation I (F ∨. G) = (v-disjunction (t-v-evaluation I F) (t-v-evaluation I G))*
| *t-v-evaluation I (F →. G) = (v-implication (t-v-evaluation I F) (t-v-evaluation I G))*

lemma *Bivaluation*:
shows *t-v-evaluation I F = Ttrue* ∨ *t-v-evaluation I F = Ffalse*

lemma *NegationValues1*:
assumes *t-v-evaluation I (¬.F) = Ffalse*
shows *t-v-evaluation I F = Ttrue*

lemma *NegationValues2*:
assumes *t-v-evaluation I (¬.F) = Ttrue*
shows *t-v-evaluation I F = Ffalse*
lemma *non-Ttrue*:
assumes *t-v-evaluation I F ≠ Ttrue* **shows** *t-v-evaluation I F = Ffalse*

lemma *ConjunctionValues*:
assumes *t-v-evaluation I (F ∧. G) = Ttrue*
shows *t-v-evaluation I F = Ttrue* ∧ *t-v-evaluation I G = Ttrue*

lemma *DisjunctionValues*:
assumes *t-v-evaluation I (F ∨. G) = Ttrue*
shows *t-v-evaluation I F = Ttrue* ∨ *t-v-evaluation I G = Ttrue*

lemma *ImplicationValues*:
assumes *t-v-evaluation I (F →. G) = Ttrue*
shows *t-v-evaluation I F = Ttrue* ⟶ *t-v-evaluation I G = Ttrue*

definition *model* :: (*'b* ⇒ *v-truth*) ⇒ *'b formula set* ⇒ *bool* (- *model* - [*80*,*80*] *80*)
where
I model S ≡ (∀ *F* ∈ *S*. *t-v-evaluation I F = Ttrue*)

definition *satisfiable* :: *'b formula set* ⇒ *bool* **where**
satisfiable S ≡ (∃ *v*. *v model S*)

definition *consequence* :: *'b formula set* ⇒ *'b formula* ⇒ *bool* (- ⊨ - [*80*,*80*] *80*)
where
S ⊨ *F* ≡ (∀ *I*. *I model S* ⟶ *t-v-evaluation I F = Ttrue*)

theorem *EquiConsSat*:
shows $S \models F = (\neg\ satisfiable\ (S \cup \{\neg.\ F\}))$

definition *tautology* :: *'b formula* ⇒ *bool* **where**
tautology F ≡ $(\forall I.\ ((t\text{-}v\text{-}evaluation\ I\ F) = Ttrue))$

lemma *tautology* $(F\ \rightarrow.\ (G \rightarrow.\ F))$
proof −
have $\forall I.\ t\text{-}v\text{-}evaluation\ I\ (F \rightarrow.\ (G \rightarrow.\ F)) = Ttrue$
proof
fix *I*
show $t\text{-}v\text{-}evaluation\ I\ (F \rightarrow.\ (G \rightarrow.\ F)) = Ttrue$
proof (*cases t-v-evaluation I F*)

Caso 1:

{ assume *t-v-evaluation I F* = *Ttrue*
thus *?thesis* **by** (*simp add*: *v-implication-def*) **}**
next

Caso 2:

{ assume *t-v-evaluation I F* = *Ffalse*
thus *?thesis* **by**(*simp add*: *v-implication-def*) **}**
qed
qed
thus *?thesis* **by** (*simp add*: *tautology-def*)
qed

theorem *CNS-tautology*: *tautology F* = $(\{\} \models F)$

theorem *TautSatis*:
shows *tautology* $(F \rightarrow.\ G) = (\neg\ satisfiable\{F,\ \neg.G\})$

fun *FormulaLiteral* :: *'b formula* ⇒ *bool* **where**
FormulaLiteral FF = *True*
| *FormulaLiteral* (¬. *FF*) = *True*
| *FormulaLiteral TT* = *True*
| *FormulaLiteral* (¬. *TT*) = *True*
| *FormulaLiteral* (*atom P*) = *True*
| *FormulaLiteral* (¬.(*atom P*)) = *True*
| *FormulaLiteral F* = *False*

```
fun FormulaNoNo :: 'b formula ⇒ bool where
  FormulaNoNo (¬. (¬. F)) = True
| FormulaNoNo F = False


fun FormulaAlfa :: 'b formula ⇒ bool where
  FormulaAlfa (F ∧. G) = True
| FormulaAlfa (¬. (F ∨. G)) = True
| FormulaAlfa (¬. (F →. G)) =  True
| FormulaAlfa F = False


fun FormulaBeta :: 'b formula ⇒ bool where
  FormulaBeta (F ∨. G) = True
| FormulaBeta (¬. (F ∧. G)) = True
| FormulaBeta (F →. G) =  True
| FormulaBeta F = False

lemma noLiteralNoNo:
  assumes FormulaLiteral formula
  shows ¬(FormulaNoNo formula)
using assms Literal NoNo
by (induct formula rule: FormulaLiteral.induct, auto)


lemma noLiteralAlfa:
  assumes FormulaLiteral formula
  shows ¬(FormulaAlfa formula)
using assms Literal Alfa
by (induct formula rule: FormulaLiteral.induct, auto)


lemma noLiteralBeta:
  assumes FormulaLiteral formula
  shows ¬(FormulaBeta formula)
using assms Literal Beta
by (induct formula rule: FormulaLiteral.induct, auto)


lemma noAlfaNoNo:
  assumes FormulaAlfa formula
  shows ¬(FormulaNoNo formula)
using assms Alfa NoNo
by (induct formula rule: FormulaAlfa.induct, auto)
```

lemma *noBetaNoNo*:
  assumes *FormulaBeta formula*
  shows ¬(*FormulaNoNo formula*)
using *assms Beta NoNo*
by (*induct formula rule*: *FormulaBeta.induct*, *auto*)

lemma *noAlfaBeta*:
  assumes *FormulaAlfa formula*
  shows ¬(*FormulaBeta formula*)
using *assms Alfa Beta*
by (*induct formula rule*: *FormulaAlfa.induct*, *auto*)

lemma *UniformNotation*:
 FormulaLiteral F ∨ *FormulaNoNo F* ∨ *FormulaAlfa F* ∨ *FormulaBeta F*

datatype *typeUniformNotation* = *Literal* | *NoNo* | *Alfa*| *Beta*

fun *typeFormula* :: ′*b formula* ⇒ *typeUniformNotation* **where**
typeFormula F =
  (*if FormulaBeta F then Beta*
   else if FormulaNoNo F then NoNo
   else if FormulaAlfa F then Alfa
   else Literal)

fun *componentes* :: ′*b formula* ⇒ ′*b formula list* **where**
  componentes (¬. (¬. *G*)) = [*G*]
| *componentes* (*G* ∧. *H*) = [*G*, *H*]
| *componentes* (¬. (*G* ∨. *H*)) = [¬. *G*, ¬. *H*]
| *componentes* (¬. (*G* →. *H*)) = [*G*, ¬. *H*]
| *componentes* (*G* ∨. *H*) = [*G*, *H*]
| *componentes* (¬. (*G* ∧. *H*)) = [¬. *G*, ¬. *H*]
| *componentes* (*G* →. *H*) = [¬.*G*, *H*]

definition *Comp1* :: ′*b formula* ⇒ ′*b formula* **where**
  Comp1 F = *hd* (*componentes F*)

definition *Comp2* :: ′*b formula* ⇒ ′*b formula* **where**
  Comp2 F = *hd* (*tl* (*componentes F*))

primrec *t-v-evaluationDisyuncionG* :: (′*b* ⇒ *v-truth*) ⇒ (′*b formula list*) ⇒ *v-truth*
where

t-v-evaluationDisyuncionG I [] = *Ffalse*
| *t-v-evaluationDisyuncionG I* (*F#Fs*) = (*if t-v-evaluation I F* = *Ttrue then Ttrue else t-v-evaluationDisyuncionG I Fs*)

primrec *t-v-evaluationConjuncionG* :: (*'b* ⇒ *v-truth*) ⇒ (*'b formula list*) *list* ⇒ *v-truth* **where**
t-v-evaluationConjuncionG I [] = *Ttrue*
| *t-v-evaluationConjuncionG I* (*D#Ds*) =
(*if t-v-evaluationDisyuncionG I D* = *Ffalse then Ffalse else t-v-evaluationConjuncionG I Ds*)

definition *equivalentesG* :: (*'b formula list*) *list* ⇒ (*'b formula list*) *list* ⇒ *bool* **where**
equivalentesG C1 C2 ≡ (∀ *I*. ((*t-v-evaluationConjuncionG I C1*) = (*t-v-evaluationConjuncionG I C2*)))

lemma *EquiNoNo*:
assumes *typeFormula F* = *NoNo*
shows *equivalentesG* [[*F*]] [[*Comp1 F*]]

lemma *EquiAlfa*:
assumes *typeFormula F* = *Alfa*
shows *equivalentesG* [[*F*]] [[*Comp1 F*],[*Comp2 F*]]

lemma *EquiBeta*:
assumes *typeFormula F* = *Beta*
shows *equivalentesG* [[*F*]] [[*Comp1 F*, *Comp2 F*]]

lemma *EquivNoNoComp*:
assumes *typeFormula F* = *NoNo*
shows *equivalent F* (*Comp1 F*)

lemma *EquivAlfaComp*:
assumes *typeFormula F* = *Alfa*
shows *equivalent F* (*Comp1 F* ∧. *Comp2 F*)

lemma *EquivBetaComp*:
assumes *typeFormula F* = *Beta*
shows *equivalent F* (*Comp1 F* ∨. *Comp2 F*)

definition *consistenceP* :: *'b formula set set* ⇒ *bool* **where**

consistenceP $\mathcal{C}$ =
(∀ *S*. *S* ∈ $\mathcal{C}$ ⟶ (∀ *P*. ¬ (*atom P* ∈ *S* ∧ (¬.*atom P*) ∈ *S*)) ∧
FF ∉ *S* ∧ (¬.*TT*) ∉ *S* ∧
(∀ *F*. (¬.¬.*F*) ∈ *S* ⟶ *S* ∪ {*F*} ∈ $\mathcal{C}$) ∧
(∀ *F*. ((*FormulaAlfa F*) ∧ *F*∈*S*) ⟶ (*S*∪{*Comp1 F*, *Comp2 F*}) ∈ $\mathcal{C}$) ∧
(∀ *F*. ((*FormulaBeta F*) ∧ *F*∈*S*) ⟶ (*S*∪{*Comp1 F*}∈$\mathcal{C}$) ∨ (*S*∪{*Comp2 F*}∈$\mathcal{C}$)))

definition *subset-closed* :: *'a set set* ⇒ *bool* **where**
subset-closed $\mathcal{C}$ = (∀ *S* ∈ $\mathcal{C}$. ∀ *S'*. *S'* ⊆ *S* ⟶ *S'* ∈ $\mathcal{C}$)

definition *closure-subset* :: *'a set set* ⇒ *'a set set* (-[*1000*] *1000*) **where**
$\mathcal{C}$ = {*S*. ∃ *S'* ∈ $\mathcal{C}$. *S* ⊆ *S'*}

lemma *closed-subset*: $\mathcal{C}$ ⊆ $\mathcal{C}$
proof −
{ **fix** *S*
assume *S* ∈ $\mathcal{C}$
moreover
have *S* ⊆ *S* **by** *simp*
ultimately
have *S* ∈ $\mathcal{C}$
by (*unfold closure-subset-def*, *auto*) }
thus *?thesis* **by** *auto*
qed

lemma *closed-closed*: *subset-closed* ($\mathcal{C}$)
proof −
{ **fix** *S T*
assume *S* ∈ $\mathcal{C}$ **and** *T* ⊆ *S*
obtain *S1* **where** *S1* ∈ $\mathcal{C}$ **and** *S* ⊆ *S1* **using** ‹*S* ∈ $\mathcal{C}$›
by (*unfold closure-subset-def*, *auto*)
have *T* ⊆ *S1* **using** ‹*T* ⊆ *S*› **and** ‹*S* ⊆ *S1*› **by** *simp*
hence *T* ∈ $\mathcal{C}$ **using** ‹*S1* ∈ $\mathcal{C}$›
by (*unfold closure-subset-def*, *auto*)}
thus *?thesis* **by** (*unfold subset-closed-def*, *auto*)
qed

lemma *cond-consistP1*:
assumes *consistenceP* $\mathcal{C}$ **and** *T* ∈ $\mathcal{C}$ **and** *S* ⊆ *T*
shows (∀ *P*. ¬(*atom P* ∈ *S* ∧ (¬.*atom P*) ∈ *S*))
lemma *cond-consistP2*:
assumes *consistenceP* $\mathcal{C}$ **and** *T* ∈ $\mathcal{C}$ **and** *S* ⊆ *T*
shows *FF* ∉ *S* ∧ (¬.*TT*)∉ *S*
lemma *cond-consistP3*:

assumes *consistenceP* $\mathcal{C}$ **and** $T \in \mathcal{C}$ **and** $S \subseteq T$
shows $\forall F. (\neg.\neg.F) \in S \longrightarrow S \cup \{F\} \in \mathcal{C}$
proof(*rule allI*)
lemma *cond-consistP4*:
assumes *consistenceP* $\mathcal{C}$ **and** $T \in \mathcal{C}$ **and** $S \subseteq T$
shows $\forall F. ((FormulaAlfa\ F) \wedge F \in S) \longrightarrow (S \cup \{Comp1\ F, Comp2\ F\}) \in \mathcal{C}$

lemma *cond-consistP5*:
assumes *consistenceP* $\mathcal{C}$ **and** $T \in \mathcal{C}$ **and** $S \subseteq T$
shows $(\forall F. ((FormulaBeta\ F) \wedge F \in S) \longrightarrow$
$(S \cup \{Comp1\ F\} \in \mathcal{C}) \vee (S \cup \{Comp2\ F\} \in \mathcal{C}))$
theorem *closed-consistenceP*:
assumes *hip1*: *consistenceP* $\mathcal{C}$
shows *consistenceP* $(\mathcal{C})$
proof –
{ fix S
assume $S \in \mathcal{C}$
hence $\exists T\in\mathcal{C}. S \subseteq T$ **by**(*simp add*: *closure-subset-def*)
then obtain T **where** *hip2*: $T \in \mathcal{C}$ **and** *hip3*: $S \subseteq T$ **by** *auto*
have $(\forall P. \neg (atom\ P \in S \wedge (\neg.atom\ P) \in S)) \wedge$
$FF \notin S \wedge (\neg.TT) \notin S \wedge$
$(\forall F. (\neg.\neg.F) \in S \longrightarrow S \cup \{F\} \in \mathcal{C}) \wedge$
$(\forall F. ((FormulaAlfa\ F) \wedge F \in S) \longrightarrow$
$(S \cup \{Comp1\ F, Comp2\ F\}) \in \mathcal{C}) \wedge$
$(\forall F. ((FormulaBeta\ F) \wedge F \in S) \longrightarrow$
$(S \cup \{Comp1\ F\} \in \mathcal{C}) \vee (S \cup \{Comp2\ F\} \in \mathcal{C}))$
using
cond-consistP1[*OF hip1 hip2 hip3*] *cond-consistP2*[*OF hip1 hip2 hip3*]
cond-consistP3[*OF hip1 hip2 hip3*] *cond-consistP4*[*OF hip1 hip2 hip3*]
cond-consistP5[*OF hip1 hip2 hip3*]
by *blast***}**
thus *?thesis* **by** (*simp add*: *consistenceP-def*)
qed

2 Finiteness Character Property

This theory formalises the theorem that states that subset closed propositional consistency properties can be extended to satisfy the finite character property.

The proof is by induction on the structure of propositional formulas based on the analysis of cases for the possible different types of formula in the sets of the collection of sets that hold the propositional consistency property.

definition *finite-character* :: *'a set set* ⇒ *bool* **where**

```
finite-character 𝒞 = (∀ S. S ∈ 𝒞 = (∀ S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ 𝒞))
```

```
theorem finite-character-closed:
  assumes finite-character 𝒞
  shows subset-closed 𝒞
proof −
  { fix S T
    assume S ∈ 𝒞 and  T ⊆ S
    have T ∈ 𝒞 using finite-character-def
    proof −
      { fix U
        assume finite U and U ⊆ T
        have U ∈ 𝒞
        proof −
          have U ⊆ S using ‹U ⊆ T› and ‹T ⊆ S› by simp
          thus U ∈ 𝒞 using ‹S ∈ 𝒞› and ‹finite U› and assms
            by (unfold finite-character-def) blast
        qed}
      thus ?thesis using assms by( unfold finite-character-def) blast
    qed }
  thus ?thesis  by(unfold  subset-closed-def) blast
qed
```

```
definition closure-cfinite :: 'a set set ⇒ 'a set set (- [1000] 999) where
  𝒞 = {S. ∀ S'. S' ⊆ S ⟶ finite S' ⟶ S' ∈ 𝒞}
```

```
lemma finite-character-subset:
  assumes subset-closed 𝒞
  shows 𝒞 ⊆ 𝒞
proof −
  { fix S
    assume S ∈ 𝒞
    have S ∈ 𝒞
    proof −
      { fix S'
        assume S' ⊆ S and finite S'
        hence S' ∈ 𝒞 using  ‹subset-closed 𝒞› and ‹S ∈ 𝒞›
          by (simp add: subset-closed-def)}
      thus ?thesis by (simp add: closure-cfinite-def)
    qed}
  thus ?thesis by auto
qed
```

lemma *finite-character*: *finite-character* ($\mathcal{C}$)
proof (*unfold finite-character-def*)
  show $\forall S. (S \in \mathcal{C}) = (\forall S'. \textit{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in \mathcal{C})$
  proof
    fix S
    { **assume** $S \in \mathcal{C}$
      hence $\forall S'. \textit{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in \mathcal{C}$
        by(*simp add*: *closure-cfinite-def*)}
    moreover
    { **assume** $\forall S'. \textit{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in \mathcal{C}$
      hence $S \in \mathcal{C}$ **by**(*simp add*: *closure-cfinite-def*)}
    ultimately
    show $(S \in \mathcal{C}) = (\forall S'. \textit{finite } S' \longrightarrow S' \subseteq S \longrightarrow S' \in \mathcal{C})$
      by *blast*
  qed
qed

lemma *cond-characterP1*:
  assumes *consistenceP* $\mathcal{C}$
  and *subset-closed* $\mathcal{C}$
  and *hip*: $\forall S' \subseteq S. \textit{finite } S' \longrightarrow S' \in \mathcal{C}$
  shows $(\forall P. \neg(\textit{atom } P \in S \wedge (\neg.\textit{atom } P) \in S))$
lemma *cond-characterP2*:
  assumes *consistenceP* $\mathcal{C}$
  and *subset-closed* $\mathcal{C}$
  and *hip*: $\forall S' \subseteq S. \textit{finite } S' \longrightarrow S' \in \mathcal{C}$
  shows $FF \notin S \wedge (\neg.TT) \notin S$

lemma *cond-characterP3*:
  assumes *consistenceP* $\mathcal{C}$
  and *subset-closed* $\mathcal{C}$
  and *hip*: $\forall S' \subseteq S. \textit{finite } S' \longrightarrow S' \in \mathcal{C}$
  shows $\forall F. (\neg.\neg.F) \in S \longrightarrow S \cup \{F\} \in \mathcal{C}$
lemma *cond-characterP4*:
  assumes *consistenceP* $\mathcal{C}$
  and *subset-closed* $\mathcal{C}$
  and *hip*: $\forall S' \subseteq S. \textit{finite } S' \longrightarrow S' \in \mathcal{C}$
  shows $(\forall F. ((\textit{FormulaAlfa } F) \wedge F \in S) \longrightarrow (S \cup \{\textit{Comp1 } F, \textit{Comp2 } F\}) \in \mathcal{C})$
lemma *cond-characterP5*:
  assumes *consistenceP* $\mathcal{C}$
  and *subset-closed* $\mathcal{C}$
  and *hip*: $\forall S' \subseteq S. \textit{finite } S' \longrightarrow S' \in \mathcal{C}$
  shows $\forall F. \textit{FormulaBeta } F \wedge F \in S \longrightarrow S \cup \{\textit{Comp1 } F\} \in \mathcal{C} \vee S \cup \{\textit{Comp2 } F\} \in \mathcal{C}$

theorem *cfinite-consistenceP*:
  assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: *subset-closed* $\mathcal{C}$

shows *consistenceP* $(\mathcal{C})$
proof –
{ fix *S*
assume $S \in \mathcal{C}$
hence *hip3*: $\forall S' \subseteq S.$ *finite* $S' \longrightarrow S' \in \mathcal{C}$
by (*simp add*: *closure-cfinite-def*)
have $(\forall P.\ \neg(atom\ P \in S \wedge (\neg.atom\ P) \in S)) \wedge$
$FF \notin S \wedge (\neg.TT) \notin S \wedge$
$(\forall F.\ (\neg.\neg.F) \in S \longrightarrow S \cup \{F\} \in \mathcal{C}) \wedge$
$(\forall F.\ ((FormulaAlfa\ F) \wedge F \in S) \longrightarrow (S \cup \{Comp1\ F,\ Comp2\ F\}) \in \mathcal{C}) \wedge$
$(\forall F.\ ((FormulaBeta\ F) \wedge F \in S) \longrightarrow$
$(S \cup \{Comp1\ F\} \in \mathcal{C}) \vee (S \cup \{Comp2\ F\} \in \mathcal{C}))$
using
cond-characterP1[*OF hip1 hip2 hip3*] *cond-characterP2*[*OF hip1 hip2 hip3*]

cond-characterP3[*OF hip1 hip2 hip3*] *cond-characterP4*[*OF hip1 hip2 hip3*]

cond-characterP5[*OF hip1 hip2 hip3*] **by** *auto* **}**
thus *?thesis* **by** (*simp add*: *consistenceP-def*)
qed

definition *maximal* :: *'a set* ⇒ *'a set set* ⇒ *bool* **where**
$maximal\ S\ \mathcal{C} = (\forall S' \in \mathcal{C}.\ S \subseteq S' \longrightarrow S = S')$

primrec *sucP* :: *'b formula set* ⇒ *'b formula set set* ⇒ (*nat* ⇒ *'b formula*) ⇒ *nat* ⇒ *'b formula set*
where
$sucP\ S\ \mathcal{C}\ f\ 0 = S$
$|\ sucP\ S\ \mathcal{C}\ f\ (Suc\ n) =$
$(if\ sucP\ S\ \mathcal{C}\ f\ n \cup \{f\ n\} \in \mathcal{C}$
$then\ sucP\ S\ \mathcal{C}\ f\ n \cup \{f\ n\}$
$else\ sucP\ S\ \mathcal{C}\ f\ n)$

definition *MsucP* :: *'b formula set* ⇒ *'b formula set set* ⇒ (*nat* ⇒ *'b formula*) ⇒ *'b formula set*
where
$MsucP\ S\ \mathcal{C}\ f = (\bigcup n.\ sucP\ S\ \mathcal{C}\ f\ n)$

theorem *Max-subsetuntoP*: $S \subseteq MsucP\ S\ \mathcal{C}\ f$

definition *chain* :: (*nat* ⇒ *'a set*) ⇒ *bool* **where**
$chain\ S = (\forall n.\ S\ n \subseteq S\ (Suc\ n))$

theorem *chain-union-closed*:
  assumes *hip1*: *finite-character* $\mathcal{C}$
  and *hip2*:*chain S*
  and *hip3*: $\forall n.\ S\ n \in \mathcal{C}$
  shows $(\bigcup n.\ S\ n) \in \mathcal{C}$

lemma *chain-suc*: *chain* (*sucP S* $\mathcal{C}$ *f*)
by (*simp add*: *chain-def*) *blast*

theorem *MaxP-in-C*:
  assumes *hip1*: *finite-character* $\mathcal{C}$ **and** *hip2*: $S \in \mathcal{C}$
  shows *MsucP S* $\mathcal{C}$ $f \in \mathcal{C}$
proof (*unfold MsucP-def*)
  have *chain* (*sucP S* $\mathcal{C}$ *f*) **by** (*rule chain-suc*)
  moreover
  have $\forall n.$ *sucP S* $\mathcal{C}$ $f\ n \in \mathcal{C}$
  proof (*rule allI*)
    fix *n*
    show *sucP S* $\mathcal{C}$ $f\ n \in \mathcal{C}$ **using** *hip2*
      by (*induct n*)(*auto simp add*: *sucP-def*)
  qed
  ultimately
  show $(\bigcup\ n.$ *sucP S* $\mathcal{C}$ $f\ n) \in \mathcal{C}$ **by** (*rule chain-union-closed*[*OF hip1*])
qed

definition *enumeration* :: (*nat* $\Rightarrow$'*b*) $\Rightarrow$ *bool* **where**
  enumeration f = $(\forall y.\exists n.\ y = (f\ n))$

lemma *enum-nat*: $\exists g.$ *enumeration* (*g*:: *nat* $\Rightarrow$ *nat*)
proof −
  have $\forall y.\ \exists\ n.\ y = (\lambda n.\ n)\ n$ **by** *simp*
  hence *enumeration* $(\lambda n.\ n)$ **by** (*unfold enumeration-def*)
  thus *?thesis* **by** *auto*
qed

theorem *suc-maximalP*:
  assumes *hip1*: *enumeration f* **and** *hip2*: *subset-closed* $\mathcal{C}$
  shows *maximal* (*MsucP S* $\mathcal{C}$ *f*) $\mathcal{C}$
proof −
  have $\forall S' \in \mathcal{C}.\ (\bigcup x.$ *sucP S* $\mathcal{C}$ $f\ x) \subseteq S' \longrightarrow (\bigcup x.$ *sucP S* $\mathcal{C}$ $f\ x) = S'$

```
  proof (rule ballI impI)+
    fix S'
    assume h1: S' ∈ 𝒞 and h2: (⋃x. sucP S 𝒞 f x) ⊆ S'
    show (⋃x. sucP S 𝒞 f x) = S'
    proof (rule ccontr)
      assume (⋃x. sucP S 𝒞 f x) ≠ S'
      hence ∃z. z ∈ S' ∧ z ∉ (⋃x. sucP S 𝒞 f x) using h2 by blast
      then obtain z where z: z ∈ S' ∧ z ∉ (⋃x. sucP S 𝒞 f x) by (rule exE)
      have ∃n. z = f n using hip1 h1 by (unfold enumeration-def) simp
      then obtain n where n: z = f n by (rule exE)
      have sucP S 𝒞 f n ∪ {f n} ⊆ S'
      proof −
        have f n ∈ S' using z n  by simp
        moreover
        have sucP S 𝒞 f n ⊆ (⋃x. sucP S 𝒞 f x) by auto
        ultimately
        show ?thesis using h2 by simp
      qed
      hence sucP S 𝒞 f n ∪ {f n} ∈ 𝒞
        using h1 hip2 by (unfold subset-closed-def) simp
      hence f n ∈ sucP S 𝒞 f (Suc n) by simp
      moreover
      have ∀x. f n ∉ sucP S 𝒞 f x using z n by simp
      ultimately show False
        by blast
    qed
  qed
  thus ?thesis
    by  (simp add: maximal-def MsucP-def)
qed

corollary ConsistentExtensionP:
  assumes hip1: finite-character 𝒞
  and hip2: S ∈ 𝒞
  and hip3:  enumeration f
  shows S ⊆ MsucP S 𝒞 f
  and MsucP S 𝒞 f ∈ 𝒞
  and maximal (MsucP S 𝒞 f) 𝒞
proof −
  show S ⊆ MsucP S 𝒞 f using Max-subsetuntoP by auto
next
  show MsucP S 𝒞 f ∈ 𝒞 using  MaxP-in-C[OF hip1 hip2] by simp
next
  show maximal (MsucP S 𝒞 f) 𝒞
    using finite-character-closed[OF hip1] and hip3 suc-maximalP
    by auto
qed
```

3 Hintikka Theorem

The formalization of Hintikka's lemma is by induction on the structure of the formulas in a Hintikka set H by applying the technical theorem `hintikkaP_model_aux`. This theorem applies a series of lemmas to address the evaluation of all possible cases of formulas in H. Indeed, considering the Boolean evaluation IH that maps all propositional letters in H to true and all other letters to false, the most interesting cases of the inductive proof are those related to implicational formulas in H and the negation of arbitrary formulas in H. These cases are not straightforward since implicational and negation formulas are not considered in the definition of Hintikka sets. For an implicational formula, say $F_1 \longrightarrow F_2$, it is necessary to prove that if it belongs to H, its evaluation by IH is true. Also, whenever $\neg(F_1 \longrightarrow F_2)$ belongs to H its evaluation is false. The proof is obtained by relating such formulas, respectively, with β and α formulas (case P6). The second interesting case is the one related to arbitrary negations. In this case, it is proved that if $\neg F$ belongs to H, its evaluation by IH is true, and in the case that $\neg\neg F$ belongs to H, its evaluation by IH is also true (Case P7).

definition *hintikkaP* :: *'b formula set* ⇒ *bool* **where**
hintikkaP H = ((∀ *P*. ¬ (*atom P* ∈ *H* ∧ (¬.*atom P*) ∈ *H*)) ∧
FF ∉ *H* ∧ (¬.*TT*) ∉ *H* ∧
(∀ *F*. (¬.¬.*F*) ∈ *H* ⟶ *F* ∈ *H*) ∧
(∀ *F*. ((*FormulaAlfa F*) ∧ *F* ∈ *H*) ⟶
((*Comp1 F*) ∈ *H* ∧ (*Comp2 F*) ∈ *H*)) ∧
(∀ *F*. ((*FormulaBeta F*) ∧ *F* ∈ *H*) ⟶
((*Comp1 F*) ∈ *H* ∨ (*Comp2 F*) ∈ *H*)))

fun *IH* :: *'b formula set* ⇒ *'b* ⇒ *v-truth* **where**
IH H P = (*if atom P* ∈ *H then Ttrue else Ffalse*)

lemma *case-P1*:
assumes *hip1*: *hintikkaP H* **and**
hip2: ∀ *G*. (*G*, *FF*) ∈ *measure f-size* ⟶
(*G* ∈ *H* ⟶ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) ∧ ((¬.*G*) ∈ *H* ⟶ *t-v-evaluation* (*IH H*) (¬.*G*)= *Ttrue*)
shows (*FF* ∈ *H* ⟶ *t-v-evaluation* (*IH H*) *FF* = *Ttrue*) ∧ ((¬.*FF*) ∈ *H* ⟶ *t-v-evaluation* (*IH H*) (¬.*FF*)=*Ttrue*)

lemma *case-P2*:
assumes *hip1*: *hintikkaP H* **and**

hip2: $\forall$ *G*. (*G*, *TT*) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*)= *Ttrue*)
shows
(*TT* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *TT* = *Ttrue*) $\wedge$ (($\neg$.*TT*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*TT*)=*Ttrue*)

lemma *case-P3*:
assumes *hip1*: *hintikkaP H* **and**
hip2: $\forall$ *G*. (*G*, *atom P*) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*)= *Ttrue*)
shows (*atom P* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) (*atom P*) = *Ttrue*) $\wedge$
(($\neg$.*atom P*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.*atom P*) = *Ttrue*)

lemma *case-P4*:
assumes *hip1*: *hintikkaP H* **and**
hip2: $\forall$ *G*. (*G*, *F1* $\wedge$. *F2*) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*) = *Ttrue*)
shows ((*F1* $\wedge$. *F2*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) (*F1* $\wedge$. *F2*) = *Ttrue*) $\wedge$
(($\neg$.(*F1* $\wedge$. *F2*)) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.(*F1* $\wedge$. *F2*)) = *Ttrue*)

lemma *case-P5*:
assumes *hip1*: *hintikkaP H* **and**
hip2: $\forall$ *G*. (*G*, *F1* $\vee$. *F2*) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*) = *Ttrue*)
shows ((*F1* $\vee$. *F2*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) (*F1* $\vee$. *F2*) = *Ttrue*) $\wedge$
(($\neg$.(*F1* $\vee$. *F2*)) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.(*F1* $\vee$. *F2*)) = *Ttrue*)

lemma *case-P6*:
assumes *hip1*: *hintikkaP H* **and**
hip2: $\forall$ *G*. (*G*, *F1* $\rightarrow$. *F2*) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*) = *Ttrue*)
shows ((*F1* $\rightarrow$. *F2*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) (*F1* $\rightarrow$. *F2*) = *Ttrue*) $\wedge$
(($\neg$.(*F1* $\rightarrow$. *F2*)) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.(*F1* $\rightarrow$. *F2*)) = *Ttrue*)

lemma *case-P7*:
assumes *hip1*: *hintikkaP H* **and**
hip2: $\forall$ *G*. (*G*, ($\neg$.*form*)) $\in$ *measure f-size* $\longrightarrow$
(*G* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) $\wedge$ (($\neg$.*G*) $\in$ *H* $\longrightarrow$ *t-v-evaluation*
(*IH H*) ($\neg$.*G*) = *Ttrue*)
shows (($\neg$.*form*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.*form*) = *Ttrue*) $\wedge$
(($\neg$.($\neg$.*form*)) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.($\neg$.*form*)) = *Ttrue*)
theorem *hintikkaP-model-aux*:
assumes *hip*: *hintikkaP H*
shows (*F* $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) *F* = *Ttrue*) $\wedge$
(($\neg$.*F*) $\in$ *H* $\longrightarrow$ *t-v-evaluation* (*IH H*) ($\neg$.*F*) = *Ttrue*)

proof (*rule wf-induct* [**where** *r*=*measure f-size* **and** *a*=*F*])
  show *wf*(*measure f-size*) **by** *simp*
next
  fix *F*
  assume *hip1*: ∀ *G*. (*G*, *F*) ∈ *measure f-size* ⟶
                (*G* ∈ *H* ⟶ *t-v-evaluation* (*IH H*) *G* = *Ttrue*) ∧
                ((¬.*G*) ∈ *H* ⟶ *t-v-evaluation* (*IH H*) (¬.*G*) = *Ttrue*)
  show (*F* ∈ *H* ⟶ *t-v-evaluation* (*IH H*) *F* = *Ttrue*) ∧
       ((¬.*F*) ∈ *H* ⟶ *t-v-evaluation* (*IH H*) (¬.*F*) = *Ttrue*)
  proof (*cases F*)
    assume *F*=*FF*
    thus *?thesis* **using** *case-P1 hip hip1* **by** *simp*
  next
    assume *F*=*TT*
    thus *?thesis* **using** *case-P2 hip hip1* **by** *auto*
  next
    fix *P*
    assume *F* = *atom P*
    thus *?thesis* **using** *hip hip1 case-P3*[*of H P*] **by** *simp*
  next
    fix *F1 F2*
    assume *F*= (*F1* ∧. *F2*)
    thus *?thesis* **using** *hip hip1 case-P4*[*of H F1 F2*] **by** *simp*
  next
    fix *F1 F2*
    assume *F*= (*F1* ∨. *F2*)
    thus *?thesis* **using** *hip hip1 case-P5*[*of H F1 F2*] **by** *simp*
  next
    fix *F1 F2*
    assume *F*= (*F1* →. *F2*)
    thus *?thesis* **using** *hip hip1 case-P6*[*of H F1 F2*] **by** *simp*
  next
    fix *F1*
    assume *F*=(¬.*F1*)
    thus *?thesis* **using** *hip hip1 case-P7*[*of H F1*] **by** *simp*
  qed
qed

corollary *ModeloHintikkaPa*:
  assumes *hintikkaP H* **and** *F* ∈ *H*
  shows *t-v-evaluation* (*IH H*) *F* = *Ttrue*
  using *assms hintikkaP-model-aux* **by** *auto*

corollary *ModeloHintikkaP*:
  assumes *hintikkaP H*
  shows (*IH H*) *model H*
proof (*unfold model-def*)

show $\forall F \in H.$ *t-v-evaluation* $(IH\ H)\ F = Ttrue$
proof (*rule ballI*)
fix F
assume $F \in H$
thus *t-v-evaluation* $(IH\ H)\ F = Ttrue$ **using** *assms ModeloHintikkaPa* **by** *auto*
qed
qed

corollary *Hintikkasatisfiable*:
assumes *hintikkaP H*
shows *satisfiable H*
using *assms ModeloHintikkaP*
by (*unfold satisfiable-def*, *auto*)

4 Maximal Hintikka

This theory formalises maximality of Hintikka sets according to Smullyan's textbook [3]. Specifically, following [1] (page 55) this theory formalises the fact that if $\mathcal{C}$ is a propositional consistence property closed by subsets, and M a maximal set belonging to $\mathcal{C}$ then M is a Hintikka set.

lemma *ext-hintikkaP1*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: $M \in \mathcal{C}$
shows $\forall p.\ \neg\ (atom\ p \in M \wedge (\neg.atom\ p) \in M)$

lemma *ext-hintikkaP2*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: $M \in \mathcal{C}$
shows $FF \notin M$

lemma *ext-hintikkaP3*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: $M \in \mathcal{C}$
shows $(\neg.TT) \notin M$

lemma *ext-hintikkaP4*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: *maximal* $M\ \mathcal{C}$ **and** *hip3*: $M \in \mathcal{C}$
shows $\forall F.\ (\neg.\neg.F) \in M \longrightarrow F \in M$

lemma *ext-hintikkaP5*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: *maximal* $M\ \mathcal{C}$ **and** *hip3*: $M \in \mathcal{C}$
shows $\forall F.\ (FormulaAlfa\ F) \wedge F \in M \longrightarrow (Comp1\ F \in M \wedge Comp2\ F \in M)$

lemma *ext-hintikkaP6*:
assumes *hip1*: *consistenceP* $\mathcal{C}$ **and** *hip2*: *maximal* $M\ \mathcal{C}$ **and** *hip3*: $M \in \mathcal{C}$
shows $\forall F.\ (FormulaBeta\ F) \wedge F \in M \longrightarrow Comp1\ F \in M \vee Comp2\ F \in M$

```
theorem MaximalHintikkaP:
  assumes hip1: consistenceP 𝒞 and hip2: maximal M 𝒞 and  hip3: M ∈ 𝒞
  shows hintikkaP M
proof (unfold hintikkaP-def)
  show (∀P. ¬ (atom P ∈ M ∧ ¬.atom P ∈ M)) ∧
        FF ∉ M ∧
        ¬.TT ∉ M ∧
        (∀F. ¬.¬.F ∈ M ⟶ F ∈ M) ∧
        (∀F. FormulaAlfa F ∧ F ∈ M ⟶ Comp1 F ∈ M ∧ Comp2 F ∈ M) ∧
        (∀F. FormulaBeta F ∧ F ∈ M ⟶ Comp1 F ∈ M ∨ Comp2 F ∈ M)
    using ext-hintikkaP1[OF hip1 hip3]
          ext-hintikkaP2[OF hip1 hip3]
          ext-hintikkaP3[OF hip1 hip3]
          ext-hintikkaP4[OF hip1 hip2 hip3]
          ext-hintikkaP5[OF hip1 hip2 hip3]
          ext-hintikkaP6[OF hip1 hip2 hip3]
    by blast
qed


lemma enumeration: enumeration f = (∃g. ∀y. f(g y) = y)
  by (metis enumeration-def)


datatype tree-b = Leaf nat | Tree tree-b tree-b

primrec diag :: nat ⇒ (nat × nat) where
  diag 0 = (0, 0)
| diag (Suc n) =
     (let (x, y) = diag n
      in case y of
          0 ⇒ (0, Suc x)
        | Suc y ⇒ (Suc x, y))


function undiag :: nat × nat ⇒ nat where
  undiag (0, 0) = 0
| undiag (0, Suc y) = Suc (undiag (y, 0))
| undiag (Suc x, y) = Suc (undiag (x, Suc y))
by pat-completeness auto

termination
  by (relation measure (λ(x, y). ((x + y) * (x + y + 1)) div 2 + x)) auto


lemma diag-undiag [simp]: diag (undiag (x, y)) = (x, y)
by (rule undiag.induct) (simp add: Let-def)+
```

lemma *enumeration-natxnat*: *enumeration* (*diag*::*nat* ⇒ (*nat* × *nat*))
proof −
  have ∀ *x y*. *diag* (*undiag* (*x*, *y*)) = (*x*, *y*) **using** *diag-undiag* **by** *auto*
  hence ∃ *undiag*. ∀ *x y*. *diag* (*undiag* (*x*, *y*)) = (*x*, *y*) **by** *blast*
  thus *?thesis* **using** *enumeration*[*of diag*] **by** *auto*
qed

function *diag-tree-b* :: *nat* ⇒ *tree-b* **where**
diag-tree-b n = (*case fst* (*diag n*) *of*
    0 ⇒ *Leaf* (*snd* (*diag n*))
  | *Suc z* ⇒ *Tree* (*diag-tree-b z*) (*diag-tree-b* (*snd* (*diag n*))))
by *auto*

primrec *undiag-tree-b* :: *tree-b* ⇒ *nat* **where**
  undiag-tree-b (*Leaf n*) = *undiag* (*0*, *n*)
| *undiag-tree-b* (*Tree t1 t2*) =
   undiag (*Suc* (*undiag-tree-b t1*), *undiag-tree-b t2*)

lemma *diag-undiag-tree-b* [*simp*]: *diag-tree-b* (*undiag-tree-b t*) = *t*
by (*induct t*) (*simp-all add*: *Let-def*)

lemma *enumeration-tree-b*: *enumeration* (*diag-tree-b* :: *nat* ⇒ *tree-b*)
proof −
  have ∀ *x*. *diag-tree-b* (*undiag-tree-b x*) = *x*
    using *diag-undiag-tree-b* **by** *blast*
  hence ∃ *undiag-tree-b*. ∀ *x* . *diag-tree-b* (*undiag-tree-b x*) = *x* **by** *blast*
  thus *?thesis* **using** *enumeration*[*of diag-tree-b*] **by** *auto*
qed

fun *formulaP-from-tree-b* :: (*nat* ⇒ *'b*) ⇒ *tree-b* ⇒ *'b formula* **where**
  formulaP-from-tree-b g (*Leaf 0*) = *FF*
| *formulaP-from-tree-b g* (*Leaf* (*Suc 0*)) = *TT*
| *formulaP-from-tree-b g* (*Leaf* (*Suc* (*Suc n*))) = (*atom* (*g n*))
| *formulaP-from-tree-b g* (*Tree* (*Leaf* (*Suc 0*)) (*Tree T1 T2*)) =
   ((*formulaP-from-tree-b g T1*) ∧. (*formulaP-from-tree-b g T2*))
| *formulaP-from-tree-b g* (*Tree* (*Leaf* (*Suc* (*Suc 0*))) (*Tree T1 T2*)) =
   ((*formulaP-from-tree-b g T1*) ∨. (*formulaP-from-tree-b g T2*))
| *formulaP-from-tree-b g* (*Tree* (*Leaf* (*Suc* (*Suc* (*Suc 0*)))) (*Tree T1 T2*)) =

```
  ((formulaP-from-tree-b g T1) →. (formulaP-from-tree-b g T2))
| formulaP-from-tree-b g (Tree (Leaf (Suc (Suc (Suc (Suc 0))))) T) =
  (¬. (formulaP-from-tree-b g T))

primrec tree-b-from-formulaP :: ('b ⇒ nat) ⇒ 'b formula ⇒ tree-b where
  tree-b-from-formulaP g FF = Leaf 0
| tree-b-from-formulaP g TT = Leaf (Suc 0)
| tree-b-from-formulaP g (atom P) = Leaf (Suc (Suc (g P)))
| tree-b-from-formulaP g (F ∧. G) = Tree (Leaf (Suc 0))
  (Tree (tree-b-from-formulaP g F) (tree-b-from-formulaP g G))
| tree-b-from-formulaP g (F ∨. G) = Tree (Leaf (Suc (Suc 0)))
  (Tree (tree-b-from-formulaP g F) (tree-b-from-formulaP g G))
| tree-b-from-formulaP g (F →. G) = Tree (Leaf (Suc (Suc (Suc 0))))
  (Tree (tree-b-from-formulaP g F) (tree-b-from-formulaP g G))
| tree-b-from-formulaP g (¬. F) = Tree (Leaf (Suc (Suc (Suc (Suc 0)))))
  (tree-b-from-formulaP g F)


definition ΔP :: (nat ⇒ 'b) ⇒ nat ⇒ 'b formula where
  ΔP g n = formulaP-from-tree-b g (diag-tree-b n)

definition ΔP' :: ('b ⇒ nat) ⇒ 'b formula ⇒ nat where
  ΔP' g' F = undiag-tree-b (tree-b-from-formulaP g' F)

theorem enumerationformulasP[simp]:
  assumes ∀ x. g(g' x) = x
  shows ΔP g (ΔP' g' F) = F
using assms
by (induct F)(simp-all add: ΔP-def ΔP'-def)


corollary EnumerationFormulasP:
  assumes ∀ P. ∃ n. P = g n
  shows ∀ F. ∃ n. F = ΔP g n
proof (rule allI)
  fix F
  { have ∀ P. P = g (SOME n. P = (g n))
    proof(rule allI)
      fix P
      obtain n where n: P=g(n) using assms by auto
      thus P = g (SOME n. P = (g n)) by (rule someI)
    qed }
  hence ∀ P. g((λP. SOME n. P = (g n)) P) = P by simp
  hence F = ΔP g (ΔP' (λP. SOME n. P = (g n)) F)
    using enumerationformulasP by simp
  thus ∃ n. F = ΔP g n
    by blast
qed
```

corollary *EnumerationFormulasP1*:
assumes *enumeration* $(g\!:: nat \Rightarrow {}'b)$
shows *enumeration* $((\Delta P\ g)\!:: nat \Rightarrow {}'b\ formula)$
proof −
have $\forall P.\ \exists\ n.\ P = g\ n$ **using** *assms* **by**(*unfold enumeration-def*)
hence $\forall F.\ \exists n.\ F = \Delta P\ g\ n$ **using** *EnumerationFormulasP* **by** *auto*
thus *?thesis* **by**(*unfold enumeration-def*)
qed

corollary *EnumeracionFormulasNat*:
shows $\exists\ f.$ *enumeration* $(f\!:: nat \Rightarrow nat\ formula)$
proof −
obtain *g* **where** *g*: *enumeration* $(g\!:: nat \Rightarrow nat)$ **using** *enum-nat* **by** *auto*
thus $\exists\ f.$ *enumeration* $(f\!:: nat \Rightarrow nat\ formula)$
using *enum-nat EnumerationFormulasP1* **by** *auto*
qed

5 Model Existence Theorem

This theory formalises the Model Existence Theorem according to Smullyan's textbook [3] as presented by Fitting in [1].

theorem *ExtensionCharacterFinitoP*:
shows $\mathcal{C} \subseteq \mathcal{C}$
and *finite-character* $(\mathcal{C})$
and *consistenceP* $\mathcal{C} \longrightarrow$ *consistenceP* $(\mathcal{C})$
proof −
show $\mathcal{C} \subseteq \mathcal{C}$
proof −
have $\mathcal{C} \subseteq \mathcal{C}$ **using** *closed-subset* **by** *auto*
also
have ... $\subseteq \mathcal{C}$
proof −
have *subset-closed* $(\mathcal{C})$ **using** *closed-closed* **by** *auto*
thus *?thesis* **using** *finite-character-subset* **by** *auto*
qed
finally show *?thesis* **by** *simp*
qed
next
show *finite-character* $(\mathcal{C})$ **using** *finite-character* **by** *auto*
next
show *consistenceP* $\mathcal{C} \longrightarrow$ *consistenceP* $(\mathcal{C})$
proof(*rule impI*)
assume *consistenceP* $\mathcal{C}$

hence *consistenceP* (𝒞) **using** *closed-consistenceP* **by** *auto*
moreover
have *subset-closed* (𝒞) **using** *closed-closed* **by** *auto*
ultimately
show *consistenceP* (𝒞) **using** *cfinite-consistenceP*
by *auto*
qed
qed

lemma *ExtensionConsistenteP1*:
assumes *h*: *enumeration g*
and *h1*: *consistenceP* 𝒞
and *h2*: *S* ∈ 𝒞
shows *S* ⊆ *MsucP S* (𝒞) *g*
and *maximal* (*MsucP S* (𝒞) *g*) (𝒞)
and *MsucP S* (𝒞) *g* ∈ 𝒞

proof −
have *consistenceP* (𝒞)
using *h1* **and** *ExtensionCharacterFinitoP* **by** *auto*
moreover
have *finite-character* (𝒞) **using** *ExtensionCharacterFinitoP* **by** *auto*
moreover
have *S* ∈ 𝒞
using *h2* **and** *ExtensionCharacterFinitoP* **by** *auto*
ultimately
show *S* ⊆ *MsucP S* (𝒞) *g*
and *maximal* (*MsucP S* (𝒞) *g*) (𝒞)
and *MsucP S* (𝒞) *g* ∈ 𝒞
using *h ConsistentExtensionP*[*of* 𝒞] **by** *auto*
qed

theorem *HintikkaP*:
assumes *h0*:*enumeration g* **and** *h1*: *consistenceP* 𝒞 **and** *h2*: *S* ∈ 𝒞
shows *hintikkaP* (*MsucP S* (𝒞) *g*)
proof −
have *1*: *consistenceP* (𝒞)
using *h1 ExtensionCharacterFinitoP* **by** *auto*
have *2*: *subset-closed* (𝒞)
proof −
have *finite-character* (𝒞)
using *ExtensionCharacterFinitoP* **by** *auto*
thus *subset-closed* (𝒞) **by** (*rule finite-character-closed*)
qed
have *3*: *maximal* (*MsucP S* (𝒞) *g*) (𝒞)
and *4*: *MsucP S* (𝒞) *g* ∈ 𝒞
using *ExtensionConsistenteP1*[*OF h0 h1 h2*] **by** *auto*

show *?thesis*
using *1* **and** *2* **and** *3* **and** *4* **and** *MaximalHintikkaP*[*of* $\mathcal{C}$] **by** *simp*
qed

theorem *ExistenceModelP*:
assumes *h0*: *enumeration g*
and *h1*: *consistenceP* $\mathcal{C}$
and *h2*: $S \in \mathcal{C}$
and *h3*: $F \in S$
shows *t-v-evaluation* (*IH* (*MsucP S* ($\mathcal{C}$) *g*)) *F* = *Ttrue*
proof (*rule ModeloHintikkaPa*)
show *hintikkaP* (*MsucP S* ($\mathcal{C}$) *g*)
using *h0* **and** *h1* **and** *h2* **by**(*rule HintikkaP*)
next
show $F \in$ *MsucP S* ($\mathcal{C}$) *g*
using *h3 Max-subsetuntoP* **by** *auto*
qed

theorem *Theo-ExistenceModels*:
assumes *h1*: $\exists$ *g*. *enumeration* (*g*:: *nat* $\Rightarrow$ *'b formula*)
and *h2*: *consistenceP* $\mathcal{C}$
and *h3*: (*S*:: *'b formula set*) $\in \mathcal{C}$
shows *satisfiable S*
proof −
obtain *g* **where** *g*: *enumeration* (*g*:: *nat* $\Rightarrow$ *'b formula*)
using *h1* **by** *auto*
{ fix *F*
assume *hip*: $F \in S$
have *t-v-evaluation* (*IH* (*MsucP S* ($\mathcal{C}$) *g*)) *F* = *Ttrue*
using *g h2 h3 ExistenceModelP hip* **by** *blast* **}**
hence $\forall F \in S$. *t-v-evaluation* (*IH* (*MsucP S* ($\mathcal{C}$) *g*)) *F* = *Ttrue*
by (*rule ballI*)
hence $\exists$ *I*. $\forall$ $F \in S$. *t-v-evaluation I F* = *Ttrue* **by** *auto*
thus *satisfiable S* **by**(*unfold satisfiable-def*, *unfold model-def*)
qed

corollary *Satisfiable-SetP1*:
assumes *h0*: $\exists$ *g*. *enumeration* (*g*:: *nat* $\Rightarrow$ *'b*)
and *h1*: *consistenceP* $\mathcal{C}$
and *h2*: (*S*:: *'b formula set*) $\in \mathcal{C}$
shows *satisfiable S*
proof −
obtain *g* **where** *g*: *enumeration* (*g*:: *nat* $\Rightarrow$ *'b*)
using *h0* **by** *auto*
have *enumeration* ((ΔP *g*):: *nat* $\Rightarrow$ *'b formula*) **using** *g EnumerationFormulasP1*

by *auto*
  hence $h'0$: $\exists\, g.$ *enumeration* (g:: *nat* $\Rightarrow$ *'b formula*) **by** *auto*
  show *?thesis* **using** *Theo-ExistenceModels*[*OF h'0 h1 h2*] **by** *auto*
qed

corollary *Satisfiable-SetP2*:
  assumes *consistenceP* $\mathcal{C}$ **and** (S:: *nat formula set*) $\in \mathcal{C}$
  shows *satisfiable S*
  using *enum-nat assms Satisfiable-SetP1* **by** *auto*

theory *PropCompactness*

imports *Main*
HOL−Library.Countable-Set
ModelExistence

begin

6 Compactness Theorem for Propositional Logic

This theory formalises the compactness theorem based on the existence model theorem. The formalisation, initially published as [2] in Spanish, was adapted to extend several combinatorial theorems over finite structures to the infinite case (e.g., see Serrano, Ayala-Rincón, and de Lima formalizations of Hall's Theorem for infinite families of sets and infinite graphs [4, 5].)

The formalization shows first Hintikka's Lemma: Hintikka sets of propositional formulas are satisfiable. Such a set is defined as a set of propositional formulas that does neither include both A and $\neg A$ for a propositional letter nor $\bot$, or $\neg\top$. Additionally, if it includes $\neg\neg F$, F is included; if it includes a conjunctive formula, which is an α formula, then the two components of the conjunction are included; and finally, if it includes a disjunction, which is a β formula, at least one of the components of the disjunction is included.

The satisfiability of any Hintikka set is proved by assuming a valuation that maps all propositional letters in the set to true and all other propositional letters to false. The second step consists in proving that families of sets of propositional formulas, which hold the so-called "propositional consistency property," consist of satisfiable sets. The last is indeed the model existence theorem. The model existence theorem compiles the essence of completeness: a family of sets of propositional formulas that holds the propositional consistency property can be extended, preserving this property to a set collection that is closed for subsets and satisfies the finite character property. The finite character property states that a set belongs to the family if and only if each of its finite subsets belongs to the family. With the model existence theorem in hands, the compactness theorem is obtained easily: given a set of propositional formulas S such that all its finite subsets are satisfiable, one considers the family $\mathcal{C}$ of subsets in S such that all their finite subsets are satisfiable. S belongs to the family $\mathcal{C}$ and the latter holds the propositional consistence property.

The auxiliary lemma of Consistence Compactness is required to apply the Model Existence Theorem to obtain the compactness theorem. This lemma states the general fact that the collection $\mathcal{C}$ of all sets of propositional formulas such that all their subsets are satisfiable is a propositional consistency property.

```
lemma UnsatisfiableAtom:
  shows ¬ (satisfiable {F, ¬.F})
proof (rule notI)
  assume hip: satisfiable {F, ¬.F}
  show False
  proof −
    have ∃I. I model {F, ¬.F} using hip by(unfold satisfiable-def, auto)
    then obtain I where I: (t-v-evaluation I F) = Ttrue
      and (t-v-evaluation I (¬.F)) = Ttrue
      by(unfold model-def, auto)
    thus False by(auto simp add: v-negation-def)
  qed
qed


lemma consistenceP-Prop1:
  assumes ∀ (A::'b formula set). (A⊆ W ∧ finite A) ⟶ satisfiable A
  shows (∀P. ¬ (Atom P ∈ W ∧ (¬. Atom P) ∈ W))
proof (rule allI notI)+
  fix P
  assume h1: Atom P ∈ W ∧ (¬.Atom P) ∈ W
  show False
  proof −
    have {Atom P, (¬.Atom P)} ⊆ W using h1 by simp
    moreover
    have finite {Atom P, (¬.Atom P)} by simp
    ultimately
    have {Atom P, (¬.Atom P)} ⊆ W ∧ finite {Atom P, (¬.Atom P)} by simp
    thus False using UnsatisfiableAtom assms
      by metis
  qed
qed

lemma UnsatisfiableFF:
  shows ¬ (satisfiable {FF})
proof −
```

have ∀ *I*. *t-v-evaluation I FF* = *Ffalse* **by** *simp*
hence ∀ *I*. ¬ (*I model* {*FF*}) **by**(*unfold model-def*, *auto*)
thus *?thesis* **by**(*unfold satisfiable-def*, *auto*)
qed

lemma *consistenceP-Prop2*:
assumes ∀ (*A*::′*b formula set*). (*A*⊆ *W* ∧ *finite A*) ⟶ *satisfiable A*
shows *FF* ∉ *W*
proof (*rule notI*)
assume *hip*: *FF* ∈ *W*
show *False*
proof −
have {*FF*} ⊆ *W* **using** *hip* **by** *simp*
moreover
have *finite* {*FF*} **by** *simp*
ultimately
have {*FF*} ⊆ *W* ∧ *finite* {*FF*} **by** *simp*
moreover
have ({*FF*::′*b formula*} ⊆ *W* ∧ *finite* {*FF*}) ⟶ *satisfiable* {*FF*::′*b formula*}
using *assms* **by** *auto*
ultimately show *False* **using** *UnsatisfiableFF* **by** *auto*
qed
qed

lemma *UnsatisfiableFFa*:
shows ¬ (*satisfiable* {¬.*TT*})
proof −
have ∀ *I*. *t-v-evaluation I TT* = *Ttrue* **by** *simp*
have ∀ *I*. *t-v-evaluation I* (¬.*TT*) = *Ffalse* **by**(*auto simp add*:*v-negation-def*)
hence ∀ *I*. ¬ (*I model* {¬.*TT*}) **by**(*unfold model-def*, *auto*)
thus *?thesis* **by**(*unfold satisfiable-def*, *auto*)
qed

lemma *consistenceP-Prop3*:
assumes ∀ (*A*::′*b formula set*). (*A*⊆ *W* ∧ *finite A*) ⟶ *satisfiable A*
shows ¬.*TT* ∉ *W*
proof (*rule notI*)
assume *hip*: ¬.*TT* ∈ *W*
show *False*
proof −
have {¬.*TT*} ⊆ *W* **using** *hip* **by** *simp*
moreover
have *finite* {¬.*TT*} **by** *simp*
ultimately
have {¬.*TT*} ⊆ *W* ∧ *finite* {¬.*TT*} **by** *simp*
moreover
have ({¬.*TT*::′*b formula*} ⊆ *W* ∧ *finite* {¬.*TT*}) ⟶
satisfiable {¬.*TT*::′*b formula*}
using *assms* **by** *auto*

```
    thus False using UnsatisfiableFFa
      using ‹{¬.TT} ⊆ W› by auto
  qed
qed

lemma Subset-Sat:
  assumes hip1: satisfiable S and hip2: S′⊆ S
  shows satisfiable S′
  using assms satisfiable-subset by blast

lemma satisfiableUnion1:
  assumes satisfiable (A ∪ {¬.¬.F})
  shows satisfiable (A ∪ {F})
proof −
  have ∃ I. ∀ G ∈ (A ∪ {¬.¬.F}). t-v-evaluation I G = Ttrue
    using assms by(unfold satisfiable-def, unfold model-def, auto)
  then obtain I where I: ∀ G ∈ (A ∪ {¬.¬.F}). t-v-evaluation I G = Ttrue
    by auto
  hence 1: ∀ G ∈ A. t-v-evaluation I G = Ttrue
    and 2: t-v-evaluation I (¬.¬.F) = Ttrue
    by auto
  have typeFormula (¬.¬.F) = NoNo by auto
  hence t-v-evaluation I F = Ttrue using EquivNoNoComp[of ¬.¬.F] 2
    by (unfold equivalent-def, unfold Comp1-def, auto)
  hence ∀ G ∈ A ∪ {F}. t-v-evaluation I G = Ttrue using 1 by auto
  thus satisfiable (A ∪ {F})
    by(unfold satisfiable-def, unfold model-def, auto)
qed


lemma consistenceP-Prop4:
  assumes hip1: ∀ (A::′b formula set). (A⊆ W ∧ finite A) ⟶ satisfiable A
  and hip2: ¬.¬.F ∈ W
  shows ∀ (A::′b formula set). (A⊆ W ∪ {F} ∧ finite A) ⟶ satisfiable A
proof (rule allI, rule impI)+
  fix A
  assume hip: A ⊆ W ∪ {F} ∧ finite A
  show satisfiable A
  proof −
    have A−{F} ⊆ W ∧ finite (A−{F}) using hip by auto
    hence (A−{F}) ∪ {¬.¬.F} ⊆ W ∧ finite ((A−{F}) ∪ {¬.¬.F})
      using hip2 by auto
    hence satisfiable ((A−{F}) ∪ {¬.¬.F}) using hip1 by auto
    hence satisfiable ((A−{F}) ∪ {F}) using satisfiableUnion1 by blast
    moreover
    have A⊆ (A−{F}) ∪ {F} by auto
    ultimately
    show satisfiable A using Subset-Sat by auto
  qed
qed
```

lemma *satisfiableUnion2*:
assumes *hip1*: *FormulaAlfa F* **and** *hip2*: *satisfiable* $(A \cup \{F\})$
shows *satisfiable* $(A \cup \{$*Comp1 F*,*Comp2 F*$\})$
proof −
have $\exists I.\forall\ G \in A \cup \{F\}$. *t-v-evaluation I G* = *Ttrue*
using *hip2* **by**(*unfold satisfiable-def*, *unfold model-def*, *auto*)
then obtain *I* **where** *I*: $\forall\ G \in A \cup \{F\}$. *t-v-evaluation I G* = *Ttrue* **by** *auto*

hence *1*: $\forall\ G \in A$. *t-v-evaluation I G* = *Ttrue* **and** *2*: *t-v-evaluation I F* = *Ttrue* **by** *auto*
have *typeFormula F* = *Alfa* **using** *hip1 noAlfaBeta noAlfaNoNo* **by** *auto*
hence *equivalent F* (*Comp1 F* ∧. *Comp2 F*)
using *2 EquivAlfaComp*[*of F*] **by** *auto*
hence *t-v-evaluation I* (*Comp1 F* ∧. *Comp2 F*) = *Ttrue*
using *2* **by**(*unfold equivalent-def*, *auto*)
hence *t-v-evaluation I* (*Comp1 F*) = *Ttrue* ∧ *t-v-evaluation I* (*Comp2 F*) = *Ttrue*
using *ConjunctionValues* **by** *auto*
hence $\forall\ G \in A \cup \{$*Comp1 F*, *Comp2 F*$\}$. *t-v-evaluation I G* = *Ttrue* **using** *1* **by** *auto*
thus *satisfiable* $(A \cup \{$*Comp1 F*,*Comp2 F*$\})$
by (*unfold satisfiable-def*, *unfold model-def*, *auto*)
qed

lemma *consistenceP-Prop5*:
assumes *hip0*: *FormulaAlfa F*
and *hip1*: ∀ (*A*::′*b formula set*). (*A*⊆ *W* ∧ *finite A*) ⟶ *satisfiable A*
and *hip2*: $F \in W$
shows ∀ (*A*::′*b formula set*). (*A*⊆ *W* ∪ {*Comp1 F*, *Comp2 F*} ∧ *finite A*) ⟶ *satisfiable A*
proof (*intro allI impI*)
fix *A*
assume *hip*: *A* ⊆ *W* ∪ {*Comp1 F*, *Comp2 F*} ∧ *finite A*
show *satisfiable A*
proof −
have *A*−{*Comp1 F*, *Comp2 F*} ⊆ *W* ∧ *finite* (*A*−{*Comp1 F*, *Comp2 F*})
using *hip* **by** *auto*
hence (*A*−{*Comp1 F*, *Comp2 F*}) ∪ {*F*} ⊆ *W* ∧
finite ((*A*−{*Comp1 F*, *Comp2 F*}) ∪ {*F*})
using *hip2* **by** *auto*
hence *satisfiable* ((*A*−{*Comp1 F*, *Comp2 F*}) ∪ {*F*})
using *hip1* **by** *auto*
hence *satisfiable* ((*A*−{*Comp1 F*, *Comp2 F*}) ∪ {*Comp1 F*, *Comp2 F*})
using *hip0 satisfiableUnion2* **by** *auto*
moreover
have *A* ⊆ (*A*−{*Comp1 F*, *Comp2 F*}) ∪ {*Comp1 F*, *Comp2 F*} **by** *auto*
ultimately

show *satisfiable A* **using** *Subset-Sat* **by** *auto*
qed
qed

lemma *satisfiableUnion3*:
assumes *hip1*: *FormulaBeta F* **and** *hip2*: *satisfiable* (*A* ∪ {*F*})
shows *satisfiable* (*A* ∪ {*Comp1 F*}) ∨ *satisfiable* (*A* ∪ {*Comp2 F*})
proof −
obtain *I* **where** *I*: ∀ *G* ∈ (*A* ∪ {*F*}). *t-v-evaluation I G* = *Ttrue*
using *hip2* **by**(*unfold satisfiable-def*, *unfold model-def*, *auto*)
hence *S1*: ∀ *G* ∈ *A*. *t-v-evaluation I G* = *Ttrue*
and *S2*: *t-v-evaluation I F* = *Ttrue*
by *auto*
have *V*: *t-v-evaluation I* (*Comp1 F*) = *Ttrue* ∨ *t-v-evaluation I* (*Comp2 F*) = *Ttrue*
using *hip1 S2 EquivBetaComp*[*of F*] *DisjunctionValues*
by (*unfold equivalent-def*, *auto*)
have ((∀ *G* ∈ *A*. *t-v-evaluation I G* = *Ttrue*) ∧ *t-v-evaluation I* (*Comp1 F*) = *Ttrue*) ∨
((∀ *G* ∈ *A*. *t-v-evaluation I G* = *Ttrue*) ∧ *t-v-evaluation I* (*Comp2 F*) = *Ttrue*)
using *V*
proof (*rule disjE*)
assume *t-v-evaluation I* (*Comp1 F*) = *Ttrue*
hence (∀ *G* ∈ *A*. *t-v-evaluation I G* = *Ttrue*) ∧ *t-v-evaluation I* (*Comp1 F*) = *Ttrue*
using *S1* **by** *auto*
thus *?thesis* **by** *simp*
next
assume *t-v-evaluation I* (*Comp2 F*) = *Ttrue*
hence (∀ *G* ∈ *A*. *t-v-evaluation I G* = *Ttrue*) ∧ *t-v-evaluation I* (*Comp2 F*) = *Ttrue*
using *S1* **by** *auto*
thus *?thesis* **by** *simp*
qed
hence (∀ *G* ∈ *A* ∪ {*Comp1 F*}. *t-v-evaluation I G* = *Ttrue*) ∨
(∀ *G* ∈ *A* ∪ {*Comp2 F*}. *t-v-evaluation I G* = *Ttrue*)
by *auto*
hence (∃ *I*.∀ *G* ∈ *A* ∪ {*Comp1 F*}. *t-v-evaluation I G* = *Ttrue*) ∨
(∃ *I*.∀ *G* ∈ *A* ∪ {*Comp2 F*}. *t-v-evaluation I G* = *Ttrue*)
by *auto*
thus *satisfiable* (*A* ∪ {*Comp1 F*}) ∨ *satisfiable* (*A* ∪ {*Comp2 F*})
by (*unfold satisfiable-def*, *unfold model-def*, *auto*)
qed

lemma *consistenceP-Prop6*:
assumes *hip0*: *FormulaBeta F*

```
  and hip1: ∀ (A::'b formula set). (A⊆ W ∧ finite A) ⟶ satisfiable A
  and hip2: F ∈ W
  shows (∀ (A::'b formula set). (A⊆ W ∪ {Comp1 F} ∧ finite A) ⟶
  satisfiable A) ∨
  (∀ (A::'b formula set). (A⊆ W ∪ {Comp2 F} ∧ finite A) ⟶
  satisfiable A)
proof −
  { assume hip3:¬((∀ (A::'b formula set). (A⊆ W ∪ {Comp1 F} ∧ finite A) ⟶
    satisfiable A) ∨
    (∀ (A::'b formula set). (A⊆ W ∪ {Comp2 F} ∧ finite A) ⟶
    satisfiable A))
    have False
    proof −
      obtain A B where A1: A ⊆ W ∪ {Comp1 F}
        and A2: finite A
        and A3: ¬ satisfiable A
        and B1: B ⊆ W ∪ {Comp2 F}
        and B2: finite B
        and B3: ¬ satisfiable B
        using hip3 by auto
      have a1: A − {Comp1 F} ⊆ W
        and a2: finite (A − {Comp1 F})
        using A1 and A2 by auto
      hence satisfiable (A − {Comp1 F}) using hip1 by simp
      have b1: B − {Comp2 F} ⊆ W
        and b2: finite (B − {Comp2 F})
        using B1 and B2 by auto
      hence satisfiable (B − {Comp2 F}) using hip1 by simp
      moreover
      have (A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {F} ⊆ W
        and finite ((A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {F})
        using a1 a2 b1 b2 hip2 by auto
      hence satisfiable ((A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {F})
        using hip1 by simp
      hence satisfiable ((A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {Comp1 F})
      ∨ satisfiable ((A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {Comp2 F})
        using hip0 satisfiableUnion3 by auto
      moreover
      have A ⊆ (A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {Comp1 F}
        and B ⊆ (A − {Comp1 F}) ∪ (B − {Comp2 F}) ∪ {Comp2 F}
        by auto
      ultimately
      have satisfiable A ∨ satisfiable B using Subset-Sat by auto
      thus False using A3 B3 by simp
    qed }
  thus ?thesis by auto
qed

lemma ConsistenceCompactness:
```

shows *consistenceP*{*W*::′*b formula set*. ∀ *A*. (*A*⊆ *W* ∧ *finite A*) ⟶
satisfiable A}
proof (*unfold consistenceP-def*, *rule allI*, *rule impI*)
let *?C* = {*W*::′*b formula set*. ∀ *A*. (*A*⊆ *W* ∧ *finite A*) ⟶ *satisfiable A*}
fix *W* :: ′*b formula set*
assume *W* ∈ *?C*
hence *hip*: ∀ *A*. (*A*⊆ *W* ∧ *finite A*) ⟶ *satisfiable A* **by** *simp*
show (∀ *P*. ¬ (*atom P* ∈ *W* ∧ (¬.*atom P*) ∈ *W*)) ∧
FF ∉ *W* ∧
¬.*TT* ∉ *W* ∧
(∀ *F*. ¬.¬.*F* ∈ *W* ⟶ *W* ∪ {*F*} ∈ *?C*) ∧
(∀ *F*. (*FormulaAlfa F*) ∧ *F* ∈ *W* ⟶
(*W* ∪ {*Comp1 F*, *Comp2 F*} ∈ *?C*)) ∧
(∀ *F*. (*FormulaBeta F*) ∧ *F* ∈ *W* ⟶
(*W* ∪ {*Comp1 F*} ∈ *?C* ∨ *W* ∪ {*Comp2 F*} ∈ *?C*))
proof −
have (∀ *P*. ¬ (*atom P* ∈ *W* ∧ (¬. *atom P*) ∈ *W*))
using *hip consistenceP-Prop1* **by** *simp*
moreover
have *FF* ∉ *W* **using** *hip consistenceP-Prop2* **by** *auto*
moreover
have ¬. *TT* ∉ *W* **using** *hip consistenceP-Prop3* **by** *auto*
moreover
have ∀ *F*. (¬.¬.*F*) ∈ *W* ⟶ *W* ∪ {*F*} ∈ *?C*
proof (*rule allI impI*)+
fix *F*
assume *hip1*: ¬.¬.*F* ∈ *W*
show *W* ∪ {*F*} ∈ *?C* **using** *hip hip1 consistenceP-Prop4* **by** *simp*
qed
moreover
have
∀ *F*. (*FormulaAlfa F*) ∧ *F* ∈ *W* ⟶ (*W* ∪ {*Comp1 F*, *Comp2 F*} ∈ *?C*)
proof (*rule allI impI*)+
fix *F*
assume *FormulaAlfa F* ∧ *F* ∈ *W*
thus *W* ∪ {*Comp1 F*, *Comp2 F*} ∈ *?C* **using** *hip consistenceP-Prop5*[*of F*]
by *blast*
qed
moreover
have ∀ *F*. (*FormulaBeta F*) ∧ *F* ∈ *W* ⟶
(*W* ∪ {*Comp1 F*} ∈ *?C* ∨ *W* ∪ {*Comp2 F*} ∈ *?C*)
proof (*rule allI impI*)+
fix *F*
assume (*FormulaBeta F*) ∧ *F* ∈ *W*
thus *W* ∪ {*Comp1 F*} ∈ *?C* ∨ *W* ∪ {*Comp2 F*} ∈ *?C*
using *hip consistenceP-Prop6*[*of F*] **by** *blast*
qed
ultimately
show *?thesis* **by** *auto*

```
  qed
qed

lemma countable-enumeration-formula:
  shows ∃f. enumeration (f:: nat ⇒'a::countable formula)
  by (metis(full-types) EnumerationFormulasP1
      enumeration-def surj-def surj-from-nat)

theorem Compactness-Theorem:
  assumes ∀A. (A ⊆ (S:: 'a::countable formula set) ∧ finite A) ⟶ satisfiable A
  shows satisfiable S
proof −
  have enum: ∃g. enumeration (g:: nat ⇒ 'a formula)
    using countable-enumeration-formula by auto
    let ?C = {W:: 'a formula set.  ∀A. (A ⊆ W ∧ finite A) ⟶ satisfiable A}
  have consistenceP ?C
    using ConsistenceCompactness by simp
  moreover
  have S ∈ ?C using assms by simp
  ultimately
  show satisfiable S using enum and Theo-ExistenceModels[of ?C S] by auto
qed

end

theory Hall-Theorem
  imports
   PropCompactness
   Marriage.Marriage
begin
```

7 Hall Theorem for countable (infinite) families of sets

Hall's Theorem for countable families of sets is proved as a consequence of compactness theorem for propositional calculus ([4]). The theory imports Marriage theory from the AFP, which proves marriage theorem for the finite case. The proof also uses an updated version of Serrano's formalization of the compactness theorem for propositional logic.

```
 definition system-representatives :: ('a ⇒ 'b set) ⇒ 'a set ⇒ ('a ⇒ 'b) ⇒ bool
where
system-representatives S I R  ≡ (∀ i∈I. (R i) ∈ (S i)) ∧ (inj-on R I)

definition set-to-list :: 'a set ⇒ 'a list
  where set-to-list s =  (SOME l. set l = s)

lemma  set-set-to-list:
```

finite s ⟹ *set* (*set-to-list s*) = *s*
unfolding *set-to-list-def* **by** (*metis* (*mono-tags*) *finite-list some-eq-ex*)

lemma *list-to-set*:
assumes *finite* (*S i*)
shows *set* (*set-to-list* (*S i*)) = (*S i*)
using *assms set-set-to-list* **by** *auto*

primrec *disjunction-atomic* :: *'b list* ⇒*'a* ⇒ (*'a* × *'b*)*formula* **where**
disjunction-atomic [] *i* = *FF*
| *disjunction-atomic* (*x#D*) *i* = (*atom* (*i*, *x*)) ∨. (*disjunction-atomic D i*)

lemma *t-v-evaluation-disjunctions1*:
assumes *t-v-evaluation I* (*disjunction-atomic* (*a* # *l*) *i*) = *Ttrue*
shows *t-v-evaluation I* (*atom* (*i*,*a*)) = *Ttrue* ∨ *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue*
proof−
have
(*disjunction-atomic* (*a* # *l*) *i*) = (*atom* (*i*,*a*)) ∨. (*disjunction-atomic l i*)
by *auto*
hence *t-v-evaluation I* ((*atom* (*i* ,*a*)) ∨. (*disjunction-atomic l i*)) = *Ttrue*
using *assms* **by** *auto*
thus *?thesis* **using** *DisjunctionValues* **by** *blast*
qed

lemma *t-v-evaluation-atom*:
assumes *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue*
shows ∃ *x*. *x* ∈ *set l* ∧ (*t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue*)
proof−
have *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue* ⟹
∃ *x*. *x* ∈ *set l* ∧ (*t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue*)
proof(*induct l*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a l*)
show ∃ *x*. *x* ∈ *set* (*a* # *l*) ∧ *t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue*
proof−
have
(*t-v-evaluation I* (*atom* (*i*,*a*)) = *Ttrue*) ∨ *t-v-evaluation I* (*disjunction-atomic l i*)=*Ttrue*
using *Cons*(*2*) *t-v-evaluation-disjunctions1*[*of I*] **by** *auto*
thus *?thesis*
proof(*rule disjE*)
assume *t-v-evaluation I* (*atom* (*i*,*a*)) = *Ttrue*
thus *?thesis* **by** *auto*
next
assume *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue*
thus *?thesis* **using** *Cons* **by** *auto*

qed
qed
qed
thus *?thesis* **using** *assms* **by** *auto*
qed

definition $\mathcal{F}$ :: $('a \Rightarrow 'b\ set) \Rightarrow 'a\ set \Rightarrow (('a \times 'b)formula)\ set$ **where**
$\mathcal{F}\ S\ I \equiv (\bigcup i{\in}I.\ \{\ disjunction\text{-}atomic\ (set\text{-}to\text{-}list\ (S\ i))\ i\ \})$

definition $\mathcal{G}$ :: $('a \Rightarrow 'b\ set) \Rightarrow 'a\ set \Rightarrow ('a \times 'b)formula\ set$ **where**
$\mathcal{G}\ S\ I \equiv \{\neg.(atom\ (i,x)\ \wedge.\ atom(i,y))$
$|x\ y\ i\ .\ x{\in}(S\ i) \wedge y{\in}(S\ i) \wedge\ x{\neq}y \wedge i{\in}I\}$

definition $\mathcal{H}$ :: $('a \Rightarrow 'b\ set) \Rightarrow 'a\ set \Rightarrow('a \times 'b)formula\ set$ **where**
$\mathcal{H}\ S\ I \equiv \{\neg.(atom\ (i,x)\ \wedge.\ atom(j,x))$
$|\ x\ i\ j.\ x \in (S\ i) \cap (S\ j) \wedge (i{\in}I \wedge j{\in}I \wedge i{\neq}j)\}$

definition $\mathcal{T}$ :: $('a \Rightarrow 'b\ set) \Rightarrow 'a\ set \Rightarrow ('a \times 'b)formula\ set$ **where**
$\mathcal{T}\ S\ I \equiv (\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I) \cup (\mathcal{H}\ S\ I)$

primrec *indices-formula* :: $('a \times 'b)formula \Rightarrow 'a\ set$ **where**
indices-formula $FF = \{\}$
| *indices-formula* $TT = \{\}$
| *indices-formula* $(atom\ P) = \{fst\ P\}$
| *indices-formula* $(\neg.\ F) =$ *indices-formula* F
| *indices-formula* $(F \wedge.\ G) =$ *indices-formula* $F \cup$ *indices-formula* G
| *indices-formula* $(F \vee.\ G) =$ *indices-formula* $F \cup$ *indices-formula* G
| *indices-formula* $(F \rightarrow.\ G) =$ *indices-formula* $F \cup$ *indices-formula* G

definition *indices-set-formulas* :: $('a \times 'b)formula\ set \Rightarrow 'a\ set$ **where**
indices-set-formulas $S = (\bigcup F{\in}\ S.$ *indices-formula* $F)$

lemma *finite-indices-formulas*:
shows *finite* (*indices-formula* F)
by(*induct* F, *auto*)

lemma *finite-set-indices*:
assumes *finite* S
shows *finite* (*indices-set-formulas* S)
using ‹*finite* S› *finite-indices-formulas*
by(*unfold indices-set-formulas-def*, *auto*)

lemma *indices-disjunction*:
assumes $F =$ *disjunction-atomic* $L\ \ i$ **and** $L \neq []$
shows *indices-formula* $F = \{i\}$
proof−
have $(F =$ *disjunction-atomic* $L\ \ i\ \wedge\ L \neq []) \Longrightarrow$ *indices-formula* $F = \{i\}$
proof(*induct* L *arbitrary*: F)
case *Nil* **hence** *False* **using** *assms* **by** *auto*

```
    thus ?case by auto
  next
    case(Cons a L)
    assume F = disjunction-atomic (a # L) i ∧ a # L ≠ []
    thus ?case
    proof(cases L)
     assume L = []
      thus indices-formula F = {i} using Cons(2) by auto
    next
      show
  ⋀b list. F = disjunction-atomic (a # L) i ∧ a # L ≠ [] ⟹ L = b # list ⟹
            indices-formula F = {i}
        using Cons(1−2) by auto
    qed
  qed
  thus ?thesis using assms  by auto
qed

lemma nonempty-set-list:
  assumes ∀ i∈I. (S i)≠{} and ∀ i∈I. finite (S i)
  shows ∀ i∈I. set-to-list (S i)≠[]
proof(rule ccontr)
  assume ¬ (∀ i∈I. set-to-list (S i) ≠ [])
  hence ∃ i∈I. set-to-list (S i) = [] by auto
  hence ∃ i∈I. set(set-to-list (S i)) = {} by auto
  then obtain i where i: i∈I and  set (set-to-list (S i)) = {} by auto
  thus False using list-to-set[of S i]  assms by auto
qed

lemma  at-least-subset-indices:
  assumes ∀ i∈I. (S i)≠{} and ∀ i∈I. finite (S i)
  shows  indices-set-formulas (ℱ S I) ⊆ I
proof
  fix i
  assume hip: i ∈ indices-set-formulas (ℱ S I) show  i ∈ I
  proof−
    have i ∈ (⋃F∈(ℱ S I). indices-formula F) using hip
      by(unfold indices-set-formulas-def,auto)
    hence ∃ F ∈ (ℱ S I). i ∈ indices-formula F by auto
    then obtain F where F∈(ℱ S I) and i: i ∈ indices-formula F by auto
    hence ∃  k∈I. F = disjunction-atomic (set-to-list (S k)) k
      by (unfold ℱ-def, auto)
    then obtain k where
    k: k∈I and F = disjunction-atomic (set-to-list (S k)) k by auto
    hence indices-formula F = {k}
      using assms  nonempty-set-list[of I S]
      indices-disjunction[OF ‹F = disjunction-atomic (set-to-list (S k))  k›]
      by auto
    hence k = i using i by auto
```

thus *?thesis* **using** *k* **by** *auto*
qed
qed

lemma *at-most-subset-indices*:
shows *indices-set-formulas* $(\mathcal{G}\ S\ I) \subseteq I$
proof
fix *i*
assume *hip*: $i \in$ *indices-set-formulas* $(\mathcal{G}\ S\ I)$ **show** $i \in I$
proof−
have $i \in (\bigcup F\in(\mathcal{G}\ S\ I).$ *indices-formula* $F)$ **using** *hip*
by(*unfold indices-set-formulas-def*,*auto*)
hence $\exists F\in(\mathcal{G}\ S\ I).\ i \in$ *indices-formula* F **by** *auto*
then obtain F **where** $F\in(\mathcal{G}\ S\ I)$ **and** i: $i \in$ *indices-formula* F
by *auto*
hence $\exists x\ y\ j.\ x\in(S\ j) \wedge y\in(S\ j) \wedge x\neq y \wedge j\in I \wedge F =$
$\neg.(atom\ (j,\ x) \wedge.\ atom(j,y))$
by (*unfold* $\mathcal{G}$*-def*, *auto*)
then obtain $x\ y\ j$ **where** $x\in(S\ j) \wedge y\in(S\ j) \wedge x\neq y \wedge j\in I$
and $F = \neg.(atom\ (j,\ x) \wedge.\ atom(j,y))$
by *auto*
hence *indices-formula* $F = \{j\} \wedge j\in I$ **by** *auto*
thus $i \in I$ **using** *i* **by** *auto*
qed
qed

lemma *different-subset-indices*:
shows *indices-set-formulas* $(\mathcal{H}\ S\ I) \subseteq I$
proof
fix *i*
assume *hip*: $i \in$ *indices-set-formulas* $(\mathcal{H}\ S\ I)$ **show** $i \in I$
proof−
have $i \in (\bigcup F\in(\mathcal{H}\ S\ I).$ *indices-formula* $F)$ **using** *hip*
by(*unfold indices-set-formulas-def*,*auto*)
hence $\exists F\in(\mathcal{H}\ S\ I)\ .\ i \in$ *indices-formula* F **by** *auto*
then obtain F **where** $F\in(\mathcal{H}\ S\ I)$ **and** i: $i \in$ *indices-formula* F
by *auto*
hence $\exists\ x\ j\ k.\ x \in (S\ j) \cap (S\ k) \wedge (j\in I \wedge k\in I \wedge j\neq k) \wedge F =$
$\neg.(atom\ (j,x) \wedge.\ atom(k,x))$
by (*unfold* $\mathcal{H}$*-def*, *auto*)
then obtain $x\ j\ k$
where $(j\in I \wedge k\in I \wedge j\neq k) \wedge F = \neg.(atom\ (j,\ x) \wedge.\ atom(k,x))$
by *auto*
hence u: $j\in I$ **and** v: $k\in I$ **and** *indices-formula* $F = \{j,k\}$
by *auto*
hence $i=j \vee i=k$ **using** *i* **by** *auto*
thus $i \in I$ **using** *u v* **by** *auto*
qed
qed

lemma *indices-union-sets*:
shows *indices-set-formulas*$(A \cup B)$ = (*indices-set-formulas* A) $\cup$ (*indices-set-formulas* B)
by(*unfold indices-set-formulas-def*, *auto*)

lemma *at-least-subset-subset-indices1*:
assumes $F\in(\mathcal{F}\ S\ I)$
shows (*indices-formula* F) $\subseteq$ (*indices-set-formulas* $(\mathcal{F}\ S\ I)$)
proof
fix i
assume *hip*: $i \in$ *indices-formula* F
show $i \in$ *indices-set-formulas* $(\mathcal{F}\ S\ I)$
proof−
have $\exists F.\ F\in(\mathcal{F}\ S\ I) \wedge i \in$ *indices-formula* F **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold indices-set-formulas-def*, *auto*)
qed
qed

lemma *at-most-subset-subset-indices1*:
assumes $F\in(\mathcal{G}\ S\ I)$
shows (*indices-formula* F) $\subseteq$ (*indices-set-formulas* $(\mathcal{G}\ S\ I)$)
proof
fix i
assume *hip*: $i \in$ *indices-formula* F
show $i \in$ *indices-set-formulas* $(\mathcal{G}\ S\ I)$
proof−
have $\exists F.\ F\in(\mathcal{G}\ S\ I) \wedge i \in$ *indices-formula* F **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold indices-set-formulas-def*, *auto*)
qed
qed

lemma *different-subset-indices1*:
assumes $F\in(\mathcal{H}\ S\ I)$
shows (*indices-formula* F) $\subseteq$ (*indices-set-formulas* $(\mathcal{H}\ S\ I)$)
proof
fix i
assume *hip*: $i \in$ *indices-formula* F
show $i \in$ *indices-set-formulas* $(\mathcal{H}\ S\ I)$
proof−
have $\exists F.\ F\in(\mathcal{H}\ S\ I) \wedge i \in$ *indices-formula* F **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold indices-set-formulas-def*, *auto*)
qed
qed

lemma *all-subset-indices*:
assumes $\forall i\in I.(S\ i)\neq\{\}$ **and** $\forall i\in I.$ *finite*$(S\ i)$
shows *indices-set-formulas* $(\mathcal{T}\ S\ I) \subseteq I$
proof

fix *i*
assume *hip*: $i \in$ *indices-set-formulas* $(\mathcal{T}\ S\ I)$ **show** $i \in I$
proof−
have $i \in$ *indices-set-formulas* $((\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I)\ \cup (\mathcal{H}\ S\ I))$
using *hip* **by** (*unfold* $\mathcal{T}$*-def*,*auto*)
hence $i \in$ *indices-set-formulas* $((\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I)) \cup$
indices-set-formulas$(\mathcal{H}\ S\ I)$
using *indices-union-sets*[*of* $(\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I)$] **by** *auto*
hence $i \in$ *indices-set-formulas* $((\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I)) \vee$
$i \in$ *indices-set-formulas*$(\mathcal{H}\ S\ I)$
by *auto*
thus *?thesis*
proof(*rule disjE*)
assume *hip*: $i \in$ *indices-set-formulas* $(\mathcal{F}\ S\ I \cup \mathcal{G}\ S\ I)$
hence $i \in (\bigcup F \in (\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I).$ *indices-formula* $F)$
by(*unfold indices-set-formulas-def*, *auto*)
then obtain F
where F: $F \in (\mathcal{F}\ S\ I) \cup (\mathcal{G}\ S\ I)$ **and** i: $i \in$ *indices-formula* F **by** *auto*
from F **have** (*indices-formula* F) $\subseteq$ (*indices-set-formulas* $(\mathcal{F}\ S\ I)$)
$\vee$ *indices-formula* $F \subseteq$ (*indices-set-formulas* $(\mathcal{G}\ S\ I)$)
using *at-least-subset-subset-indices1 at-most-subset-subset-indices1* **by** *blast*
hence $i \in$ *indices-set-formulas* $(\mathcal{F}\ S\ I) \vee$
$i \in$ *indices-set-formulas* $(\mathcal{G}\ S\ I)$
using i **by** *auto*
thus $i \in I$
using *assms at-least-subset-indices*[*of I S*] *at-most-subset-indices*[*of S I*] **by**
auto
next
assume $i \in$ *indices-set-formulas* $(\mathcal{H}\ S\ I)$
hence
$i \in (\bigcup F \in (\mathcal{H}\ S\ I).$ *indices-formula* $F)$
by(*unfold indices-set-formulas-def*, *auto*)
then obtain F **where** F: $F \in (\mathcal{H}\ S\ I)$ **and** i: $i \in$ *indices-formula* F
by *auto*
from F **have** (*indices-formula* F) $\subseteq$ (*indices-set-formulas* $(\mathcal{H}\ S\ I)$)
using *different-subset-indices1* **by** *blast*
hence $i \in$ *indices-set-formulas* $(\mathcal{H}\ S\ I)$ **using** i **by** *auto*
thus $i \in I$ **using** *different-subset-indices*[*of S I*]
by *auto*
qed
qed
qed

lemma *inclusion-indices*:
assumes $S \subseteq H$
shows *indices-set-formulas* $S \subseteq$ *indices-set-formulas* H
proof
fix *i*
assume $i \in$ *indices-set-formulas* S

hence ∃ *F*. *F* ∈ *S* ∧ *i* ∈ *indices-formula F*
by(*unfold indices-set-formulas-def*, *auto*)
hence ∃ *F*. *F* ∈ *H* ∧ *i* ∈ *indices-formula F* **using** *assms* **by** *auto*
thus *i* ∈ *indices-set-formulas H*
by(*unfold indices-set-formulas-def*, *auto*)
qed

lemma *indices-subset-formulas*:
assumes ∀ *i*∈*I*.(*S i*)≠{} **and** ∀ *i*∈*I*. *finite*(*S i*) **and** *A* ⊆ (𝒯 *S I*)
shows (*indices-set-formulas A*) ⊆ *I*
proof−
have (*indices-set-formulas A*) ⊆ (*indices-set-formulas* (𝒯 *S I*))
using *assms*(*3*) *inclusion-indices* **by** *auto*
thus *?thesis* **using** *assms*(*1*−*2*) *all-subset-indices*[*of I S*] **by** *auto*
qed

lemma *To-subset-all-its-indices*:
assumes ∀ *i*∈*I*. (*S i*)≠{} **and** ∀ *i*∈*I*. *finite* (*S i*) **and** *To* ⊆ (𝒯 *S I*)
shows *To* ⊆ (𝒯 *S* (*indices-set-formulas To*))
proof
fix *F*
assume *hip*: *F* ∈ *To*
hence *F* ∈ (𝒯 *S I*) **using** *assms*(*3*) **by** *auto*
hence *F* ∈ (ℱ *S I*) ∪ (𝒢 *S I*) ∪ (ℋ *S I*) **by**(*unfold* 𝒯*-def*,*auto*)
hence *F* ∈ (ℱ *S I*) ∨ *F* ∈ (𝒢 *S I*) ∨ *F* ∈ (ℋ *S I*) **by** *auto*
thus *F*∈(𝒯 *S* (*indices-set-formulas To*))
proof(*rule disjE*)
assume *F* ∈ (ℱ *S I*)
hence ∃ *i*∈*I*. *F* = *disjunction-atomic* (*set-to-list* (*S i*)) *i*
by(*unfold* ℱ*-def*,*auto*)
then obtain *i*
where *i*: *i*∈*I* **and** *F*: *F* = *disjunction-atomic* (*set-to-list* (*S i*)) *i*
by *auto*
hence *indices-formula F* = {*i*}
using
assms(*1*−*2*) *nonempty-set-list*[*of I S*] *indices-disjunction*[*of F* (*set-to-list* (*S*
i)) *i*]
by *auto*
hence *i*∈(*indices-set-formulas To*) **using** *hip*
by(*unfold indices-set-formulas-def*,*auto*)
hence *F*∈(ℱ *S* (*indices-set-formulas To*))
using *F* **by**(*unfold* ℱ*-def*,*auto*)
thus *F*∈(𝒯 *S* (*indices-set-formulas To*))
by(*unfold* 𝒯*-def*,*auto*)
next
assume *F* ∈ (𝒢 *S I*) ∨ *F* ∈ (ℋ *S I*)
thus *?thesis*
proof(*rule disjE*)
assume *F* ∈ (𝒢 *S I*)

hence ∃ *x*.∃ *y*.∃ *i*. *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*)) ∧ *x*∈(*S i*) ∧
y∈(*S i*) ∧ *x*≠*y* ∧ *i*∈*I*
by(*unfold 𝒢-def*, *auto*)
then obtain *x y i*
where *F1*: *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*)) **and**
F2: *x*∈(*S i*) ∧ *y*∈(*S i*) ∧ *x*≠*y* ∧ *i*∈*I*
by *auto*
hence *indices-formula F* = {*i*} **by** *auto*
hence *i*∈(*indices-set-formulas To*) **using** *hip*
by(*unfold indices-set-formulas-def*,*auto*)
hence *F*∈(*𝒢 S* (*indices-set-formulas To*))
using *F1 F2* **by**(*unfold 𝒢-def*,*auto*)
thus *F*∈(*𝒯 S* (*indices-set-formulas To*)) **by**(*unfold 𝒯-def*,*auto*)
next
assume *F* ∈ (*ℋ S I*)
hence ∃ *x*.∃ *i*.∃ *j*. *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*j*,*x*)) ∧
x ∈ (*S i*) ∩ (*S j*) ∧ (*i*∈*I* ∧ *j*∈*I* ∧ *i*≠*j*)
by(*unfold ℋ-def*, *auto*)
then obtain *x i j*
where *F3*: *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*j*,*x*)) **and**
F4: *x* ∈ (*S i*) ∩ (*S j*) ∧ (*i*∈*I* ∧ *j*∈*I* ∧ *i*≠*j*)
by *auto*
hence *indices-formula F* = {*i*,*j*} **by** *auto*
hence *i*∈(*indices-set-formulas To*) ∧ *j*∈(*indices-set-formulas To*)
using *hip* **by**(*unfold indices-set-formulas-def*,*auto*)
hence *F*∈(*ℋ S* (*indices-set-formulas To*))
using *F3 F4* **by**(*unfold ℋ-def*,*auto*)
thus *F*∈(*𝒯 S* (*indices-set-formulas To*)) **by**(*unfold 𝒯-def*,*auto*)
qed
qed
qed

lemma *all-nonempty-sets*:
assumes ∀ *i*∈*I*. (*S i*)≠{} **and** ∀ *i*∈*I*. *finite* (*S i*) **and** *A* ⊆ (*𝒯 S I*)
shows ∀ *i*∈(*indices-set-formulas A*). (*S i*)≠{}
proof−
have (*indices-set-formulas A*)⊆*I*
using *assms*(*1*−*3*) *indices-subset-formulas*[*of I S A*] **by** *auto*
thus *?thesis* **using** *assms*(*1*) **by** *auto*
qed

lemma *all-finite-sets*:
assumes ∀ *i*∈*I*. (*S i*)≠{} **and** ∀ *i*∈*I*. *finite* (*S i*) **and** *A* ⊆ (*𝒯 S I*)
shows ∀ *i*∈(*indices-set-formulas A*). *finite* (*S i*)
proof−
have (*indices-set-formulas A*)⊆*I*
using *assms*(*1*−*3*) *indices-subset-formulas*[*of I S A*] **by** *auto*
thus ∀ *i*∈(*indices-set-formulas A*). *finite* (*S i*) **using** *assms*(*2*) **by** *auto*
qed

lemma *all-nonempty-sets1*:
  assumes $\forall J \subseteq I.$ *finite* $J \longrightarrow$ *card* $J \leq$ *card* $(\bigcup (S \text{ ` } J))$
  shows $\forall i \in I.$ $(S\ i) \neq \{\}$ **using** *assms* **by** *auto*

lemma *system-distinct-representatives-finite*:
  assumes
  $\forall i \in I.$ $(S\ i) \neq \{\}$ **and** $\forall i \in I.$ *finite* $(S\ i)$ **and** $To \subseteq (\mathcal{T}\ S\ I)$ **and** *finite To*
   and $\forall J \subseteq$(*indices-set-formulas To*). *card* $J \leq$ *card* $(\bigcup (S \text{ ` } J))$
 shows $\exists R.$ *system-representatives S* (*indices-set-formulas To*) *R*
proof−
  have *1*: *finite* (*indices-set-formulas To*)
    using *assms*(*4*) *finite-set-indices* **by** *auto*
  have $\forall i \in$(*indices-set-formulas To*). *finite* $(S\ i)$
    using *all-finite-sets assms*(*1*−*3*) **by** *auto*
  hence $\exists R.$ ($\forall i \in$(*indices-set-formulas To*). $R\ i \in S\ i$) $\wedge$
            inj-on R (*indices-set-formulas To*)
    using *1 assms*(*5*) *marriage-HV*[*of* (*indices-set-formulas To*) *S*] **by** *auto*
  then obtain *R*
    where *R*: ($\forall i \in$(*indices-set-formulas To*). $R\ i \in S\ i$) $\wedge$
            inj-on R (*indices-set-formulas To*) **by** *auto*
  thus *?thesis* **by**(*unfold system-representatives-def*, *auto*)
qed

fun *Hall-interpretation* :: $('a \Rightarrow 'b\ set) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow (('a \times 'b) \Rightarrow$ *v-truth*) **where**
Hall-interpretation $A\ \mathcal{I}\ R = (\lambda(i,x).($*if* $i \in \mathcal{I} \wedge x \in (A\ i) \wedge (R\ i) = x$ *then Ttrue else Ffalse*))

lemma *t-v-evaluation-index*:
  assumes *t-v-evaluation* (*Hall-interpretation S I R*) (*atom* (*i*,*x*)) = *Ttrue*
  shows $(R\ i) = x$
proof(*rule ccontr*)
  assume $(R\ i) \neq x$ **hence** *t-v-evaluation* (*Hall-interpretation S I R*) (*atom* (*i*,*x*)) $\neq$ *Ttrue*
    by *auto*
  hence *t-v-evaluation* (*Hall-interpretation S I R*) (*atom* (*i*,*x*)) = *Ffalse*
  using *non-Ttrue*[*of Hall-interpretation S I R atom* (*i*,*x*)] **by** *auto*
  thus *False* **using** *assms* **by** *simp*
qed

lemma *distinct-elements-distinct-indices*:
  assumes $F = \neg.$(*atom* (*i*,*x*) $\wedge.$ *atom*(*i*,*y*)) **and** $x \neq y$
  shows *t-v-evaluation* (*Hall-interpretation S I R*) *F* = *Ttrue*
proof(*rule ccontr*)
  assume *t-v-evaluation* (*Hall-interpretation S I R*) $F \neq$ *Ttrue*
  hence
  t-v-evaluation (*Hall-interpretation S I R*) ($\neg.$(*atom* (*i*,*x*) $\wedge.$ *atom* (*i*, *y*))) $\neq$ *Ttrue*

using *assms*(*1*) **by** *auto*
hence
t-v-evaluation (*Hall-interpretation S I R*) (¬.(*atom* (*i*,*x*) ∧. *atom* (*i*, *y*))) = *Ffalse*
using
non-Ttrue[*of Hall-interpretation S I R* ¬.(*atom* (*i*,*x*) ∧. *atom* (*i*, *y*))]
by *auto*
hence *t-v-evaluation* (*Hall-interpretation S I R*) ((*atom* (*i*,*x*) ∧. *atom* (*i*, *y*))) = *Ttrue*
using
NegationValues1[*of Hall-interpretation S I R* (*atom* (*i*,*x*) ∧. *atom* (*i*, *y*))]
by *auto*
hence *t-v-evaluation* (*Hall-interpretation S I R*) (*atom* (*i*,*x*)) = *Ttrue* **and**
t-v-evaluation (*Hall-interpretation S I R*) (*atom* (*i*, *y*)) = *Ttrue*
using
ConjunctionValues[*of Hall-interpretation S I R atom* (*i*,*x*) *atom* (*i*, *y*)]
by *auto*
hence (*R i*)= *x* **and** (*R i*)= *y* **using** *t-v-evaluation-index* **by** *auto*
hence *x*=*y* **by** *auto*
thus *False* **using** *assms*(*2*) **by** *auto*
qed

lemma *same-element-same-index*:
assumes
F = ¬.(*atom* (*i*,*x*) ∧. *atom*(*j*,*x*)) **and** *i*∈*I* ∧ *j*∈*I* **and** *i*≠*j* **and** *inj-on R I*
shows *t-v-evaluation* (*Hall-interpretation S I R*) *F* = *Ttrue*
proof(*rule ccontr*)
assume *t-v-evaluation* (*Hall-interpretation S I R*) *F* ≠ *Ttrue*
hence *t-v-evaluation* (*Hall-interpretation S I R*) (¬.(*atom* (*i*,*x*) ∧. *atom* (*j*,*x*))) ≠ *Ttrue*
using *assms*(*1*) **by** *auto*
hence
t-v-evaluation (*Hall-interpretation S I R*) (¬.(*atom* (*i*,*x*) ∧. *atom* (*j*, *x*))) = *Ffalse*
using
non-Ttrue[*of Hall-interpretation S I R* ¬.(*atom* (*i*,*x*) ∧. *atom* (*j*, *x*))]
by *auto*
hence *t-v-evaluation* (*Hall-interpretation S I R*) ((*atom* (*i*,*x*) ∧. *atom* (*j*, *x*))) = *Ttrue*
using
NegationValues1[*of Hall-interpretation S I R* (*atom* (*i*,*x*) ∧. *atom* (*j*, *x*))]
by *auto*
hence *t-v-evaluation* (*Hall-interpretation S I R*) (*atom* (*i*,*x*)) = *Ttrue* **and**
t-v-evaluation (*Hall-interpretation S I R*) (*atom* (*j*, *x*)) = *Ttrue*
using *ConjunctionValues*[*of Hall-interpretation S I R atom* (*i*,*x*) *atom* (*j*,*x*)]
by *auto*
hence (*R i*)= *x* **and** (*R j*)= *x* **using** *t-v-evaluation-index* **by** *auto*
hence (*R i*) = (*R j*) **by** *auto*
hence *i*=*j* **using** ‹*i*∈*I* ∧ *j*∈*I*› ‹*inj-on R I*› **by**(*unfold inj-on-def*, *auto*)
thus *False* **using** ‹*i*≠*j*› **by** *auto*
qed

lemma *disjunctor-Ttrue-in-atomic-disjunctions*:
  assumes $x \in$ *set l* **and** *t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue*
  shows *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue*
proof−
  have $x \in$ *set l* $\Longrightarrow$ *t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue* $\Longrightarrow$
  t-v-evaluation I (*disjunction-atomic l i*) = *Ttrue*
  proof(*induct l*)
    case *Nil*
    then show *?case* **by** *auto*
  next
    case (*Cons a l*)
    then show *t-v-evaluation I* (*disjunction-atomic* (*a* # *l*) *i*) = *Ttrue*
    proof−
      have $x = a \lor x \neq a$ **by** *auto*
      thus *t-v-evaluation I* (*disjunction-atomic* (*a* # *l*) *i*) = *Ttrue*
      proof(*rule disjE*)
        assume $x = a$
          hence
          1:(*disjunction-atomic* (*a*#*l*) *i*) =
            (*atom* (*i*,*x*)) ∨. (*disjunction-atomic l i*)
          by *auto*
        have *t-v-evaluation I* ((*atom* (*i*,*x*)) ∨. (*disjunction-atomic l i*)) = *Ttrue*
        using *Cons*(*3*) **by**(*unfold t-v-evaluation-def*,*unfold v-disjunction-def*, *auto*)
        thus *?thesis* **using** *1* **by** *auto*
      next
        assume $x \neq a$
        hence $x \in$ *set l* **using** *Cons*(*2*) **by** *auto*
        hence *t-v-evaluation I* (*disjunction-atomic l i*) = *Ttrue*
          using *Cons*(*1*) *Cons*(*3*) **by** *auto*
        thus *?thesis*
          by(*unfold t-v-evaluation-def*,*unfold v-disjunction-def*, *auto*)
      qed
    qed
  qed
  thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *t-v-evaluation-disjunctions*:
  assumes *finite* (*S i*)
  and $x \in$ (*S i*) $\land$ *t-v-evaluation I* (*atom* (*i*,*x*)) = *Ttrue*
  and *F* = *disjunction-atomic* (*set-to-list* (*S i*)) *i*
  shows *t-v-evaluation I F* = *Ttrue*
proof−
  have *set* (*set-to-list* (*S i*)) = (*S i*)
  using *set-set-to-list assms*(*1*) **by** *auto*
  hence $x \in$ *set* (*set-to-list* (*S i*))
    using *assms*(*2*) **by** *auto*
  thus *t-v-evaluation I F* = *Ttrue*

```
    using assms(2−3) disjunctor-Ttrue-in-atomic-disjunctions by auto
qed

theorem SDR-satisfiable:
  assumes ∀ i∈𝓘. (A i) ≠ {}  and  ∀ i∈𝓘. finite (A i) and  X ⊆ (𝓣 A 𝓘)
  and  system-representatives A 𝓘 R
shows satisfiable X
proof−
  have satisfiable (𝓣 A 𝓘)
  proof−
    have inj-on R 𝓘 using assms(4) system-representatives-def[of A 𝓘 R] by auto
    have (Hall-interpretation A 𝓘 R) model (𝓣 A 𝓘)
    proof(unfold model-def)
      show ∀ F ∈ (𝓣 A 𝓘). t-v-evaluation (Hall-interpretation A 𝓘 R) F = Ttrue
      proof
        fix F assume F ∈ (𝓣 A 𝓘)
        show  t-v-evaluation (Hall-interpretation A 𝓘 R) F  = Ttrue
        proof−
          have F ∈ (𝓕 A 𝓘) ∪ (𝓖 A 𝓘) ∪ (𝓗 A 𝓘)
            using  ‹F ∈ (𝓣 A 𝓘)› assms(3)  by(unfold 𝓣-def,auto)
          hence  F ∈ (𝓕 A 𝓘) ∨ F ∈ (𝓖 A 𝓘) ∨ F ∈ (𝓗 A 𝓘) by auto
          thus ?thesis
          proof(rule disjE)
            assume F ∈ (𝓕 A 𝓘)
            hence ∃ i∈𝓘. F =  disjunction-atomic (set-to-list (A i)) i
              by(unfold 𝓕-def,auto)
            then obtain i
              where i: i∈𝓘 and F: F =  disjunction-atomic (set-to-list (A i)) i
              by auto
            have 1: finite (A i) using i  assms(2) by auto
            have 2:  i ∈ 𝓘 ∧ (R i) ∈ (A i)
              using i assms(4) by (unfold system-representatives-def, auto)
               hence t-v-evaluation (Hall-interpretation A 𝓘 R) (atom (i,(R i))) =
Ttrue
              by auto
            thus t-v-evaluation (Hall-interpretation A 𝓘 R) F  = Ttrue
              using 1 2 assms(4) F
            t-v-evaluation-disjunctions
            [of A i (R i) (Hall-interpretation A 𝓘 R) F]
              by auto
          next
            assume F ∈ (𝓖 A 𝓘) ∨ F ∈ (𝓗 A 𝓘)
            thus ?thesis
            proof(rule disjE)
              assume F ∈ (𝓖 A 𝓘)
              hence
            ∃ x.∃ y.∃ i. F = ¬.(atom (i,x) ∧. atom(i,y)) ∧ x∈(A i) ∧
              y∈(A i) ∧  x≠y ∧ i∈𝓘
                by(unfold 𝓖-def, auto)
```

then obtain *x y i*
where *F*: *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*))
and *x*∈(*A i*) ∧ *y*∈(*A i*) ∧ *x*≠*y* ∧ *i*∈ℐ
by *auto*
thus *t-v-evaluation* (*Hall-interpretation A ℐ R*) *F* = *Ttrue*
using ‹*inj-on R ℐ*› *distinct-elements-distinct-indices*[*of F i x y A ℐ R*]
by *auto*
next
assume *F* ∈ (ℋ *A ℐ*)
hence ∃ *x*.∃ *i*.∃ *j*. *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*j*,*x*)) ∧
x ∈ (*A i*) ∩ (*A j*) ∧ (*i*∈ℐ ∧ *j*∈ℐ ∧ *i*≠*j*)
by(*unfold* ℋ-*def*, *auto*)
then obtain *x i j*
where *F* = ¬.(*atom* (*i*,*x*) ∧. *atom*(*j*,*x*)) **and** (*i*∈ℐ ∧ *j*∈ℐ ∧ *i*≠*j*)
by *auto*
thus *t-v-evaluation* (*Hall-interpretation A ℐ R*) *F* = *Ttrue* **using**
‹*inj-on R ℐ*›
same-element-same-index[*of F i x j ℐ*] **by** *auto*
qed
qed
qed
qed
qed
thus *satisfiable* (𝒯 *A ℐ*) **by**(*unfold satisfiable-def*, *auto*)
qed
thus *satisfiable X* **using** *satisfiable-subset assms*(*3*) **by** *auto*
qed

lemma *finite-is-satisfiable*:
assumes
∀ *i*∈*I*. (*S i*)≠{} **and** ∀ *i*∈*I*. *finite* (*S i*) **and** *To* ⊆ (𝒯 *S I*) **and** *finite To*
and ∀ *J*⊆(*indices-set-formulas To*). *card J* ≤ *card* (⋃ (*S* ‘ *J*))
shows *satisfiable To*
proof−
have *0*: ∃ *R*. *system-representatives S* (*indices-set-formulas To*) *R*
using *assms system-distinct-representatives-finite*[*of I S To*] **by** *auto*
then obtain *R*
where *R*: *system-representatives S* (*indices-set-formulas To*) *R* **by** *auto*
have *1*: ∀ *i*∈(*indices-set-formulas To*). (*S i*)≠{}
using *assms*(*1*−*3*) *all-nonempty-sets* **by** *blast*
have *2*: ∀ *i*∈(*indices-set-formulas To*). *finite* (*S i*)
using *assms*(*1*−*3*) *all-finite-sets* **by** *blast*
have *3*: *To*⊆(𝒯 *S* (*indices-set-formulas To*))
using *assms*(*1*−*3*) *To-subset-all-its-indices*[*of I S To*] **by** *auto*
thus *satisfiable To*
using *1 2 3 0 SDR-satisfiable* **by** *auto*
qed

lemma *diag-nat*:

shows ∀ *y z*.∃ *x*. (*y*,*z*) = *diag x*
using *enumeration-natxnat* **by**(*unfold enumeration-def*,*auto*)

lemma *EnumFormulasHall*:
assumes ∃ *g*. *enumeration* (*g*:: *nat* ⇒′*a*) **and** ∃ *h*. *enumeration* (*h*:: *nat* ⇒′*b*)
shows ∃ *f*. *enumeration* (*f*:: *nat* ⇒(′*a* ×′*b*)*formula*)
proof−
from *assms*(*1*) **obtain** *g* **where** *e1*: *enumeration* (*g*:: *nat* ⇒′*a*) **by** *auto*
from *assms*(*2*) **obtain** *h* **where** *e2*: *enumeration* (*h*:: *nat* ⇒′*b*) **by** *auto*
have *enumeration* ((λ*m*.(*g*(*fst*(*diag m*)),(*h*(*snd*(*diag m*))))):: *nat* ⇒(′*a* ×′*b*))
proof(*unfold enumeration-def*)
show ∀ *y*::(′*a* × ′*b*). ∃ *m*. *y* = (*g* (*fst* (*diag m*)), *h* (*snd* (*diag m*)))
proof
fix *y*::(′*a* ×′*b*)
show ∃ *m*. *y* = (*g* (*fst* (*diag m*)), *h* (*snd* (*diag m*)))
proof−
have *y* = ((*fst y*), (*snd y*)) **by** *auto*
from *e1* **have** ∀ *w*::′*a*. ∃ *n1*. *w* = (*g n1*) **by**(*unfold enumeration-def*, *auto*)
hence ∃ *n1*. (*fst y*) = (*g n1*) **by** *auto*
then obtain *n1* **where** *n1*: (*fst y*) = (*g n1*) **by** *auto*
from *e2* **have** ∀ *w*::′*b*. ∃ *n2*. *w* = (*h n2*) **by**(*unfold enumeration-def*, *auto*)
hence ∃ *n2*. (*snd y*) = (*h n2*) **by** *auto*
then obtain *n2* **where** *n2*: (*snd y*) = (*h n2*) **by** *auto*
have ∃ *m*. (*n1*, *n2*) = *diag m* **using** *diag-nat* **by** *auto*
hence ∃ *m*. (*n1*, *n2*) = (*fst* (*diag m*), *snd* (*diag m*)) **by** *simp*
hence ∃ *m*.((*fst y*), (*snd y*)) = (*g*(*fst* (*diag m*)), *h*(*snd* (*diag m*)))
using *n1 n2* **by** *blast*
thus ∃ *m*. *y* = (*g* (*fst* (*diag m*)), *h*(*snd* (*diag m*))) **by** *auto*
qed
qed
qed
thus ∃ *f*. *enumeration* (*f*:: *nat* ⇒(′*a* ×′*b*)*formula*)
using *EnumerationFormulasP1* **by** *auto*
qed

theorem *all-formulas-satisfiable*:
fixes *S* :: (′*a*::*countable* ⇒ ′*b*::*countable set*) **and** *I* :: ′*a set*
assumes ∀ *i*∈(*I*::′*a set*). *finite* (*S i*) **and** ∀ *J*⊆*I*. *finite J* ⟶ *card J* ≤ *card* (⋃ (*S* ‘ *J*))
shows *satisfiable* (𝒯 *S I*)
proof−
have ∀ *A*. *A* ⊆ (𝒯 *S I*) ∧ (*finite A*) ⟶ *satisfiable A*
proof(*rule allI*, *rule impI*)
fix *A* **assume** *A* ⊆ (𝒯 *S I*) ∧ (*finite A*)
hence *hip1*: *A* ⊆ (𝒯 *S I*) **and** *hip2*: *finite A* **by** *auto*
show *satisfiable A*
proof −
have *0*: ∀ *i*∈*I*. (*S i*)≠{} **using** *assms*(*2*) *all-nonempty-sets1* **by** *auto*
hence *1*: (*indices-set-formulas A*)⊆*I*

```
      using assms(1) hip1 indices-subset-formulas[of I S A] by auto
    have 2: finite (indices-set-formulas A)
      using hip2 finite-set-indices by auto
    have 3: card (indices-set-formulas A) ≤ card(⋃ (S ‘(indices-set-formulas A)))
      using 1 2 assms(2) by auto
    have ∀ J⊆(indices-set-formulas A). card J ≤ card(⋃ (S ‘ J))
    proof(rule allI)
      fix J
      show J ⊆ indices-set-formulas A ⟶ card J ≤ card (⋃ (S ‘ J))
      proof(rule impI)
        assume hip: J⊆(indices-set-formulas A)
        hence 4: finite J
          using 2 rev-finite-subset by auto
        have J⊆I using hip 1 by auto
        thus card J ≤ card (⋃ (S ‘ J)) using 4 assms(2) by auto
      qed
    qed
    thus satisfiable A
      using 0 assms(1) hip1 hip2 finite-is-satisfiable[of I S A] by auto
  qed
 qed
 thus satisfiable (𝒯 S I)
   using Compactness-Theorem by auto
qed

fun SDR :: (('a × 'b) ⇒ v-truth) ⇒ ('a ⇒ 'b set) ⇒ 'a set ⇒ ('a ⇒'b )
 where
SDR M S I = (λi. (THE x. (t-v-evaluation M (atom (i,x)) = Ttrue) ∧ x∈(S i)))

lemma existence-representants:
 assumes i ∈ I and M model (ℱ S I) and finite(S i)
 shows ∃x. (t-v-evaluation M (atom (i,x)) = Ttrue) ∧ x ∈ (S i)
proof−
 from ‹i ∈ I›
 have (disjunction-atomic (set-to-list (S i)) i) ∈ (ℱ S I)
   by(unfold ℱ-def,auto)
 hence t-v-evaluation M (disjunction-atomic(set-to-list (S i)) i) = Ttrue
   using assms(2) model-def[of M ℱ S I] by auto
 hence 1: ∃x. x ∈ set (set-to-list (S i)) ∧ (t-v-evaluation M (atom (i,x)) = Ttrue)
   using t-v-evaluation-atom[of M (set-to-list (S i)) i] by auto
 thus ∃x. (t-v-evaluation M (atom (i,x)) = Ttrue) ∧ x ∈ (S i)
   using ‹finite(S i)› set-set-to-list[of (S i)] by auto
qed

lemma unicity-representants:
 shows ∀y.(x∈(S i) ∧ y∈(S i) ∧ x≠y ∧ i∈I) ⟶
        (¬.(atom (i,x) ∧. atom(i,y))∈ (𝒢 S I))
proof(rule allI)
 fix y
```

```
  show x∈(S i) ∧ y∈(S i) ∧  x≠y ∧ i∈I ⟶
       (¬.(atom (i,x) ∧. atom(i,y))∈ (𝒢 S I))
  proof(rule impI)
    assume x∈(S i) ∧ y∈(S i) ∧  x≠y ∧ i∈I
    thus  ¬.(atom (i,x) ∧. atom(i,y)) ∈ (𝒢 S I)
   by(unfold 𝒢-def, auto)
  qed
qed

lemma unicity-selection-representants:
 assumes i ∈ I and M model (𝒢 S I)
  shows  ∀ y.(x∈(S i) ∧ y∈(S i) ∧  x≠y ∧ i∈I) ⟶
  (t-v-evaluation M (¬.(atom (i,x) ∧. atom(i,y))) = Ttrue)
proof−
  have  ∀ y.(x∈(S i) ∧ y∈(S i) ∧  x≠y ∧ i∈I) ⟶
  (¬.(atom (i,x) ∧. atom(i,y))∈ (𝒢 S I))
    using unicity-representants[of x S i] by auto
  thus  ∀ y.(x∈(S i) ∧ y∈(S i) ∧  x≠y ∧ i∈I) ⟶
  (t-v-evaluation M (¬.(atom (i,x) ∧. atom(i,y))) = Ttrue)
    using assms(2)  model-def[of M 𝒢 S I] by blast
qed

lemma uniqueness-satisfaction:
  assumes t-v-evaluation M (atom (i,x)) = Ttrue ∧ x∈(S i) and
  ∀ y. y ∈ (S i) ∧ x≠y  ⟶  t-v-evaluation M (atom (i, y)) = Ffalse
shows ∀ z. t-v-evaluation M (atom (i, z)) = Ttrue ∧ z∈(S i) ⟶ z = x
proof(rule allI)
  fix z
  show t-v-evaluation M (atom (i, z)) = Ttrue ∧ z ∈ S i  ⟶ z = x
  proof(rule impI)
    assume hip: t-v-evaluation M (atom (i, z)) = Ttrue ∧ z ∈ (S i)
    show z = x
    proof(rule ccontr)
      assume 1: z ≠ x
      have  2: z ∈ (S i) using hip by auto
      hence  t-v-evaluation M (atom(i,z)) =  Ffalse using 1 assms(2) by auto
      thus False using hip by auto
    qed
  qed
qed

lemma uniqueness-satisfaction-in-Si:
  assumes t-v-evaluation M (atom (i,x)) = Ttrue ∧ x∈(S i) and
  ∀ y. y ∈ (S i) ∧ x≠y ⟶ (t-v-evaluation M (¬.(atom (i,x) ∧. atom(i,y))) =
Ttrue)
  shows ∀ y. y ∈ (S i)  ∧ x≠y  ⟶  t-v-evaluation M (atom (i, y)) = Ffalse
proof(rule allI, rule impI)
  fix y
  assume hip: y ∈ S i ∧ x ≠ y
```

show *t-v-evaluation M* (*atom* (*i*, *y*)) = *Ffalse*
proof(*rule ccontr*)
assume *t-v-evaluation M* (*atom* (*i*, *y*)) ≠ *Ffalse*
hence *t-v-evaluation M* (*atom* (*i*, *y*)) = *Ttrue* **using** *Bivaluation* **by** *blast*
hence *1*: *t-v-evaluation M* (*atom* (*i*,*x*) ∧. *atom*(*i*,*y*)) = *Ttrue*
using *assms*(*1*) *v-conjunction-def* **by** *auto*
have *t-v-evaluation M* (¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*))) = *Ttrue*
using *hip assms*(*2*) **by** *auto*
hence *t-v-evaluation M* (*atom* (*i*,*x*) ∧. *atom*(*i*,*y*)) = *Ffalse*
using *NegationValues2* **by** *blast*
thus *False* **using** *1* **by** *auto*
qed
qed

lemma *uniqueness-aux1*:
assumes *t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue* ∧ *x*∈(*S i*)
and ∀ *y*. *y* ∈ (*S i*) ∧ *x*≠*y* ⟶ (*t-v-evaluation M* (¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*))) = *Ttrue*)
shows ∀ *z*. *t-v-evaluation M* (*atom* (*i*, *z*)) = *Ttrue* ∧ *z*∈(*S i*) ⟶ *z* = *x*
using *assms uniqueness-satisfaction-in-Si*[*of M i x*] *uniqueness-satisfaction*[*of M i x*] **by** *blast*

lemma *uniqueness-aux2*:
assumes *t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue* ∧ *x*∈(*S i*) **and**
(⋀*z*.(*t-v-evaluation M* (*atom* (*i*, *z*)) = *Ttrue* ∧ *z*∈(*S i*)) ⟹ *z* = *x*)
shows (*THE a*. (*t-v-evaluation M* (*atom* (*i*,*a*)) = *Ttrue*) ∧ *a*∈(*S i*)) = *x*
using *assms* **by**(*rule the-equality*)

lemma *uniqueness-aux*:
assumes *t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue* ∧ *x*∈(*S i*) **and**
∀ *y*. *y* ∈ (*S i*) ∧ *x*≠*y* ⟶ (*t-v-evaluation M* (¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*))) = *Ttrue*)
shows (*THE a*. (*t-v-evaluation M* (*atom* (*i*,*a*)) = *Ttrue*) ∧ *a*∈(*S i*)) = *x*
using *assms uniqueness-aux1*[*of M i x*] *uniqueness-aux2*[*of M i x*] **by** *blast*

lemma *function-SDR*:
assumes *i* ∈ *I* **and** *M model* (𝓕 *S I*) **and** *M model* (𝓖 *S I*) **and** *finite*(*S i*)
shows ∃!*x*. (*t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue*) ∧ *x* ∈ (*S i*) ∧ (*SDR M S I i*) = *x*
proof−
have ∃ *x*. (*t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue*) ∧ *x* ∈ (*S i*)
using *assms*(*1*−*2*,*4*) *existence-representants* **by** *auto*
then obtain *x* **where** *x*: (*t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue*) ∧ *x* ∈ (*S i*)
by *auto*
moreover
have ∀ *y*.(*x*∈(*S i*) ∧ *y*∈(*S i*) ∧ *x*≠*y* ∧ *i*∈*I*) ⟶
(*t-v-evaluation M* (¬.(*atom* (*i*,*x*) ∧. *atom*(*i*,*y*))) = *Ttrue*)
using *assms*(*1*,*3*) *unicity-selection-representants*[*of i I M S*] **by** *auto*
hence (*THE a*. (*t-v-evaluation M* (*atom* (*i*,*a*)) = *Ttrue*) ∧ *a*∈(*S i*)) = *x*

using *x* ‹*i* ∈ *I*› *uniqueness-aux*[*of M i x*] **by** *auto*
hence *SDR M S I i* = *x* **by** *auto*
hence (*t-v-evaluation M* (*atom* (*i*,*x*)) = *Ttrue* ∧ *x* ∈ (*S i*)) ∧ *SDR M S I i* = *x*
using *x* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *aux-for-ℋ-formulas*:
assumes
(*t-v-evaluation M* (*atom* (*i*,*a*)) = *Ttrue*) ∧ *a* ∈ (*S i*)
and (*t-v-evaluation M* (*atom* (*j*,*b*)) = *Ttrue*) ∧ *b* ∈ (*S j*)
and *i*∈*I* ∧ *j*∈*I* ∧ *i*≠*j*
and (*a* ∈ (*S i*) ∩ (*S j*) ∧ *i*∈*I* ∧ *j*∈*I* ∧ *i*≠*j* ⟶
(*t-v-evaluation M* (¬.(*atom* (*i*,*a*) ∧. *atom*(*j*,*a*))) = *Ttrue*))
shows *a* ≠ *b*
proof(*rule ccontr*)
assume ¬ *a* ≠ *b*
hence *hip*: *a*=*b* **by** *auto*
hence *t-v-evaluation M* (*atom* (*i*, *a*)) = *Ttrue* **and** *t-v-evaluation M* (*atom* (*j*, *a*)) = *Ttrue*
using *assms* **by** *auto*
hence *t-v-evaluation M* (*atom* (*i*, *a*) ∧. *atom*(*j*,*a*)) = *Ttrue* **using** *v-conjunction-def*
by *auto*
hence *t-v-evaluation M* (¬.(*atom* (*i*, *a*) ∧. *atom*(*j*,*a*))) = *Ffalse*
using *v-negation-def* **by** *auto*
moreover
have *a* ∈ (*S i*) ∩ (*S j*) **using** *hip assms*(*1*−*2*) **by** *auto*
hence *t-v-evaluation M* (¬.(*atom* (*i*, *a*) ∧. *atom*(*j*, *a*))) = *Ttrue*
using *assms*(*3*−*4*) **by** *auto*
ultimately show *False* **by** *auto*
qed

lemma *model-of-all*:
assumes *M model* (𝒯 *S I*)
shows *M model* (ℱ *S I*) **and** *M model* (𝒢 *S I*) **and** *M model* (ℋ *S I*)
proof(*unfold model-def*)
show ∀ *F*∈ℱ *S I*. *t-v-evaluation M F* = *Ttrue*
proof
fix *F*
assume *F*∈ (ℱ *S I*) **hence** *F*∈(𝒯 *S I*) **by**(*unfold 𝒯-def*, *auto*)
thus *t-v-evaluation M F* = *Ttrue* **using** *assms* **by**(*unfold model-def*, *auto*)
qed
next
show ∀ *F*∈(𝒢 *S I*). *t-v-evaluation M F* = *Ttrue*
proof
fix *F*
assume *F*∈(𝒢 *S I*) **hence** *F*∈(𝒯 *S I*) **by**(*unfold 𝒯-def*, *auto*)
thus *t-v-evaluation M F* = *Ttrue* **using** *assms* **by**(*unfold model-def*, *auto*)
qed

```
next
  show ∀F∈(H S I). t-v-evaluation M F = Ttrue
  proof
    fix F
    assume F∈(H S I) hence F∈(T S I) by(unfold T-def, auto)
    thus t-v-evaluation M F = Ttrue using assms by(unfold model-def, auto)
  qed
qed

lemma sets-have-distinct-representants:
  assumes
  hip1: i∈I and  hip2: j∈I and  hip3: i≠j and  hip4: M model (T S I)
  and hip5: finite(S i) and  hip6: finite(S j)
  shows  SDR M S I i  ≠  SDR M S I j
proof−
  have 1: M model F S I and 2:  M model G S I
    using hip4 model-of-all by auto
  hence ∃!x. (t-v-evaluation M (atom (i,x)) = Ttrue) ∧ x ∈ (S i) ∧  SDR M S I
i = x
    using  hip1  hip4  hip5 function-SDR[of i I M S] by auto
  then obtain x where
  x1: (t-v-evaluation M (atom (i,x)) = Ttrue) ∧ x ∈ (S i) and x2: SDR M S I i
= x
    by auto
  have ∃!y. (t-v-evaluation M (atom (j,y)) = Ttrue) ∧ y ∈ (S j) ∧ SDR M S I j
= y
  using 1 2  hip2  hip4  hip6 function-SDR[of j I M S] by auto
  then obtain y where
  y1: (t-v-evaluation M (atom (j,y)) = Ttrue) ∧ y ∈ (S j) and y2: SDR M S I j
= y
    by auto
  have (x ∈ (S i) ∩ (S j) ∧ i∈I ∧ j∈I ∧ i≠j) −→
  (¬.(atom (i,x) ∧. atom(j,x))∈ (H S I))
    by(unfold  H-def, auto)
  hence (x ∈ (S i) ∩ (S j) ∧ i∈I ∧ j∈I ∧ i≠j) −→
  (¬.(atom (i,x) ∧. atom(j,x)) ∈ (T S I))
    by(unfold  T-def, auto)
  hence (x ∈ (S i) ∩ (S j) ∧ i∈I ∧ j∈I ∧ i≠j) −→
  (t-v-evaluation M (¬.(atom (i,x) ∧. atom(j,x))) = Ttrue)
    using hip4 model-def[of M  T  S I] by auto
  hence x ≠ y using x1 y1 assms(1−3) aux-for-H-formulas[of M i x  S  j y I]
    by auto
  thus ?thesis using x2 y2 by auto
qed

lemma satisfiable-representant:
  assumes satisfiable (T S I) and ∀i∈I. finite (S i)
  shows ∃R. system-representatives S I R
proof−
```

```
  from assms have ∃M. M model (𝒯 S I)  by(unfold satisfiable-def)
  then obtain M where M: M model (𝒯 S I) by auto
  hence  system-representatives S I (SDR M S I)
  proof(unfold system-representatives-def)
    show (∀i∈I. (SDR M S I i) ∈ (S i)) ∧ inj-on (SDR M S I) I
    proof(rule conjI)
      show ∀i∈I. (SDR M S I i) ∈ (S i)
      proof
        fix i
        assume i: i ∈ I
        have M model ℱ S I and 2: M model 𝒢 S I using M model-of-all
          by auto
        thus (SDR M S I i) ∈ (S i)
          using i M assms(2) model-of-all[of M S I]
                function-SDR[of i I M S ] by auto
      qed
    next
      show inj-on (SDR M S I) I
      proof(unfold  inj-on-def)
        show ∀i∈I. ∀j∈I. SDR M S I i = SDR M S I j ⟶ i = j
        proof
          fix i
          assume 1:  i ∈ I
          show ∀j∈I. SDR M S I i = SDR M S I j ⟶ i = j
          proof
            fix j
            assume 2:  j ∈ I
            show SDR M S I i = SDR M S I j ⟶ i = j
            proof(rule ccontr)
              assume  ¬ (SDR M S I i = SDR M S I j ⟶ i = j)
              hence 5:  SDR M S I i = SDR M S I j and 6:  i≠ j by auto
              have  3: finite(S i) and 4: finite(S j) using 1 2  assms(2) by auto
              have SDR M S I i ≠ SDR M S I j
                 using 1 2 3 4 6 M sets-have-distinct-representants[of i I j M S] by
auto
              thus False  using 5 by auto
            qed
          qed
        qed
      qed
    qed
  qed
  thus  ∃R. system-representatives S I R by auto
qed

theorem Hall:
  fixes S :: ('a::countable ⇒ 'b::countable set) and I :: 'a set
  assumes Finite: ∀i∈I. finite (S i)
  and Marriage: ∀J⊆I. finite J ⟶  card J ≤ card (⋃ (S ‘ J))
```

```
 shows ∃R. system-representatives S I R
proof−
  have satisfiable (𝒯 S I) using assms all-formulas-satisfiable[of I] by auto
  thus ?thesis using Finite Marriage satisfiable-representant[of S I] by auto
qed

theorem marriage-necessity:
  fixes A :: 'a ⇒ 'b set and I :: 'a set
  assumes ∀ i∈I. finite (A i)
  and ∃R. (∀ i∈I. R i ∈ A i) ∧ inj-on R I (is ∃R. ?R R A & ?inj R A)
  shows ∀ J⊆I. finite J ⟶ card J ≤ card (⋃(A ‘ J))
proof clarify
  fix J
  assume J ⊆ I and 1: finite J
  show card J ≤ card (⋃(A ‘ J))
  proof−
    from assms(2) obtain R where ?R R A and ?inj R A by auto
    have inj-on R J by(rule subset-inj-on[OF ‹?inj R A› ‹J⊆I›])
    moreover have (R ‘ J) ⊆ (⋃(A ‘ J)) using ‹J⊆I› ‹?R R A› by auto
    moreover have finite (⋃(A ‘ J)) using ‹J⊆I› 1 assms
      by auto
    ultimately show ?thesis by (rule card-inj-on-le)
  qed
qed

end

theory Hall-Theorem-Graphs
  imports
          Background-on-graphs
          HOL−Library.Countable-Set
          Hall-Theorem


begin
```

8 Hall Theorem for countable (infinite) Graphs

This section formalizes Hall Theorem for countable infinite Graphs ([5]). The proof applied a proof of Hall's theorem for countable infinite families of sets, obtained by the authors directly from the compactness theorem for propositional logic. The proof is based on Smullyan's approach given in the third chapter of his influential textbook on mathematical logic [3], based on Henkin's model existence theorem. It follows the impeccable presentation in Fitting's textbook [1].

```
definition dirBD-to-Hall::
  ('a,'b) pre-digraph ⇒ 'a set ⇒ 'a set ⇒ 'a set ⇒ ('a ⇒ 'a set) ⇒ bool
```

```
  where
  dirBD-to-Hall G X Y I S ≡
  dir-bipartite-digraph G X Y ∧ I = X ∧ (∀ v∈I. (S v) = (neighbourhood G v))

theorem dir-BD-to-Hall:
  dirBD-perfect-matching G X Y E ⟶
   system-representatives (neighbourhood G) X (E-head G E)
proof(rule impI)
 assume dirBD-pm :dirBD-perfect-matching G X Y E
 show system-representatives (neighbourhood G) X (E-head G E)
 proof−
  have wS : dirBD-to-Hall G X Y X (neighbourhood G)
  using dirBD-pm
  by(unfold dirBD-to-Hall-def,unfold dirBD-perfect-matching-def,
     unfold dirBD-matching-def, auto)
  have arc: E ⊆ arcs G using dirBD-pm
   by(unfold dirBD-perfect-matching-def, unfold dirBD-matching-def,auto)
  have a: ∀ i. i ∈ X ⟶ E-head G E i ∈ neighbourhood G i
  proof(rule allI)
   fix i
   show i ∈ X ⟶ E-head G E i ∈ neighbourhood G i
   proof
    assume 1: i ∈ X
    show E-head G E i ∈ neighbourhood G i
    proof−
     have 2: ∃!e ∈ E. tail G e = i
     using 1 dirBD-pm Edge-unicity-in-dirBD-P-matching [of X G Y E ]
      by auto
     then obtain e where 3: e ∈ E ∧ tail G e = i by auto
    thus E-head G E i ∈ neighbourhood G i
     using  dirBD-pm 1 3 E-head-in-neighbourhood[of G X Y E e i]
     by (unfold dirBD-perfect-matching-def, auto)
    qed
   qed
  qed
  thus system-representatives (neighbourhood G) X (E-head G E)
  using a dirBD-pm dirBD-matching-inj-on [of G X Y E]
  by (unfold system-representatives-def, auto)
 qed
qed


lemma marriage-necessary-graph:
 assumes (dirBD-perfect-matching G X Y E) and ∀ i∈X. finite (neighbourhood
G i)
 shows ∀ J⊆X. finite J ⟶ (card J) ≤ card (⋃ (neighbourhood G ‘ J))
proof(rule allI, rule impI)
 fix J
```

assume *hip1*: $J \subseteq X$
show *finite* $J \longrightarrow$ *card* $J \leq$ *card* $(\bigcup$ (*neighbourhood* G ‘ J))
proof
assume *hip2*: *finite* J
show *card* $J \leq$ *card* $(\bigcup$ (*neighbourhood* G ‘ J))
proof−
have $\exists R. (\forall i{\in}X.\ R\ i \in$ *neighbourhood* $G\ i) \wedge$ *inj-on* $R\ X$
using *assms* *dir-BD-to-Hall*[*of* $G\ X\ Y\ E$]
by(*unfold system-representatives-def*, *auto*)
thus *?thesis* **using** *assms*(*2*) *marriage-necessity*[*of* X *neighbourhood* G] *hip1* *hip2* **by** *auto*
qed
qed
qed

lemma *neighbour3*:
fixes G :: ($'a$, $'b$) *pre-digraph* **and** X:: $'a$ *set*
assumes *dir-bipartite-digraph* $G\ X\ Y$ **and** $x{\in}X$
shows *neighbourhood* $G\ x = \{y\ |y.\ \exists e.\ e \in$ *arcs* $G \wedge ((x =$ *tail* $G\ e) \wedge (y =$ *head* $G\ e))\}$
proof
show *neighbourhood* $G\ x \subseteq \{y\ |y.\ \exists e.\ e \in$ *arcs* $G \wedge x =$ *tail* $G\ e \wedge y =$ *head* $G\ e\}$
proof
fix z
assume *hip*: $z \in$ *neighbourhood* $G\ x$
show $z \in \{y\ |y.\ \exists e.\ e \in$ *arcs* $G \wedge x =$ *tail* $G\ e \wedge y =$ *head* $G\ e\}$
proof−
have *neighbour* $G\ z\ x$ **using** *hip* **by**(*unfold neighbourhood-def*, *auto*)
hence $\exists e.\ e \in$ *arcs* $G \wedge ((z = ($*head* $G\ e) \wedge x =($*tail* $G\ e) \vee$
$((x = ($*head* $G\ e) \wedge z = ($*tail* $G\ e)))))$
using *assms* **by** (*unfold neighbour-def*, *auto*)
hence $\exists e.\ e \in$ *arcs* $G \wedge (z = ($*head* $G\ e) \wedge x =($*tail* $G\ e))$
using *assms*
by(*unfold dir-bipartite-digraph-def*, *unfold bipartite-digraph-def*, *unfold tails-def*, *blast*)
thus *?thesis* **by** *auto*
qed
qed
next
show $\{y\ |y.\ \exists e.\ e \in$ *arcs* $G \wedge x =$ *tail* $G\ e \wedge y =$ *head* $G\ e\} \subseteq$ *neighbourhood* $G\ x$
proof
fix z
assume *hip1*: $z \in \{y\ |y.\ \exists e.\ e \in$ *arcs* $G \wedge x =$ *tail* $G\ e \wedge y =$ *head* $G\ e\}$
thus $z \in$ *neighbourhood* $G\ x$
by(*unfold neighbourhood-def*, *unfold neighbour-def*, *auto*)
qed
qed

```
lemma perfect:
  fixes  G :: ('a, 'b) pre-digraph and X:: 'a set
  assumes dir-bipartite-digraph G X Y and system-representatives (neighbourhood G) X R
  shows  tails-set G {e |e. e ∈ (arcs G) ∧ ((tail G e) ∈ X ∧ (head G e) = R(tail G e))} = X
proof(unfold tails-set-def)
  let ?E = {e |e. e ∈ (arcs G) ∧ ((tail G e) ∈ X ∧ (head G e) = R (tail G e))}
  show  {tail G e |e. e ∈ ?E ∧ ?E ⊆ arcs G} = X
  proof
    show {tail G e |e. e ∈ ?E ∧ ?E ⊆ arcs G}⊆ X
    proof
      fix x
      assume hip1: x ∈ {tail G e |e. e ∈ ?E ∧ ?E ⊆ arcs G}
      show x∈X
      proof−
        have ∃ e. x = tail G e ∧ e ∈ ?E ∧ ?E ⊆ arcs G using hip1 by auto
        then obtain e where e: x = tail G e ∧ e ∈ ?E ∧ ?E ⊆ arcs G by auto
        thus x∈X
          using assms(1) by(unfold dir-bipartite-digraph-def, unfold tails-def, auto)
      qed
    qed
    next
    show X ⊆ {tail G e |e. e ∈ ?E ∧ ?E ⊆ arcs G}
    proof
      fix x
      assume hip2: x∈X
      show x∈{tail G e |e. e ∈ ?E ∧ ?E ⊆ arcs G}
      proof−
        have R (x) ∈ neighbourhood G x
          using assms(2) hip2 by (unfold system-representatives-def, auto)
        hence ∃ e. e∈ arcs G ∧ (x = tail G e ∧ R(x) = (head G e))
          using assms(1) hip2 neighbour3[of G  X Y] by auto
        moreover
        have  ?E ⊆ arcs G by auto
        ultimately show ?thesis
          using hip2 assms(1) by(unfold dir-bipartite-digraph-def, unfold tails-def, auto)
      qed
    qed
  qed
qed

lemma dirBD-matching:
  fixes  G :: ('a, 'b) pre-digraph and X:: 'a set
  assumes dir-bipartite-digraph G X Y and R: system-representatives (neighbourhood G) X R
  and  e1 ∈ arcs G ∧ tail G e1 ∈ X and e2 ∈ arcs G ∧ tail G e2 ∈ X
```

```
  and  R(tail G e1) = head G e1
  and  R(tail G e2) = head G e2
shows e1 ≠ e2 ⟶ head G e1 ≠ head G e2 ∧ tail G e1 ≠ tail G e2
proof
  assume hip: e1 ≠ e2
  show head G e1 ≠ head G e2 ∧ tail G e1 ≠ tail G e2
  proof−
    have  (e1 = e2) = (head G e1 = head G e2 ∧ tail G e1 = tail G e2)
      using assms(1)  assms(3−4)  by(unfold dir-bipartite-digraph-def, auto)
    hence 1:  tail G e1 = tail G e2 ⟶ head G e1 ≠ head G e2
      using hip assms(1)  by auto
    have  2:  tail G e1 = tail G e2 ⟶ head G e1 = head  G e2
      using  assms(1−2) assms(5−6)  by auto
    have 3: tail G e1 ≠ tail G e2
    proof(rule notI)
      assume ∗:  tail G e1 = tail G e2
      thus False using 1 2 by auto
    qed
    have 4:  tail G e1 ≠ tail G e2 ⟶ head G e1 ≠ head G e2
      proof
        assume ∗∗: tail G e1 ≠ tail G e2
        show  head G e1 ≠ head G e2
          using ∗∗ assms(3−6) R  inj-on-def[of R X]
          system-representatives-def[of (neighbourhood G) X R] by auto
      qed
    thus ?thesis using 3 by auto
  qed
qed

lemma marriage-sufficiency-graph:
  fixes  G :: ('a::countable, 'b::countable) pre-digraph and X:: 'a set
  assumes dir-bipartite-digraph G X Y  and ∀ i∈X. finite (neighbourhood G i)
  shows
  (∀ J⊆X. finite J ⟶ (card J) ≤ card (⋃ (neighbourhood G ` J))) ⟶
   (∃ E. dirBD-perfect-matching G X Y E)
proof(rule impI)
  assume hip: ∀ J⊆X. finite J ⟶ card J ≤ card (⋃ (neighbourhood G ` J))
  show ∃ E. dirBD-perfect-matching G X Y E
  proof−
    have ∃ R. system-representatives (neighbourhood G) X R
      using assms hip Hall[of X neighbourhood G] by auto
     then obtain R where R: system-representatives (neighbourhood G) X R by
auto
    let ?E = {e |e. e ∈ (arcs G) ∧ ((tail G e) ∈ X ∧ (head G e) = R (tail G e))}
    have dirBD-perfect-matching G X Y ?E
    proof(unfold dirBD-perfect-matching-def, rule conjI)
      show dirBD-matching G X Y ?E
      proof(unfold dirBD-matching-def, rule conjI)
        show dir-bipartite-digraph G X Y using assms(1) by auto
```

```
      next
        show ?E ⊆ arcs G ∧ (∀ e1∈?E. ∀ e2∈?E.
             e1 ≠ e2 ⟶ head G e1 ≠ head G e2 ∧ tail G e1 ≠ tail G e2)
        proof(rule conjI)
          show  ?E ⊆ arcs G  by auto
        next
          show ∀ e1∈?E. ∀ e2∈?E. e1 ≠ e2 ⟶ head G e1 ≠ head G e2 ∧ tail G
e1 ≠ tail G e2
          proof
            fix e1
            assume H1: e1 ∈ ?E
             show ∀ e2∈ ?E. e1 ≠ e2 ⟶ head G e1  ≠ head G e2 ∧ tail G e1 ≠
tail G e2
            proof
              fix e2
              assume H2: e2 ∈ ?E
              show  e1 ≠ e2 ⟶ head G e1 ≠ head G e2 ∧ tail G e1 ≠ tail G e2
              proof−
                have  e1 ∈ (arcs G) ∧ ((tail G e1) ∈ X ∧ (head G e1) = R (tail G
e1)) using H1 by auto
               hence 1:  e1 ∈ (arcs G) ∧ (tail G e1) ∈ X and 2:  R (tail G e1) =
(head G e1) by auto
                have  e2 ∈ (arcs G) ∧ ((tail G e2) ∈ X ∧ (head G e2) = R (tail G
e2)) using H2 by auto
               hence 3:  e2 ∈ (arcs G) ∧ (tail G e2) ∈ X and 4: R (tail G e2) =
(head G e2) by auto
              show ?thesis using assms(1) R  1 2 3 4 assms(1) dirBD-matching[of
G X Y R e1 e2] by auto
              qed
            qed
          qed
        qed
      qed
  next
    show tails-set G {e |e. e ∈ arcs G ∧ tail G e ∈ X ∧ head G e = R (tail G e)}
= X
       using perfect[of G X Y]  assms(1) R by auto
    qed thus ?thesis by auto
  qed
qed
```

```
theorem Hall-digraph:
 fixes  G :: ('a::countable, 'b::countable) pre-digraph and X:: 'a set
  assumes dir-bipartite-digraph G X Y  and ∀ i∈X. finite (neighbourhood G i)
  shows (∃ E. dirBD-perfect-matching G X Y E) ⟷
```

(∀ J⊆X. *finite* J ⟶ (*card* J) ≤ *card* (⋃ (*neighbourhood* G ‘ J)))
proof
  assume *hip1*: ∃ E. *dirBD-perfect-matching* G X Y E
  show (∀ J⊆X. *finite* J ⟶ (*card* J) ≤ *card* (⋃ (*neighbourhood* G ‘ J)))
    using *hip1 assms*(1−2) *marriage-necessary-graph*[*of* G X Y] **by** *auto*
next
  assume *hip2*: ∀ J⊆X. *finite* J ⟶ *card* J ≤ *card* (⋃ (*neighbourhood* G ‘ J))
  show ∃ E. *dirBD-perfect-matching* G X Y E **using** *assms marriage-sufficiency-graph*[*of* G X Y] *hip2*
  proof−
    have (∀ J⊆X. *finite* J ⟶ (*card* J) ≤ *card* (⋃ (*neighbourhood* G ‘ J)))
                                              ⟶ (∃ E. *dirBD-perfect-matching* G X Y E)
      using *assms marriage-sufficiency-graph*[*of* G X Y] **by** *auto*
    thus *?thesis* **using** *hip2* **by** *auto*
  qed
qed

locale *set-family* =
  fixes I :: ′a *set* **and** X :: ′a ⇒ ′b *set*

locale *sdr* = *set-family* +
  fixes *repr* :: ′a ⇒ ′b
  assumes *inj-repr*: *inj-on repr* I **and** *repr-X*: x ∈ I ⟹ *repr* x ∈ X x

locale *bipartite-digraph* =
  fixes X :: ′a *set* **and** Y :: ′b *set* **and** E :: (′a × ′b) *set*
  assumes *E-subset*: E ⊆ X × Y

locale *Count-Nbhdfin-bipartite-digraph* =
  fixes X :: ′a:: *countable set* **and** Y :: ′b:: *countable set*
      and E :: (′a × ′b) *set*
  assumes *E-subset*: E ⊆ X × Y

  assumes *Nbhd-Tail-finite*: ∀ x ∈ X. *finite* {y. (x, y) ∈ E}

locale *matching* = *bipartite-digraph* +
  fixes M :: (′a × ′b) *set*
  assumes *M-subset*: M ⊆ E

```
  assumes M-right-unique: (x, y) ∈ M ⟹ (x, y') ∈ M ⟹ y = y'
  assumes M-left-unique: (x, y) ∈ M ⟹ (x', y) ∈ M ⟹ x = x'
```

```
locale perfect-matching = matching +
  assumes M-perfect: fst ` M = X
```

```
lemma (in sdr) perfect-matching:
       perfect-matching I (⋃ i∈I. X i) (Sigma I X) {(x, repr x)|x. x ∈ I}
 by  unfold-locales (use inj-repr repr-X in ‹force simp: inj-on-def›)+
```

```
lemma (in perfect-matching) sdr: sdr X (λx. {y. (x,y) ∈ E}) (λx. the-elem {y.
(x,y) ∈ M})
proof unfold-locales
  define Y where Y = (λx. {y. (x,y) ∈ M})
  have Y: ∃ y. Y x = {y} if x ∈ X for x
    using that M-right-unique M-perfect unfolding Y-def by fastforce
  show inj-on (λx. the-elem (Y x)) X
    unfolding Y-def inj-on-def
   by (metis (mono-tags, lifting) M-left-unique Y Y-def mem-Collect-eq singletonI
the-elem-eq)
  show the-elem (Y x) ∈ {y. (x, y) ∈ E} if x ∈ X for x
    using Y M-subset Y-def ‹x ∈ X› by fastforce
qed
```

From these transformations, the formalization of the countable version of Hall's Theorem for Graphs (more specifically, its sufficiency) can be stated as below; in words "if for any finite $X_s \subseteq X$ the subgraph induced by X_s has a perfect matching then the whole graph has a perfect matching"

```
theorem (in Count-Nbhdfin-bipartite-digraph) Hall-Graph:
 assumes ∃ g. enumeration (g:: nat ⇒'a) and ∃ h. enumeration (h:: nat ⇒'b)
 shows (∀ Xs ⊆ X. (finite Xs) ⟶
           (∃ Ms.  perfect-matching Xs
                              {y. x ∈ Xs ∧ (x,y) ∈ E}
                              {(x,y). x ∈ Xs ∧ (x,y) ∈ E}
                              Ms))
        ⟶ (∃ M. perfect-matching X Y E M)
proof(unfold-locales, rule impI)
  assume premisse1: (∀ Xs ⊆ X. (finite Xs) ⟶
           (∃ Ms. perfect-matching Xs
                             {y. x ∈ Xs ∧ (x,y) ∈ E}
                             {(x,y). x ∈ Xs ∧ (x,y) ∈ E}
                             Ms))
  show (∃ M. perfect-matching X Y E M)
  proof−
```

```
  have A: ∀ Xs⊆X. finite Xs ⟶  card Xs ≤ card (⋃ ( (λx. {y. (x,y) ∈ E}) `
Xs))
  proof(rule allI, rule impI)
    fix Xs
    define Ys where Ys =  {y. x  ∈ Xs  ∧ (x,y) ∈ E}
    define Es where Es = {(x,y). x  ∈ Xs  ∧ (x,y) ∈ E}
    assume hip1:  Xs ⊆ X
    show finite Xs ⟶  card Xs ≤ card (⋃ ( (λx. {y. (x,y) ∈ E}) ` Xs))
    proof
      assume hip2: finite Xs
      show  card Xs ≤ card (⋃ ( (λx. {y. (x,y) ∈ E}) ` Xs))
      proof−
        have  (∃ Ms. perfect-matching Xs Ys Es Ms)
        using hip1 hip2 premisse1 Ys-def Es-def by auto
      then obtain Ms where Ms: perfect-matching Xs Ys Es Ms
        using Ys-def Es-def by auto
      have sdrXs : sdr Xs (λx. {y. (x,y) ∈ Es}) (λx. the-elem {y. (x,y) ∈ Ms})
        using Ms perfect-matching.sdr[of Xs Ys Es Ms]  by blast
      define Rs where Rs =  (λx. the-elem {y. (x,y) ∈ Ms})
      have inj-Rs: inj-on Rs Xs
        using sdrXs Rs-def sdr.inj-repr[of Xs (λx. {y. (x,y) ∈ Es}) Rs] by auto
      have B: ∀ x. x∈Xs  ⟶  Rs x ∈  (λx. {y. (x,y) ∈ Es}) x
      proof(rule allI, rule impI)
        fix x
        assume x∈Xs
        thus Rs x ∈  (λx. {y. (x,y) ∈ Es}) x
          using sdrXs Rs-def sdr.repr-X[of Xs  (λx. {y. (x,y) ∈ Es})  Rs x]
          by auto
      qed
      have YsE : Ys = (⋃x∈Xs. {y. (x, y) ∈ E})
      proof
        show Ys ⊆ (⋃x∈Xs. {y. (x, y) ∈ E})
        proof fix x
          assume x ∈ Ys
          thus  x ∈ (⋃x∈Xs. {y. (x, y) ∈ E})  using Ys-def by blast
        qed
        next
        show (⋃x∈Xs. {y. (x, y) ∈ E}) ⊆ Ys
        proof fix x
          assume x ∈ (⋃x∈Xs. {y. (x, y) ∈ E})
          thus x ∈ Ys
            using Es-def Ms UN-iff bipartite-digraph.E-subset
            case-prodI matching-def mem-Collect-eq mem-Sigma-iff
            perfect-matching-def by fastforce
          qed
        qed
        have YsFin: finite Ys
          using Nbhd-Tail-finite Ys-def hip1 hip2 by fastforce
        have (∀ x∈Xs. Rs x ∈ (λx. {y. (x,y) ∈ Es}) x) ∧ inj-on Rs Xs
```

```
        using B inj-Rs by auto
      thus ?thesis using YsFin YsE Es-def card-inj-on-le[of Rs Xs Ys]  by blast
      qed
    qed
  qed
  have premisse2: Count-Nbhdfin-bipartite-digraph X Y E
    by (simp add: Count-Nbhdfin-bipartite-digraph-axioms)
  have X-countable : countable X by simp
  have P2: ∃ R. system-representatives (λx. {y. (x,y) ∈ E}) X R
    using premisse2 A Hall[of X (λx. {y. (x,y) ∈ E})]
          Nbhd-Tail-finite by blast
   then obtain R where system-representatives (λx. {y. (x, y) ∈ E}) X R by
auto
    hence sdr X (λx. {y. (x,y) ∈ E}) R unfolding system-representatives-def
sdr-def by auto
  hence ∃ M. perfect-matching X (⋃ i∈X. (λx. {y. (x,y) ∈ E}) i) (Sigma X (λx.
{y. (x,y) ∈ E})) M
    using sdr.perfect-matching[of X (λx. {y. (x,y) ∈ E}) R] by auto
  then obtain M
  where PM0: perfect-matching X (⋃ i∈X. (λx. {y. (x,y) ∈ E}) i)
             (Sigma X (λx. {y. (x,y) ∈ E})) M by auto
  have Ed2: E = (Sigma X (λx. {y. (x,y) ∈ E}))
  proof
    show   E ⊆ (SIGMA x:X. {y. (x, y) ∈ E})
    proof fix x
      assume x ∈ E
      thus x ∈ (SIGMA x:X. {y. (x, y) ∈ E})
        using E-subset by blast
    qed
    next
    show (SIGMA x:X. {y. (x, y) ∈ E}) ⊆ E
    proof fix x
      assume  x ∈ (SIGMA x:X. {y. (x, y) ∈ E})
      thus x ∈ E by blast
    qed
  qed
  have PM1: perfect-matching X (⋃ i∈X. (λx. {y. (x,y) ∈ E}) i) E M
    using PM0 Ed2 by auto
  hence PM2: perfect-matching X Y E M
    using Count-Nbhdfin-bipartite-digraph-axioms unfolding  matching-def per-
fect-matching-def
  proof −
    assume (bipartite-digraph X (⋃ i∈X. {y. (i, y) ∈ E}) E ∧ matching-axioms
E M) ∧ perfect-matching-axioms X M
      then show (bipartite-digraph X Y E ∧ matching-axioms E M) ∧ per-
fect-matching-axioms X M
      using E-subset bipartite-digraph.intro by blast
  qed
  thus PM : ∃ M. perfect-matching X Y E M using PM2 by auto
```

qed
qed

end

9 de Bruijn-Erdős k-coloring theorem for countable infinite graphs

This section formalizes de Bruijn-Erdős k-coloring theorem for countable infinite graphs. The construction applies the compactness theorem for propositional logic directly.

type-synonym *'v digraph* = (*'v set*) × ((*'v* × *'v*) *set*)

abbreviation *vert* :: *'v digraph* ⇒ *'v set* (*V*[-] [*80*] *80*) **where**
V[*G*] ≡ *fst G*

abbreviation *edge* :: *'v digraph* ⇒ (*'v* × *'v*) *set* (*E*[-] [*80*] *80*) **where**
E[*G*] ≡ *snd G*

definition *is-graph* :: *'v digraph* ⇒ *bool* **where**
is-graph G ≡ ∀ *u v*. (*u*,*v*) ∈ *E*[*G*] ⟶ *u* ∈ *V*[*G*] ∧ *v* ∈ *V*[*G*] ∧ *u* ≠ *v*

definition *is-induced-subgraph* :: *'v digraph* ⇒*'v digraph* ⇒ *bool* **where**
is-induced-subgraph H G ≡
(*V*[*H*] ⊆ *V*[*G*]) ∧ *E*[*H*] = *E*[*G*] ∩ ((*V*[*H*]) × (*V*[*H*]))

lemma
assumes *is-graph G* **and** *is-induced-subgraph H G*
shows *is-graph H*

definition *coloring* :: (*'v* ⇒ *nat*) ⇒ *nat* ⇒ *'v digraph* ⇒ *bool* **where**
coloring c k G ≡
(∀ *u*. *u*∈*V*[*G*]⟶ *c*(*u*)≤*k*) ∧ (∀ *u v*.(*u*,*v*)∈*E*[*G*] ⟶ *c*(*u*)≠*c*(*v*))

definition *colorable* :: *'v digraph* ⇒ *nat* ⇒ *bool* **where**
colorable G k ≡ ∃ *c*. *coloring c k G*

primrec *atomic-disjunctions* :: *'v* ⇒ *nat* ⇒ (*'v* × *nat*)*formula* **where**
atomic-disjunctions v 0 = *atom* (*v*, *0*)
| *atomic-disjunctions v* (*Suc k*) =
(*atom* (*v*, *Suc k*)) ∨. (*atomic-disjunctions v k*)

definition $\mathcal{F}$ :: *'v digraph* ⇒ *nat* ⇒ ((*'v* × *nat*)*formula*) *set* **where**
$\mathcal{F}$ *G k* ≡ (⋃*v*∈*V*[*G*]. {*atomic-disjunctions v k*})

definition $\mathcal{G}$ :: *'v digraph* ⇒ *nat* ⇒ (*'v* × *nat*)*formula set* **where**
$\mathcal{G}$ *G k* ≡ {¬.(*atom* (*v*, *i*) ∧. *atom*(*v*,*j*))
| *v i j*. (*v*∈*V*[*G*]) ∧ (*0*≤*i* ∧ *0*≤*j* ∧ *i*≤*k* ∧ *j*≤*k* ∧ *i*≠*j*)}

definition $\mathcal{H}$ :: *'v digraph* ⇒ *nat* ⇒ (*'v* × *nat*)*formula set* **where**
$\mathcal{H}$ *G k* ≡ {¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))
|*u v i* . (*u*∈*V*[*G*] ∧ *v*∈*V*[*G*] ∧ (*u*,*v*)∈*E*[*G*]) ∧ (*0*≤*i* ∧ *i*≤*k*)}

definition $\mathcal{T}$ :: *'v digraph* ⇒ *nat* ⇒ (*'v* × *nat*)*formula set* **where**
$\mathcal{T}$ *G k* ≡ ($\mathcal{F}$ *G k*) ∪ ($\mathcal{G}$ *G k*) ∪ ($\mathcal{H}$ *G k*)

primrec *vertices-formula* :: (*'v* × *nat*)*formula* ⇒ *'v set* **where**
vertices-formula FF = {}
| *vertices-formula TT* = {}
| *vertices-formula* (*atom P*) = {*fst P*}
| *vertices-formula* (¬. *F*) = *vertices-formula F*
| *vertices-formula* (*F* ∧. *G*) = *vertices-formula F* ∪ *vertices-formula G*
| *vertices-formula* (*F* ∨. *G*) = *vertices-formula F* ∪ *vertices-formula G*
| *vertices-formula* (*F* →.*G*) = *vertices-formula F* ∪ *vertices-formula G*

definition *vertices-set-formulas* :: (*'v* × *nat*)*formula set* ⇒ *'v set* **where**
vertices-set-formulas S = (⋃*F*∈ *S*. *vertices-formula F*)

lemma *finite-vertices*:
shows *finite* (*vertices-formula F*)
by(*induct F*, *auto*)

lemma *vertices-disjunction*:
assumes *F* = *atomic-disjunctions v k* **shows** *vertices-formula F* = {*v*}
proof−
have *F* = *atomic-disjunctions v k* ⟹ *vertices-formula F* = {*v*}
proof(*induct k arbitrary*: *F*)
case *0*
assume *F* = *atomic-disjunctions v 0*
hence *F* = *atom* (*v*, *0*) **by** *auto*
thus *vertices-formula F* = {*v*} **by** *auto*
next
case(*Suc k*)
have *F* =(*atom* (*v*, *Suc k*)) ∨. (*atomic-disjunctions v k*)
using *Suc*(*2*) **by** *auto*
hence *vertices-formula F* = *vertices-formula* (*atom* (*v*, *Suc k*)) ∪ *vertices-formula* (*atomic-disjunctions v k*) **by** *auto*
hence *vertices-formula F* = {*v*} ∪ *vertices-formula* (*atomic-disjunctions v k*)
by *auto*

```
    hence vertices-formula F = {v} ∪ {v} using Suc(1) by auto
    thus vertices-formula F = {v} by auto
  qed
  thus ?thesis using assms by auto
qed


lemma all-vertices-colored:
  shows vertices-set-formulas (ℱ G k) ⊆ V[G]
proof
  fix x
  assume hip: x ∈ vertices-set-formulas (ℱ G k) show x ∈ V[G]
  proof−
    have x ∈ (⋃F∈(ℱ G k). vertices-formula F) using hip
      by(unfold vertices-set-formulas-def,auto)
    hence ∃F∈(ℱ G k). x ∈ vertices-formula F by auto
    then obtain F where F∈(ℱ G k) and x: x ∈ vertices-formula F by auto
    hence ∃ v∈V[G]. F∈{atomic-disjunctions v k} by (unfold ℱ-def, auto)
    then obtain v where v: v∈V[G] and F∈{atomic-disjunctions v k} by auto
    hence F = atomic-disjunctions v k by auto
    hence vertices-formula F = {v}
      using vertices-disjunction[OF ‹F = atomic-disjunctions v k›] by auto
    hence x = v using x by auto
    thus ?thesis using v by auto
  qed
qed

lemma vertices-maximumC:
  shows vertices-set-formulas(𝒢 G k) ⊆ V[G]
proof
  fix x
  assume hip: x ∈ vertices-set-formulas (𝒢 G k) show x ∈ V[G]
  proof−
    have x ∈ (⋃F∈(𝒢 G k). vertices-formula F) using hip
      by(unfold vertices-set-formulas-def,auto)
    hence ∃F∈(𝒢 G k). x ∈ vertices-formula F by auto
    then obtain F where F∈(𝒢 G k) and x: x ∈ vertices-formula F
      by auto
    hence ∃v i j. v∈V[G] ∧ F = ¬.(atom (v, i) ∧. atom(v,j))
      by (unfold 𝒢-def, auto)
    then obtain v i j where v∈V[G] and F = ¬.(atom (v, i) ∧. atom(v,j))
      by auto
    hence v: v∈V[G] and F = ¬.(atom (v, i) ∧. atom(v,j)) by auto
    hence v: v∈V[G] and vertices-formula F = {v} by auto
    thus x ∈ V[G] using x by auto
  qed
qed

lemma distinct-verticesC:
```

shows *vertices-set-formulas*($\mathcal{H}$ *G k*)⊆ *V*[*G*]
proof
fix *x*
assume *hip*: *x* ∈ *vertices-set-formulas* ($\mathcal{H}$ *G k*) **show** *x* ∈ *V*[*G*]
proof−
have *x* ∈ (⋃ *F*∈($\mathcal{H}$ *G k*). *vertices-formula F*) **using** *hip*
by(*unfold vertices-set-formulas-def*,*auto*)
hence ∃ *F*∈($\mathcal{H}$ *G k*) . *x* ∈ *vertices-formula F* **by** *auto*
then obtain *F* **where** *F*∈($\mathcal{H}$ *G k*) **and** *x*: *x* ∈ *vertices-formula F*
by *auto*
hence ∃ *u v i* . *u*∈*V*[*G*] ∧ *v*∈*V*[*G*] ∧ *F* = ¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))
by (*unfold* $\mathcal{H}$*-def*, *auto*)
then obtain *u v i*
where *u*∈*V*[*G*] **and** *v*∈*V*[*G*] **and** *F* = ¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))
by *auto*
hence *u*∈*V*[*G*] **and** *v*∈*V*[*G*] **and** *F* = ¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))
by *auto*
hence *u*: *u*∈*V*[*G*] **and** *v*: *v*∈*V*[*G*] **and** *vertices-formula F* = {*u*, *v*}
by *auto*
hence *x*=*u* ∨ *x*=*v* **using** *x* **by** *auto*
thus *x* ∈ *V*[*G*] **using** *u v* **by** *auto*
qed
qed

lemma *vv*:
shows *vertices-set-formulas* (*A* ∪ *B*) = (*vertices-set-formulas A*) ∪ (*vertices-set-formulas B*)
by(*unfold vertices-set-formulas-def*, *auto*)

lemma *vv1*:
assumes *F*∈($\mathcal{F}$ *G k*)
shows (*vertices-formula F*) ⊆ (*vertices-set-formulas* ($\mathcal{F}$ *G k*))
proof
fix *x*
assume *hip*: *x* ∈ *vertices-formula F*
show *x* ∈ *vertices-set-formulas* ($\mathcal{F}$ *G k*)
proof−
have ∃ *F*. *F*∈($\mathcal{F}$ *G k*) ∧ *x* ∈ *vertices-formula F* **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold vertices-set-formulas-def*, *auto*)
qed
qed

lemma *vv2*:
assumes *F*∈($\mathcal{G}$ *G k*)
shows (*vertices-formula F*) ⊆ (*vertices-set-formulas* ($\mathcal{G}$ *G k*))
proof
fix *x*
assume *hip*: *x* ∈ *vertices-formula F*
show *x* ∈ *vertices-set-formulas* ($\mathcal{G}$ *G k*)

proof−
have $\exists F.\ F \in (\mathcal{G}\ G\ k) \wedge x \in$ *vertices-formula F* **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold vertices-set-formulas-def*, *auto*)
qed
qed

lemma *vv3*:
assumes $F \in (\mathcal{H}\ G\ k)$
shows (*vertices-formula F*) $\subseteq$ (*vertices-set-formulas* $(\mathcal{H}\ G\ k)$)
proof
fix *x*
assume *hip*: $x \in$ *vertices-formula F*
show $x \in$ *vertices-set-formulas* $(\mathcal{H}\ G\ k)$
proof−
have $\exists F.\ F \in (\mathcal{H}\ G\ k) \wedge x \in$ *vertices-formula F* **using** *assms hip* **by** *auto*
thus *?thesis* **by**(*unfold vertices-set-formulas-def*, *auto*)
qed
qed

lemma *vertex-set-inclusion*:
shows *vertices-set-formulas* $(\mathcal{T}\ G\ k) \subseteq V[G]$
proof
fix *x*
assume *hip*: $x \in$ *vertices-set-formulas* $(\mathcal{T}\ G\ k)$ **show** $x \in V[G]$
proof−
have $x \in$ *vertices-set-formulas* $((\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k) \cup (\mathcal{H}\ G\ k))$
using *hip* **by** (*unfold* $\mathcal{T}$*-def*,*auto*)
hence $x \in$ *vertices-set-formulas* $((\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k)) \cup$
vertices-set-formulas$(\mathcal{H}\ G\ k)$
using *vv*[*of* $(\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k)$] **by** *auto*
hence $x \in$ *vertices-set-formulas* $((\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k)) \vee$
$x \in$ *vertices-set-formulas*$(\mathcal{H}\ G\ k)$
by *auto*
thus *?thesis*
proof(*rule disjE*)
assume *hip*: $x \in$ *vertices-set-formulas* $(\mathcal{F}\ G\ k \cup \mathcal{G}\ G\ k)$
hence $x \in (\bigcup F \in (\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k).$ *vertices-formula F*)
by(*unfold vertices-set-formulas-def*, *auto*)
then obtain *F*
where *F*: $F \in (\mathcal{F}\ G\ k) \cup (\mathcal{G}\ G\ k)$ **and** *x*: $x \in$ *vertices-formula F* **by** *auto*
from *F* **have** (*vertices-formula F*) $\subseteq$ (*vertices-set-formulas* $(\mathcal{F}\ G\ k)$)
$\vee$ *vertices-formula F* $\subseteq$ (*vertices-set-formulas* $(\mathcal{G}\ G\ k)$)
using *vv1 vv2* **by** *blast*
hence $x \in$ *vertices-set-formulas* $(\mathcal{F}\ G\ k) \vee x \in$ *vertices-set-formulas* $(\mathcal{G}\ G\ k)$
using *x* **by** *auto*
thus $x \in V[G]$
using *all-vertices-colored*[*of G k*] *vertices-maximumC*[*of G k*] **by** *auto*

next
assume *x* ∈ *vertices-set-formulas* ($\mathcal{H}$ *G k*)
hence
x ∈ (⋃ *F*∈($\mathcal{H}$ *G k*). *vertices-formula F*)
by(*unfold vertices-set-formulas-def*, *auto*)
then obtain *F* **where** *F*: *F*∈($\mathcal{H}$ *G k*) **and** *x*: *x* ∈ *vertices-formula F*
by *auto*
from *F* **have** (*vertices-formula F*) ⊆ (*vertices-set-formulas* ($\mathcal{H}$ *G k*))
using *vv3* **by** *blast*
hence *x* ∈ *vertices-set-formulas* ($\mathcal{H}$ *G k*) **using** *x* **by** *auto*
thus *x* ∈ *V*[*G*] **using** *distinct-verticesC*[*of G k*]
by *auto*
qed
qed
qed

lemma *vsf*:
assumes *G* ⊆ *H*
shows *vertices-set-formulas G* ⊆ *vertices-set-formulas H*
using *assms* **by**(*unfold vertices-set-formulas-def*, *auto*)

lemma *vertices-subset-formulas*:
assumes *S* ⊆ ($\mathcal{T}$ *G k*)
shows *vertices-set-formulas S* ⊆ *V*[*G*]
proof−
have *vertices-set-formulas S* ⊆ *vertices-set-formulas* ($\mathcal{T}$ *G k*)
using *assms vsf* **by** *auto*
thus *?thesis* **using** *vertex-set-inclusion*[*of G*] **by** *auto*
qed

definition *subgraph-aux* :: *'v digraph* ⇒ *'v set* ⇒*'v digraph* **where**
subgraph-aux G V ≡ (*V*, *E*[*G*] ∩ (*V* × *V*))

lemma *induced-subgraph*:
assumes *is-graph G* **and** *S* ⊆($\mathcal{T}$ *G k*)
shows *is-induced-subgraph* (*subgraph-aux G* (*vertices-set-formulas S*)) *G*
proof−
let *?V* = *vertices-set-formulas S*
let *?H* = (*?V*, *E*[*G*] ∩ (*?V* × *?V*))
have *1*: *E*[*?H*] = *E*[*G*] ∩ (*?V* × *?V*) **and** *2*: *V*[*?H*]= *?V* **by** *auto*
have (*V*[*?H*] ⊆ *V*[*G*]) **using** *2 assms*(*2*) *vertices-subset-formulas*[*of S G*] **by** *auto*
moreover
have *E*[*?H*] = (*E*[*G*] ∩ ((*V*[*?H*]) × (*V*[*?H*]))) **using** *1 2* **by** *auto*
ultimately

```
  have  is-induced-subgraph ?H G by(unfold is-induced-subgraph-def, auto)
  thus ?thesis
    by (simp add: subgraph-aux-def)
qed

lemma finite-subgraph:
  assumes is-graph G and S ⊆ (𝒯  G k) and finite S
  shows finite-graph (subgraph-aux G (vertices-set-formulas S))
proof−
  let ?V = vertices-set-formulas S
  let ?H = (?V, E[G] ∩ (?V × ?V))
  have 1: E[?H] =  E[G] ∩ (?V × ?V) and 2: V[?H]= ?V  by auto
  have 3: finite ?V using ‹finite S› finite-vertices
    by(unfold vertices-set-formulas-def, auto)
  hence  finite (V[?H]) using 2 by auto
  thus ?thesis
    by (simp add: finite-graph-def subgraph-aux-def)
qed



fun graph-interpretation :: 'v digraph ⇒ ('v ⇒ nat) ⇒ (('v × nat) ⇒ v-truth)
where
graph-interpretation G f = (λ(v,i).(if v ∈ V[G] ∧ f(v) = i  then Ttrue else Ffalse))

lemma value1:
  assumes v ∈ V[G] and f(v)≤ k  and  F = atomic-disjunctions v  k
  shows t-v-evaluation (graph-interpretation G f) F = Ttrue
proof−
  let ?i = f(v)
  have 0 ≤ ?i by auto
   {have  v ∈ V[G] ⟹ 0 ≤ ?i ⟹ ?i≤k ⟹ F = atomic-disjunctions v  k ⟹
   t-v-evaluation (graph-interpretation G f) F = Ttrue
  proof(induct k arbitrary: F)
    case 0
    have ?i = 0 using 0 (2−3)  by auto
    hence t-v-evaluation (graph-interpretation G f) (atom (v, 0)) =  Ttrue
      using ‹v ∈ V[G]›  by auto
    thus ?case using  0 (4)  by auto
    next
    case(Suc k)
    from Suc(1) Suc(2) Suc(3) Suc(4) Suc(5) show ?case
    proof(cases)
      assume (Suc  k) = ?i
      hence t-v-evaluation (graph-interpretation G f) (atom (v,Suc  k )) =  Ttrue
      using Suc(2) Suc(3) Suc(5) by auto
      hence
      t-v-evaluation (graph-interpretation G f) (atom (v, Suc  k)
```

```
  ∨.atomic-disjunctions v  k) = Ttrue
      using v-disjunction-def by auto
      thus ?case using  Suc(5) by auto
      next
      assume 1: (Suc  k) ≠ ?i
      hence t-v-evaluation (graph-interpretation G f) (atom (v, Suc k)) =  Ffalse
        using Suc(5) by auto
      moreover
      have ?i < (Suc  k) using Suc(4) 1  by auto
      hence ?i ≤ k by auto
       hence t-v-evaluation (graph-interpretation G f) (atomic-disjunctions v k) =
Ttrue
      using  Suc(1) Suc(2) Suc(3)  Suc(5)  by auto
      thus ?case using  Suc(5)  v-disjunction-def by auto
    qed
  qed
 }
  thus ?thesis using assms by auto
qed

lemma t-value-vertex:
  assumes  t-v-evaluation (graph-interpretation G f) (atom (v, i)) = Ttrue
  shows  f(v)=i
proof(rule ccontr)
  assume  f v ≠ i hence t-v-evaluation (graph-interpretation G f) (atom (v, i))
≠ Ttrue by auto
  hence t-v-evaluation (graph-interpretation G f) (atom (v, i)) = Ffalse
  using non-Ttrue[of graph-interpretation G f atom (v, i)] by auto
  thus False using assms by simp
qed

lemma value2:
  assumes  i≠j and F =¬.(atom (v, i) ∧. atom (v, j))
  shows t-v-evaluation (graph-interpretation G f) F = Ttrue
proof(rule ccontr)
  assume t-v-evaluation (graph-interpretation G f) F ≠ Ttrue
  hence  t-v-evaluation (graph-interpretation G f) (¬.(atom (v, i) ∧. atom (v, j)))
≠ Ttrue
    using assms(2) by auto
  hence t-v-evaluation (graph-interpretation G f) (¬.(atom (v, i) ∧. atom (v, j)))
= Ffalse using
  non-Ttrue[of graph-interpretation G f ¬.(atom (v, i) ∧. atom (v, j)) ]
    by auto
   hence  t-v-evaluation (graph-interpretation G f) ((atom (v, i) ∧. atom (v, j)))
= Ttrue
  using  NegationValues1[of graph-interpretation G f (atom (v, i) ∧. atom (v, j))]
by auto
  hence t-v-evaluation (graph-interpretation G f) (atom (v, i)) = Ttrue and
  t-v-evaluation (graph-interpretation G f) (atom (v, j)) = Ttrue
```

using *ConjunctionValues[of graph-interpretation G f atom (v, i) atom (v, j)]* **by** *auto*
hence *f(v)=i* **and** *f(v)=j* **using** *t-value-vertex* **by** *auto*
hence *i=j* **by** *auto*
thus *False* **using** *assms(1)* **by** *auto*
qed

lemma *value3*:
assumes *f(u)≠f(v)* **and** *F =¬.(atom (u, i) ∧. atom (v, i))*
shows *t-v-evaluation (graph-interpretation G f) F = Ttrue*
proof(*rule ccontr*)
assume *t-v-evaluation (graph-interpretation G f) F ≠ Ttrue*
hence
t-v-evaluation (graph-interpretation G f) (¬.(atom (u, i) ∧. atom (v, i))) ≠ Ttrue

using *assms(2)* **by** *auto*
hence *t-v-evaluation (graph-interpretation G f) (¬.(atom (u, i) ∧. atom (v, i))) = Ffalse*
using
non-Ttrue[of graph-interpretation G f ¬.(atom (u, i) ∧. atom (v, i))]
by *auto*
hence *t-v-evaluation (graph-interpretation G f) ((atom (u, i) ∧. atom (v, i))) = Ttrue*
using *NegationValues1[of graph-interpretation G f (atom (u, i) ∧. atom (v, i))]*
by *auto*
hence *t-v-evaluation (graph-interpretation G f) (atom (u, i)) = Ttrue* **and**
t-v-evaluation (graph-interpretation G f) (atom (v, i)) = Ttrue
using *ConjunctionValues[of graph-interpretation G f atom (u, i) atom (v, i)]*
by *auto*
hence *f(u)=i* **and** *f(v)=i* **using** *t-value-vertex* **by** *auto*
hence *f(u)=f(v)* **by** *auto*
thus *False* **using** *assms(1)* **by** *auto*
qed

theorem *coloring-satisfiable*:
assumes *is-graph G* **and** *S ⊆ (𝒯 G k)* **and**
coloring f k (subgraph-aux G (vertices-set-formulas S))
shows *satisfiable S*
proof−
let *?V = vertices-set-formulas S*
let *?H = subgraph-aux G ?V*
have *(graph-interpretation ?H f) model S*
proof(*unfold model-def*)
show *∀ F ∈ S. t-v-evaluation (graph-interpretation ?H f) F = Ttrue*
proof
fix *F* **assume** *F ∈ S*
show *t-v-evaluation (graph-interpretation ?H f) F = Ttrue*
proof−

```
have 1:  vertices-formula F ⊆?V
proof
  fix v
  assume v ∈ (vertices-formula F) thus v ∈ ?V
  using ‹F ∈ S› by(unfold vertices-set-formulas-def,auto)
qed
have F ∈ (𝓕 G k) ∪ (𝓖 G k)  ∪ (𝓗 G k)
using  ‹F ∈ S› assms(2)  by(unfold 𝓣-def,auto)
hence  F ∈ (𝓕 G k) ∨ F ∈ (𝓖 G k) ∨ F ∈ (𝓗 G k) by auto
thus ?thesis
proof(rule disjE)
  assume F ∈ (𝓕 G k)
  hence ∃ v∈V[G]. F = atomic-disjunctions v  k  by(unfold 𝓕-def,auto)
  then obtain v
  where v: v∈V[G] and F: F = atomic-disjunctions v k
    by auto
  have v∈?V using F vertices-disjunction[of F] 1 by auto
  hence v∈ V[?H] by(unfold subgraph-aux-def, auto)
  hence f(v)≤ k using coloring-def[of f k ?H] assms(3) by auto
  thus ?thesis using F  value1[OF ‹v∈V[?H]›] by auto
  next
  assume F ∈ (𝓖 G k) ∨ F ∈ (𝓗 G k)
  thus ?thesis
  proof(rule disjE)
    assume F ∈ (𝓖 G k)
    hence ∃ v.∃ i.∃ j. F = ¬.(atom (v, i) ∧. atom(v,j)) ∧ ( i≠j)
    by(unfold 𝓖-def, auto)
    then obtain v i j
    where F = ¬.(atom (v, i) ∧. atom(v,j)) and (i≠j)
    by auto
    thus t-v-evaluation (graph-interpretation ?H f) F = Ttrue
    using value2[OF ‹i≠j› ‹F = ¬.(atom (v, i) ∧. atom(v,j))›]
    by auto
    next
    assume  F ∈ (𝓗 G k)
    hence  ∃ u.∃ v.∃ i.(F = ¬.(atom (u, i) ∧. atom(v,i)) ∧  (u,v)∈E[G])
    by(unfold 𝓗-def, auto)
    then obtain u v i
    where F:  F = ¬.(atom (u, i) ∧. atom(v,i)) and uv: (u,v)∈E[G]
    by auto
    have vertices-formula F = {u,v} using F by auto
    hence {u,v} ⊆ ?V using 1 by auto
    hence (u,v)∈E[?H] using uv  by(unfold subgraph-aux-def, auto)
    hence  f(u) ≠f(v) using  coloring-def[of f k ?H] assms(3)
      by auto
    show ?thesis
      using value3[OF ‹f(u) ≠f(v)› ‹F = ¬.(atom (u, i) ∧. atom(v,i))›]
      by auto
  qed
```

```
        qed
      qed
    qed
  qed
  thus satisfiable S by(unfold satisfiable-def, auto)
qed
```

```
fun graph-coloring :: (('v × nat) ⇒ v-truth) ⇒ nat  ⇒ ('v ⇒ nat)
  where
graph-coloring I k = (λv.(THE i. (t-v-evaluation I (atom (v,i)) = Ttrue) ∧ 0≤i ∧
i≤k))

lemma unicity:
  assumes (t-v-evaluation I (atom (v, i)) = Ttrue ∧ 0≤i ∧ i ≤ k)
  and ∀ j. (0≤j ∧ j≤k ∧ i≠j) ⟶ (t-v-evaluation I (¬.(atom (v, i) ∧. atom(v,j)))
= Ttrue)
  shows ∀ j. (0≤j ∧ j≤k ∧ i≠j) ⟶  t-v-evaluation I (atom (v, j)) = Ffalse
proof(rule allI, rule impI)
  fix j
  assume hip: 0≤j ∧ j≤k ∧ i≠j
  show t-v-evaluation I (atom (v, j)) = Ffalse
  proof(rule ccontr)
    assume t-v-evaluation I (atom (v, j)) ≠ Ffalse
    hence t-v-evaluation I (atom (v, j)) = Ttrue using Bivaluation by blast
    hence 1: t-v-evaluation I (atom (v, i) ∧. atom(v,j))  = Ttrue
      using assms(1) v-conjunction-def by auto
    have t-v-evaluation I (¬.(atom (v, i) ∧. atom(v,j))) = Ttrue
      using hip assms(2)  by auto
    hence t-v-evaluation I (atom (v, i) ∧. atom(v,j)) = Ffalse
      using NegationValues2  by blast
    thus False using 1  by auto
  qed
qed

lemma existence:
  assumes (t-v-evaluation I (atom (v, i)) = Ttrue ∧ 0≤i ∧ i ≤ k)
  and  ∀ j. (0≤j ∧ j≤k ∧ i≠j) ⟶  t-v-evaluation I (atom (v, j)) = Ffalse
shows  (∀ x. (t-v-evaluation I (atom (v, x)) = Ttrue ∧ 0≤x ∧ x ≤ k) ⟶ x = i)
proof(rule allI)
  fix x
  show t-v-evaluation I (atom (v, x)) = Ttrue ∧ 0 ≤ x ∧ x ≤ k ⟶ x = i
  proof(rule impI)
    assume hip: t-v-evaluation I (atom (v, x)) = Ttrue ∧ 0≤x ∧ x ≤ k show x
= i
    proof(rule ccontr)
      assume 1:  x ≠ i
```

have $0 \leq x \wedge x \leq k$ **using** *hip* **by** *auto*
hence *t-v-evaluation I* (*atom* (*v*, *x*)) = *Ffalse* **using** *1 assms*(*2*) **by** *auto*
thus *False* **using** *hip* **by** *auto*
qed
qed
qed

lemma *exist-unicity1*:
assumes (*t-v-evaluation I* (*atom* (*v*, *i*)) = *Ttrue* $\wedge$ $0 \leq i \wedge i \leq k$)
and $\forall j$. ($0 \leq j \wedge j \leq k \wedge i \neq j$) $\longrightarrow$ (*t-v-evaluation I* ($\neg$.(*atom* (*v*, *i*) $\wedge$. *atom*(*v*,*j*))) = *Ttrue*)
shows ($\forall x$. (*t-v-evaluation I* (*atom* (*v*, *x*)) = *Ttrue* $\wedge$ $0 \leq x \wedge x \leq k$) $\longrightarrow$ $x = i$)
using *assms unicity*[*of I v i k*] *existence*[*of I v i k*] **by** *blast*

lemma *exist-unicity2*:
assumes (*t-v-evaluation I* (*atom* (*v*, *i*)) = *Ttrue* $\wedge$ $0 \leq i \wedge i \leq k$) **and**
($\bigwedge x$. (*t-v-evaluation I* (*atom* (*v*, *x*)) = *Ttrue* $\wedge$ $0 \leq x \wedge x \leq k$) $\Longrightarrow$ $x = i$)
shows (*THE a*. (*t-v-evaluation I* (*atom* (*v*,*a*)) = *Ttrue* $\wedge$ $0 \leq a \wedge a \leq k$)) = *i*
using *assms* **by** (*rule the-equality*)

lemma *exist-unicity*:
assumes (*t-v-evaluation I* (*atom* (*v*, *i*)) = *Ttrue* $\wedge$ $0 \leq i \wedge i \leq k$) **and**
$\forall j$. ($0 \leq j \wedge j \leq k \wedge i \neq j$) $\longrightarrow$ (*t-v-evaluation I* ($\neg$.(*atom* (*v*, *i*) $\wedge$. *atom*(*v*,*j*))) = *Ttrue*)
shows (*THE a*. (*t-v-evaluation I* (*atom* (*v*,*a*)) = *Ttrue* $\wedge$ $0 \leq a \wedge a \leq k$)) = *i*
using *assms exist-unicity1*[*of I v i k*] *exist-unicity2*[*of I v i k*] **by** *blast*

lemma *unique-color*:
assumes $v \in V[G]$
shows $\forall i\ j$.($0 \leq i \wedge 0 \leq j \wedge i \leq k \wedge j \leq k \wedge i \neq j$) $\longrightarrow$ ($\neg$.(*atom* (*v*, *i*) $\wedge$. *atom*(*v*,*j*))$\in$ ($\mathcal{G}$ *G k*))
proof(*rule allI*)+
fix *i j*
show $0 \leq i \wedge 0 \leq j \wedge i \leq k \wedge j \leq k \wedge i \neq j \longrightarrow$ $\neg$.(*atom* (*v*, *i*) $\wedge$. *atom* (*v*, *j*)) $\in$ ($\mathcal{G}$ *G k*)
proof(*rule impI*)
assume $0 \leq i \wedge 0 \leq j \wedge i \leq k \wedge j \leq k \wedge i \neq j$
thus $\neg$.(*atom* (*v*, *i*) $\wedge$. *atom* (*v*, *j*)) $\in$ ($\mathcal{G}$ *G k*)
using ‹$v \in V[G]$› **by**(*unfold* $\mathcal{G}$*-def*, *auto*)
qed
qed

lemma *different-colors*:
assumes $u \in V[G]$ **and** $v \in V[G]$ **and** $(u,v) \in E[G]$
shows $\forall i$.($0 \leq i \wedge i \leq k$) $\longrightarrow$ ($\neg$.(*atom* (*u*, *i*) $\wedge$. *atom*(*v*,*i*))$\in$ ($\mathcal{H}$ *G k*))
proof(*rule allI*)
fix *i*
show $0 \leq i \wedge i \leq k \longrightarrow$ ($\neg$.(*atom* (*u*, *i*) $\wedge$. *atom*(*v*,*i*))$\in$ ($\mathcal{H}$ *G k*))
proof(*rule impI*)

```
    assume 0≤i ∧ i≤k
    thus  ¬.(atom (u, i) ∧. atom(v,i))∈ (ℋ G k)
      using assms  by(unfold  ℋ-def, auto)
  qed
qed

lemma atom-value:
  assumes (t-v-evaluation I (atomic-disjunctions u  k)) = Ttrue
  shows ∃ i.(t-v-evaluation I (atom (u,i)) = Ttrue) ∧ 0≤i ∧ i≤k
proof−
  have (t-v-evaluation I (atomic-disjunctions u  k)) = Ttrue ⟹
  ∃ i.(t-v-evaluation I (atom (u,i)) = Ttrue) ∧ 0≤i ∧ i≤k
  proof(induct k)
    case(0)
    assume (t-v-evaluation I (atomic-disjunctions u 0)) = Ttrue
    thus ∃ i. t-v-evaluation I (atom (u, i)) = Ttrue ∧ 0≤i ∧  i ≤ 0 by auto
    next
    case(Suc k)
    from Suc(1) Suc(2) show ?case
    proof−
       have t-v-evaluation I (atom (u, (Suc k)) ∨. (atomic-disjunctions u k)) =
Ttrue
        using Suc(2) by auto
      hence t-v-evaluation I (atom (u, (Suc k))) = Ttrue ∨
      (t-v-evaluation I (atomic-disjunctions u k)) = Ttrue
        using  DisjunctionValues[of I (atom (u, (Suc k)))] by auto
      thus ?case
        using Suc.hyps le-SucI by blast
    qed
  qed
  thus ?thesis using assms by auto
qed



lemma coloring-function:
  assumes u ∈ V[G] and I model (𝒯 G k)
 shows ∃!i. (t-v-evaluation I (atom (u,i)) = Ttrue ∧ 0≤i ∧ i≤k) ∧ graph-coloring
I k u = i
proof−
  from  ‹u ∈ V[G]›
  have atomic-disjunctions u  k ∈ ℱ G k by(induct, unfold  ℱ-def, auto)
  hence  atomic-disjunctions u  k ∈ 𝒯 G k  by(unfold  𝒯-def, auto)
  hence (t-v-evaluation I (atomic-disjunctions u  k)) = Ttrue
    using assms(2) model-def[of I 𝒯 G k] by auto
  hence ∃ i.(t-v-evaluation I (atom (u,i)) = Ttrue ∧ 0≤i ∧ i≤k)
    using atom-value by auto
  then obtain i where i: (t-v-evaluation I (atom (u,i)) = Ttrue) ∧ 0≤i ∧  i≤k
    by auto
```

moreover
have $\forall\, i\ j.(0\leq i \wedge 0\leq j \wedge i\leq k \wedge j\leq k \wedge i\neq j)\longrightarrow$
$(\neg.(atom\ (u,\ i)\ \wedge.atom(u,j))\in (\mathcal{G}\ G\ k))$
using ‹$u \in V[G]$› *unique-color*[*of u*] **by** *auto*
hence $\forall\, j.(0\leq j\ \wedge j\leq k \wedge i\neq j) \longrightarrow (\neg.(atom\ (u,\ i)\ \wedge.\ atom(u,j))\in \mathcal{T}\ G\ k)$
using *i* **by**(*unfold* $\mathcal{T}$-*def*, *auto*)
hence
$\forall\, j.\ (0\leq j \wedge j\leq k \wedge i\neq j) \longrightarrow (t\text{-}v\text{-}evaluation\ I\ (\neg.(atom\ (u,\ i)\ \wedge.\ atom(u,j)))\ =$ *Ttrue*)
using *assms*(*2*) *model-def*[*of I* $\mathcal{T}$ *G k*] **by** *blast*
hence $(THE\ a.\ (t\text{-}v\text{-}evaluation\ I\ (atom\ (u,a)) = Ttrue \wedge 0\leq a \wedge a \leq k\ ))= i$
using *i exist-unicity*[*of I u*] **by** *blast*
hence *graph-coloring I k u* = *i* **by** *auto*
hence
$(t\text{-}v\text{-}evaluation\ I\ (atom\ (u,i)) = Ttrue \wedge 0\leq i \wedge\ i\leq k) \wedge$
graph-coloring I k u = *i*
using *i* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma $\mathcal{H}1$:
assumes $(t\text{-}v\text{-}evaluation\ I\ (atom\ (u,\ a)) = Ttrue \wedge 0\leq a \wedge a\leq k\ )$ **and** $(t\text{-}v\text{-}evaluation\ I\ (atom\ (v,\ b)) = Ttrue \wedge 0\leq b \wedge b\leq k)$
and $\forall\, i.(0\leq i \wedge i\leq k) \longrightarrow (t\text{-}v\text{-}evaluation\ I\ (\neg.(atom\ (u,\ i)\ \wedge.\ atom(v,i))) =$ *Ttrue*)
shows $a\neq b$
proof(*rule ccontr*)
assume $\neg\ a \neq b$
hence $a=b$ **by** *auto*
hence $t\text{-}v\text{-}evaluation\ I\ (atom\ (u,\ a)) = Ttrue$ **and** $t\text{-}v\text{-}evaluation\ I\ (atom\ (v,\ a)) = Ttrue$ **using** *assms* **by** *auto*
hence $t\text{-}v\text{-}evaluation\ I\ (atom\ (u,\ a)\ \wedge.\ atom(v,a)) = Ttrue$ **using** *v-conjunction-def* **by** *auto*
hence $t\text{-}v\text{-}evaluation\ I\ (\neg.(atom\ (u,\ a)\ \wedge.\ atom(v,a))) = Ffalse$ **using** *v-negation-def* **by** *auto*
moreover
have $0\leq a \wedge a\leq k$ **using** *assms*(*1*) **by** *auto*
hence $t\text{-}v\text{-}evaluation\ I\ (\neg.(atom\ (u,\ a)\ \wedge.\ atom(v,a))) = Ttrue$ **using** *assms*(*3*) **by** *auto*
finally show *False* **by** *auto*
qed

lemma *distinct-colors*:
assumes *is-graph G* **and** $(u,v) \in E[G]$ **and** *I*: *I model* ($\mathcal{T}$ *G k*)
shows *graph-coloring I k u* $\neq$ *graph-coloring I k v*
proof−
have $u \neq v$ **and** $u \in V[G]$ **and** $v \in V[G]$ **using** ‹$(u,v) \in E[G]$› ‹*is-graph G*›
by(*unfold is-graph-def*, *auto*)

have ∃!*i*. (*t-v-evaluation I* (*atom* (*u*,*i*)) = *Ttrue* ∧ *0*≤*i* ∧ *i*≤*k*) ∧ *graph-coloring I k u* = *i*
using *coloring-function*[*OF* ‹*u* ∈ *V*[*G*]› *I*] **by** *blast*
then obtain *i* **where** *i1*: (*t-v-evaluation I* (*atom* (*u*,*i*)) = *Ttrue* ∧ *0*≤*i* ∧ *i*≤*k*) **and** *i2*: *graph-coloring I k u* = *i*
by *auto*
have ∃!*j*. (*t-v-evaluation I* (*atom* (*v*,*j*)) = *Ttrue* ∧ *0*≤*j* ∧ *j*≤*k*) ∧ *graph-coloring I k v* = *j*
using *coloring-function*[*OF* ‹*v* ∈ *V*[*G*]› *I*] **by** *blast*
then obtain *j* **where** *j1*: (*t-v-evaluation I* (*atom* (*v*,*j*)) = *Ttrue* ∧ *0*≤*j* ∧ *j*≤*k*) **and**
j2: *graph-coloring I k v* = *j* **by** *auto*
have ∀*i*.(*0*≤*i* ∧ *i*≤*k*) ⟶ (¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))∈ $\mathcal{H}$ *G k*)
using ‹*u* ∈ *V*[*G*]› ‹*v* ∈ *V*[*G*]› ‹(*u*,*v*) ∈ *E*[*G*]› **by**(*unfold* $\mathcal{H}$-*def*, *auto*)
hence ∀*i*. (*0*≤*i* ∧ *i*≤*k*) ⟶ ¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*)) ∈ $\mathcal{T}$ *G k*
by(*unfold* $\mathcal{T}$-*def*, *auto*)
hence ∀*i*. (*0*≤*i* ∧ *i*≤*k*) ⟶ (*t-v-evaluation I* (¬.(*atom* (*u*, *i*) ∧. *atom*(*v*,*i*))) = *Ttrue*)
using *assms*(*2*) *I model-def*[*of I* $\mathcal{T}$ *G k*] **by** *blast*
hence *i* ≠ *j* **using** *i1 j1* $\mathcal{H}$*1*[*of I u i k v j*] **by** *blast*
thus *?thesis* **using** *i2 j2* **by** *auto*
qed

theorem *satisfiable-coloring*:
assumes *is-graph G* **and** *satisfiable* ($\mathcal{T}$ *G k*)
shows *colorable G k*
proof(*unfold colorable-def*)
show ∃*f*. *coloring f k G*
proof−
from *assms*(*2*) **have** ∃*I*. *I model* ($\mathcal{T}$ *G k*) **by**(*unfold satisfiable-def*)
then obtain *I* **where** *I*: *I model* ($\mathcal{T}$ *G k*) **by** *auto*
hence *coloring* (*graph-coloring I k*) *k G*
proof(*unfold coloring-def*)
show
(∀*u*. *u* ∈ *V*[*G*] ⟶ (*graph-coloring I k u*) ≤ *k*) ∧ (∀*u v*. (*u*, *v*) ∈ *E*[*G*]
⟶ *graph-coloring I k u* ≠ *graph-coloring I k v*)
proof(*rule conjI*)
show ∀*u*. *u* ∈ *V*[*G*] ⟶ *graph-coloring I k u* ≤ *k*
proof(*rule allI*, *rule impI*)
fix *u*
assume *u* ∈ *V*[*G*]
show *graph-coloring I k u* ≤ *k*
using *coloring-function*[*OF* ‹*u* ∈ *V*[*G*]› *I*] **by** *blast*
qed
next
show
∀*u v*. (*u*, *v*) ∈ *E*[*G*] ⟶
graph-coloring I k u ≠ *graph-coloring I k v*
proof(*rule allI*,*rule allI*,*rule impI*)

```
        fix u v
        assume (u,v) ∈ E[G]
        thus graph-coloring I k u ≠ graph-coloring I k v
        using  distinct-colors[OF ‹is-graph G› ‹(u,v) ∈ E[G]›  I]  by blast
      qed
    qed
  qed
  thus ∃f. coloring f k G by auto
 qed
qed


theorem deBruijn-Erdos-coloring:
 assumes is-graph (G::('vertices:: countable) set × ('vertices × 'vertices) set)
 and ∀H. (is-induced-subgraph H G ∧ finite-graph H ⟶ colorable H k)
 shows colorable G k
proof−
 have ∀ S. S ⊆ (𝒯 G k) ∧ (finite S) ⟶ satisfiable S
 proof(rule allI, rule impI)
  fix S assume S ⊆ (𝒯 G k) ∧ (finite S)
  hence hip1:  S ⊆ (𝒯 G k) and  hip2: finite S by auto
  show satisfiable S
  proof −
   let ?V = vertices-set-formulas S
   let ?H = (?V, E[G] ∩ (?V × ?V))
   have is-induced-subgraph ?H G
    using assms(1) hip1 induced-subgraph[of G S k]
    by(unfold subgraph-aux-def, auto)
   moreover
   have finite-graph ?H
    using assms(1) hip1 hip2 finite-subgraph[of G S k]
    by(unfold subgraph-aux-def, auto)
   ultimately
   have colorable ?H k using assms by auto
   hence  ∃f. coloring f k ?H by(unfold colorable-def, auto)
   then obtain f where coloring f k ?H by auto
   thus satisfiable S using coloring-satisfiable[OF assms(1) hip1]
    by(unfold subgraph-aux-def, auto)
  qed
 qed
 hence satisfiable (𝒯 G k) using
  Compactness-Theorem by auto
 thus ?thesis using assms(1) satisfiable-coloring by blast
qed

end
```

10 König Lemma

This section formalizes König Lemma from the compactness theorem for propositional logic directly.

type-synonym *'a rel* = (*'a* × *'a*) *set*

definition *irreflexive-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where *irreflexive-on A r* ≡ (∀ *x*∈*A*. (*x*, *x*) ∉ *r*)

definition *transitive-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where *transitive-on A r* ≡
(∀ *x*∈*A*. ∀ *y*∈*A*. ∀ *z*∈*A*. (*x*, *y*) ∈ *r* ∧ (*y*, *z*) ∈ *r* ⟶ (*x*, *z*) ∈ *r*)

definition *total-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where *total-on A r* ≡ (∀ *x*∈*A*. ∀ *y*∈*A*. *x* ≠ *y* ⟶ (*x*, *y*) ∈ *r* ∨ (*y*, *x*) ∈ *r*)

definition *minimum* :: *'a set* ⇒ *'a* ⇒*'a rel* ⇒ *bool*
where *minimum A a r* ≡ (*a*∈*A* ∧ (∀ *x*∈*A*. *x* ≠ *a* ⟶ (*a*,*x*) ∈ *r*))

definition *predecessors* :: *'a set* ⇒*'a* ⇒*'a rel* ⇒ *'a set*
where *predecessors A a r* ≡ {*x*∈*A*.(*x*, *a*) ∈ *r*}

definition *height* :: *'a set* ⇒*'a* ⇒ *'a rel* ⇒ *nat*
where *height A a r* ≡ *card* (*predecessors A a r*)

definition *level* :: *'a set* ⇒ *'a rel* ⇒ *nat* ⇒*'a set*
where *level A r n* ≡ {*x*∈*A*. *height A x r* = *n*}

definition *imm-successors* :: *'a set* ⇒ *'a* ⇒ *'a rel* ⇒ *'a set*
where *imm-successors A a r* ≡
{*x*∈*A*. (*a*,*x*)∈ *r* ∧ *height A x r* = (*height A a r*)+*1*}

definition *strict-part-order* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where *strict-part-order A r* ≡ *irreflexive-on A r* ∧ *transitive-on A r*

lemma *minimum-element*:
assumes *strict-part-order A r* **and** *minimum A a r* **and** *r*={}
shows *A*={*a*}
proof(*rule ccontr*)
assume *hip*: *A* ≠ {*a*} **show** *False*
proof(*cases*)
assume *hip1*: *A*={}
have *a*∈*A* **using** ‹*minimum A a r*› **by**(*unfold minimum-def*, *auto*)
thus *False* **using** *hip1* **by** *auto*
next
assume *A* ≠ {}
hence ∃ *x*. *x*≠*a* ∧ *x*∈*A* **using** *hip* **by** *auto*
then obtain *x* **where** *x*≠*a* ∧ *x*∈*A* **by** *auto*
hence (*a*,*x*)∈*r* **using** ‹*minimum A a r*› **by**(*unfold minimum-def*, *auto*)

```
    hence r ≠ {} by auto
    thus False using ‹r={}› by auto
  qed
qed

lemma spo-uniqueness-min:
  assumes  strict-part-order A r and  minimum A a r and minimum A b r
  shows a=b
proof(rule ccontr)
  assume hip: a ≠ b
  have a∈Aand b∈A using assms(2−3) by(unfold minimum-def, auto)
  show False
  proof(cases)
    assume r = {}
     hence  A={a} ∧ A={b}  using assms(1−3) minimum-element[of A r] by
auto
    thus False using hip by auto
  next
    assume r≠{}
    hence 1: (a,b)∈r ∧ (b,a)∈r using hip assms(2−3)
      by(unfold minimum-def, auto)
    have  irr: irreflexive-on A r and tran: transitive-on A r
    using assms(1) by(unfold strict-part-order-def, auto)
    have  (a,a)∈r using  ‹a∈A›  ‹b∈A› 1 tran by(unfold transitive-on-def, blast)
    thus False using  ‹a∈A›  irr  by(unfold irreflexive-on-def, blast)
  qed
qed

lemma emptyness-pred-min-spo:
  assumes  minimum A a r and strict-part-order A r
  shows  predecessors A a r = {}
proof(rule ccontr)
  have  irr:  irreflexive-on A r and tran: transitive-on A r using assms(2)
  by(unfold strict-part-order-def, auto)
  assume 1: predecessors A a r ≠ {} show False
  proof−
    have ∃x∈A. (x,a)∈ r using 1  by(unfold predecessors-def, auto)
    then obtain x where x∈A and (x,a)∈ r by auto
    hence x≠a using irr by (unfold irreflexive-on-def, auto)
   hence (a,x)∈r using  ‹x∈A› ‹minimum A a r› by(unfold minimum-def, auto)
    have  a∈A using  ‹minimum A a r› by(unfold minimum-def, auto)
    hence (a,a)∈r using ‹(a,x)∈r›  ‹(x,a)∈ r›  ‹x∈A›  tran
      by(unfold transitive-on-def, blast)
    thus False using ‹(a,a)∈r› ‹a∈A› irr  irreflexive-on-def
      by (unfold irreflexive-on-def, auto)
  qed
qed

lemma  emptyness-pred-min-spo2:
```

```
  assumes strict-part-order A r and minimum A a r
  shows ∀x∈A.(predecessors A x r = {}) ⟷ (x=a)
proof
  fix x
  assume x ∈ A
  show (predecessors A x r = {}) ⟷ (x = a)
  proof−
    have 1: a ∈ A using ‹minimum A a r› by(unfold minimum-def, auto)
    have 2: (predecessors A x r = {})⟶ (x=a)
    proof(rule impI)
      assume h: predecessors A x r = {} show x=a
      proof(rule ccontr)
      assume x ≠ a
      hence (a,x)∈ r using ‹x ∈ A› ‹minimum A a r›
        by(unfold minimum-def, auto)
      hence a ∈ predecessors A x r
        using 1 by(unfold predecessors-def,auto)
      thus False using h by auto
    qed
  qed
  have 3: x=a ⟶ (predecessors A x r = {})
  proof(rule impI)
    assume x=a
    thus predecessors A x r = {}
      using assms emptyness-pred-min-spo[of A a] by auto
  qed
  show ?thesis using 2 3 by auto
   qed
qed

lemma height-minimum:
  assumes strict-part-order A r and minimum A a r
  shows height A a r = 0
proof−
  have a∈A using ‹minimum A a r› by(unfold minimum-def, auto)
  hence predecessors A a r = {}
    using assms emptyness-pred-min-spo2[of A r] by auto
  thus height A a r = 0 by(unfold height-def, auto)
qed

lemma zero-level:
  assumes strict-part-order A r
  and minimum A a r and ∀x∈A. finite (predecessors A x r)
  shows (level A r 0) = {a}
proof−
  have ∀x∈A.(card (predecessors A x r) = 0) ⟷ (x=a)
  using assms emptyness-pred-min-spo2[of A r a] card-eq-0-iff by auto
  hence 1: ∀x∈A.(height A x r = 0) ⟷ (x=a)
    by(unfold height-def, auto)
```

```
  have a∈A using ‹minimum A a r› by(unfold minimum-def, auto)
  thus ?thesis using assms 1 level-def[of A r 0] by auto
qed

lemma min-predecessor:
  assumes minimum A a r
  shows ∀x∈A. x≠a ⟶ a∈predecessors A x r
proof
  fix x
  assume x∈A
  show x ≠ a ⟶ a ∈ predecessors A x r
  proof(rule impI)
    assume x ≠ a
    show a ∈ predecessors A x r
    proof−
      have (a,x)∈r using ‹x∈A› ‹x ≠ a› ‹minimum A a r›
        by(unfold minimum-def, auto)
      hence a∈A using ‹minimum A a r› by(unfold minimum-def, auto)
      thus a∈predecessors A x r using ‹(a,x)∈r›
        by(unfold predecessors-def, auto)
    qed
  qed
qed

lemma spo-subset-preservation:
  assumes strict-part-order A r and B⊆A
  shows strict-part-order B r
proof−
  have irreflexive-on A r and transitive-on A r
    using ‹strict-part-order A r›
    by(unfold strict-part-order-def, auto)
  have 1: irreflexive-on B r
  proof(unfold irreflexive-on-def)
    show ∀x∈B. (x, x) ∉ r
    proof
      fix x
      assume x∈B
      hence x∈A using ‹B⊆A› by auto
      thus (x,x)∉r using ‹irreflexive-on A r›
        by (unfold irreflexive-on-def, auto)
    qed
  qed
  have 2: transitive-on B r
  proof(unfold transitive-on-def)
    show ∀x∈B. ∀y∈B. ∀z∈B. (x, y) ∈ r ∧ (y, z) ∈ r ⟶ (x, z) ∈ r
    proof
      fix x assume x∈B
      show ∀y∈B. ∀z∈B. (x, y) ∈ r ∧ (y, z) ∈ r ⟶ (x, z) ∈ r
      proof
```

```
      fix y  assume y∈B
      show ∀z∈B. (x, y) ∈ r ∧ (y, z) ∈ r ⟶ (x, z) ∈ r
      proof
        fix z  assume z∈B
        show (x, y) ∈ r ∧ (y, z) ∈ r ⟶ (x, z) ∈ r
        proof(rule impI)
          assume hip: (x, y) ∈ r ∧ (y, z) ∈ r
          show (x, z) ∈ r
        proof−
          have x∈A and  y∈A and  z∈A using ‹x∈B› ‹y∈B› ‹z∈B› ‹B⊆A›
            by auto
        thus (x, z) ∈ r using hip ‹transitive-on A  r› by(unfold transitive-on-def,
blast)
          qed
        qed
      qed
    qed
  qed
qed
thus strict-part-order B r
  using 1 2  by(unfold strict-part-order-def, auto)
qed

lemma total-ord-subset-preservation:
  assumes total-on A r and  B⊆A
  shows total-on B r
proof(unfold total-on-def)
  show  ∀x∈B. ∀y∈B. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
  proof
    fix x
    assume x∈B show  ∀y∈B. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
    proof
      fix y
      assume y∈B
      show  x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
      proof(rule impI)
        assume  x ≠ y
        show (x, y) ∈ r ∨ (y, x) ∈ r
        proof−
          have x∈A ∧ y∈A using  ‹x∈B›  ‹y∈B› ‹B⊆A› by auto
          thus  (x, y) ∈ r ∨ (y, x) ∈ r
            using  ‹x ≠ y› ‹total-on A r› by(unfold total-on-def, auto)
        qed
      qed
    qed
  qed
qed

definition maximum ::  'a set ⇒ 'a ⇒'a rel ⇒ bool
```

```
  where maximum A a r ≡  (a∈A ∧ (∀x∈A. x ≠ a ⟶ (x,a) ∈ r))

lemma maximum-strict-part-order:
  assumes strict-part-order A r and A≠{} and total-on A r
  and finite A
  shows (∃a. maximum A a r)
proof−
  have strict-part-order A r ⟹ A≠{} ⟹ total-on A r ⟹ finite A
  ⟹ (∃a. maximum A a r)  using assms(4)
  proof(induct A rule:finite-induct)
    case empty
    then show ?case by auto
  next
    case (insert x A)
    show (∃a. maximum (insert x A) a r)
  proof(cases A={})
    case True
    hence insert x A ={x} by simp
    hence  maximum (insert x A) x r by(unfold maximum-def, auto)
    then show ?thesis by auto
  next
    case False
    assume A ≠ {}
    show ∃a. maximum (insert x A) a r
    proof−
      have 1: strict-part-order A r
        using insert(4) spo-subset-preservation by auto
      have 2: total-on A r using insert(6) total-ord-subset-preservation by auto
      have ∃a. maximum A a r using 1  ‹A≠{}›  insert(1) 2 insert(3) by auto
      then obtain a where a: maximum A a r by auto
     hence  a∈A and ∀y∈A. y ≠ a  ⟶ (y,a) ∈ r by(unfold maximum-def, auto)
      have 3: a∈(insert x A) using ‹a∈A›  by auto
      have 4: a≠x using ‹a∈A› and  ‹x ∉ A› by auto
      have  x∈(insert x A) by auto
      hence (a,x)∈r ∨ (x,a)∈r using 3 4 ‹total-on (insert x A) r›
        by(unfold total-on-def, auto)
      thus ∃a. maximum (insert x A) a r
      proof(rule disjE)
        have  transitive-on (insert x A) r using  insert(4)
          by(unfold strict-part-order-def, auto)
        assume casoa: (a, x) ∈ r
        have  ∀z∈(insert x A). z ≠ x  ⟶ (z,x) ∈ r
        proof
          fix z
          assume hip1: z ∈ (insert x A)
          show z ≠ x ⟶ (z, x) ∈ r
          proof(rule impI)
            assume z ≠ x
            hence hip2:  z∈A using ‹z ∈ (insert x A)› by auto
```

```
          thus (z, x) ∈ r
          proof(cases)
            assume z=a
            thus (z, x) ∈ r using ‹(a, x) ∈ r› by auto
          next
            assume z≠a
            hence (z,a) ∈ r using ‹z∈A› ‹∀ y∈A. y ≠ a ⟶ (y,a) ∈ r› by auto
            have a∈(insert x A) and z∈(insert x A) and x∈(insert x A)
              using ‹a∈A› ‹z∈A› by auto
            thus (z, x) ∈ r
              using ‹(z,a) ∈ r› ‹(a, x) ∈ r› ‹transitive-on (insert x A) r›
              by(unfold transitive-on-def, blast)
          qed
        qed
      qed
      thus ∃ a. maximum (insert x A) a r
        using ‹x∈(insert x A)› by(unfold maximum-def, auto)
    next
      assume casob: (x, a) ∈ r
      have ∀ z∈(insert x A). z ≠ a ⟶ (z,a) ∈ r
      proof
        fix z
        assume hip1: z ∈ (insert x A)
        show z ≠ a ⟶ (z, a) ∈ r
        proof(rule impI)
          assume z ≠ a show (z, a) ∈ r
          proof−
            have z∈A ∨ z=x using ‹z ∈ (insert x A)› by auto
            thus (z, a) ∈ r
            proof(rule disjE)
              assume z ∈ A
              thus (z, a) ∈ r
                using ‹z ≠ a› ‹∀ y∈A. y ≠ a ⟶ (y,a) ∈ r› by auto
            next
              assume z = x
              thus (z, a) ∈ r using ‹(x, a) ∈ r› by auto
            qed
          qed
        qed
      qed
      thus ∃ a. maximum (insert x A) a r
        using ‹a∈(insert x A)› by(unfold maximum-def, auto)
    qed
  qed
 qed
qed
 thus ?thesis using assms by auto
qed
```

lemma *finiteness-union-finite-sets*:
  fixes S :: $'a \Rightarrow 'a$ *set*
  assumes $\forall x.$ *finite* $(S\ x)$ **and** *finite* A
  shows *finite* $(\bigcup a{\in}A.\ (S\ a))$ **using** *assms* **by** *auto*

lemma *uniqueness-level-aux*:
  assumes $k{>}0$
  shows $(level\ A\ r\ n) \cap (level\ A\ r\ (n{+}k)) = \{\}$
proof(*rule ccontr*)
  assume $level\ A\ r\ n \cap level\ A\ r\ (n + k) \neq \{\}$
  hence $\exists x.\ x{\in}(level\ A\ r\ n) \cap level\ A\ r\ (n + k)$ **by** *auto*
  then obtain x **where** $x{\in}(level\ A\ r\ n) \cap level\ A\ r\ (n + k)$ **by** *auto*
  hence $x{\in}A \wedge height\ A\ x\ r = n$ **and** $x{\in}A \wedge height\ A\ x\ r = n{+}k$
    by(*unfold level-def*, *auto*)
  thus *False* **using** ‹$k{>}0$› **by** *auto*
qed

lemma *uniqueness-level*:
  assumes $n{\neq}m$
  shows $(level\ A\ r\ n) \cap (level\ A\ r\ m) = \{\}$
proof−
  have $n < m \vee m < n$ **using** *assms* **by** *auto*
  thus *?thesis*
  proof(*rule disjE*)
    assume $n < m$
    hence $\exists k.\ k{>}0 \wedge m{=}n{+}k$ **by** *arith*
    thus *?thesis* **using** *uniqueness-level-aux*[*of* - *A r*] **by** *auto*
  next
    assume $m < n$
    hence $\exists k.\ k{>}0 \wedge n{=}m{+}k$ **by** *arith*
    thus *?thesis* **using** *uniqueness-level-aux*[*of* - *A r*] **by** *auto*
  qed
qed

definition *tree* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$
  where $tree\ A\ r \equiv$
 $r \subseteq A \times A \wedge r{\neq}\{\} \wedge (strict\text{-}part\text{-}order\ A\ r) \wedge (\exists a.\ minimum\ A\ a\ r) \wedge$
 $(\forall a{\in}A.\ finite\ (predecessors\ A\ a\ r) \wedge (total\text{-}on\ (predecessors\ A\ a\ r)\ r))$

definition *finite-tree*:: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$
  where
$finite\text{-}tree\ A\ r \equiv tree\ A\ r \wedge finite\ A$

abbreviation *infinite-tree*:: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$
  where
$infinite\text{-}tree\ A\ r \equiv tree\ A\ r \wedge \neg\ finite\ A$

definition *enumerable-tree* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$ **where**
 $enumerable\text{-}tree\ A\ r \equiv \exists g.\ enumeration\ (g{::}\ nat \Rightarrow 'a)$

definition *finitely-branching* :: *'a set* ⇒ *'a rel* ⇒ *bool*
  where *finitely-branching A r* ≡ (∀ *x*∈*A*. *finite* (*imm-successors A x r*))

definition *sub-linear-order* :: *'a set* ⇒ *'a set* ⇒ *'a rel* ⇒ *bool*
  where *sub-linear-order B A r* ≡ *B*⊆*A* ∧ (*strict-part-order A r*) ∧ (*total-on B r*)

definition *path* :: *'a set* ⇒ *'a set* ⇒ *'a rel* ⇒ *bool*
  where *path B A r* ≡
 (*sub-linear-order B A r*) ∧
 (∀ *C*. *B* ⊆ *C* ∧ *sub-linear-order C A r* ⟶ *B* = *C*)

definition *finite-path*:: *'a set* ⇒ *'a set* ⇒ *'a rel* ⇒ *bool*
  where *finite-path B A r* ≡ *path B A r* ∧ *finite B*

definition *infinite-path*:: *'a set* ⇒ *'a set* ⇒ *'a rel* ⇒ *bool*
  where *infinite-path B A r* ≡ *path B A r* ∧ ¬ *finite B*

lemma *tree*:
  assumes *tree A r*
  shows
  r ⊆ *A* × *A* **and** *r*≠{}
  and *strict-part-order A r*
  and ∃ *a*. *minimum A a r*
  and (∀ *a*∈*A*. *finite* (*predecessors A a r*) ∧ (*total-on* (*predecessors A a r*) *r*))
  using ‹*tree A r*› **by**(*unfold tree-def*, *auto*)

lemma *non-empty*:
  assumes *tree A r* **shows** *A*≠{}
proof−
  have ∃ *a*. *minimum A a r* **using** ‹*tree A r*› *tree*[*of A r*] **by** *auto*
  hence ∃ *a*. *a*∈*A* **by**(*unfold minimum-def*, *auto*)
  thus *A*≠{} **by** *auto*
qed

lemma *predecessors-spo*:
  assumes *tree A r*
  shows ∀ *x*∈*A*. *strict-part-order* (*predecessors A x r*) *r*
proof−
  have *irreflexive-on A r* **and** *transitive-on A r* **using** ‹*tree A r*›
    by(*unfold tree-def*,*unfold strict-part-order-def*,*auto*)
  thus *?thesis*
proof(*unfold strict-part-order-def*)
  show ∀ *x*∈*A*. *irreflexive-on* (*predecessors A x r*) *r* ∧
      transitive-on (*predecessors A x r*) *r*
  proof
    fix *x*
    assume *x*∈*A*

```
  show irreflexive-on (predecessors A x r) r ∧ transitive-on (predecessors A x r)
r
  proof−
    have 1: irreflexive-on (predecessors A x r) r
    proof(unfold irreflexive-on-def)
      show ∀ y∈(predecessors A x r). (y, y) ∉ r
      proof
        fix y
        assume y∈(predecessors A x r)
        hence y∈A by(unfold predecessors-def,auto)
      thus (y, y) ∉ r using ‹irreflexive-on A r› by(unfold irreflexive-on-def,auto)
      qed
    qed
    have 2: transitive-on (predecessors A x r) r
    proof(unfold transitive-on-def)
      let ?B= (predecessors A x r)
      show ∀ w∈?B. ∀ y∈?B. ∀ z∈?B. (w, y) ∈ r ∧ (y, z) ∈ r ⟶ (w, z) ∈ r
      proof
        fix w assume w∈?B
       show ∀ y∈?B. ∀ z∈?B. (w, y) ∈ r ∧ (y, z) ∈ r ⟶ (w, z) ∈ r
       proof
         fix y assume y∈?B
         show ∀ z∈?B. (w, y) ∈ r ∧ (y, z) ∈ r ⟶ (w, z) ∈ r
         proof
           fix z  assume z∈?B
           show (w, y) ∈ r ∧ (y, z) ∈ r ⟶ (w, z) ∈ r
           proof(rule impI)
             assume hip: (w, y) ∈ r ∧ (y, z) ∈ r
             show (w, z) ∈ r
             proof−
               have  w∈A and  y∈A and  z∈A using ‹w∈?B› ‹y∈?B› ‹z∈?B›
                 by(unfold predecessors-def,auto)
               thus (w, z) ∈ r
                 using hip ‹transitive-on A  r› by(unfold transitive-on-def, blast)
               qed
             qed
           qed
         qed
       qed
     qed
     show
      irreflexive-on (predecessors A x r) r ∧ transitive-on (predecessors A x r) r
     using 1 2 by auto
     qed
    qed
  qed
qed
```

lemma *predecessors-maximum*:

assumes *tree A r* **and** *minimum A a r*
shows ∀ *x*∈*A*. *x*≠*a* ⟶ (∃ *b*. *maximum* (*predecessors A x r*) *b r*)
proof
fix *x*
assume *x*∈*A*
show *x*≠*a* ⟶ (∃ *b*. *maximum* (*predecessors A x r*) *b r*)
proof(*rule impI*)
assume *x*≠*a*
show (∃ *b*. *maximum* (*predecessors A x r*) *b r*)
proof−
have *1*: *strict-part-order* (*predecessors A x r*) *r*
using ‹*tree A r*› ‹*x*∈*A*› *predecessors-spo* **by** *auto*
have *2*: *total-on* (*predecessors A x r*) *r* **and**
3: *finite* (*predecessors A x r*) **and** *r* ⊆ *A* × *A*
using ‹*tree A r*› ‹*x*∈*A*› **by**(*unfold tree-def*, *auto*)
have *4*: (*predecessors A x r*)≠{}
using ‹*r* ⊆ *A* × *A*› ‹*minimum A a r*› ‹*x*∈*A*› ‹*x*≠*a*›
min-predecessor[*of A a*] **by** *auto*
have *5*: *A*≠{} **using** ‹*tree A r*› *non-empty* **by** *auto*
show (∃ *b*. *maximum* (*predecessors A x r*) *b r*)
using *1 2 3 4 5 maximum-strict-part-order* **by** *auto*
qed
qed
qed

lemma *non-empty-preds-in-tree*:
assumes *tree A r* **and** *card* (*predecessors A x r*) = *n+1*
shows *x*∈*A*
proof−
have *r* ⊆ *A* × *A* **using** ‹*tree A r*› **by**(*unfold tree-def*, *auto*)
have (*predecessors A x r*) ≠ {} **using** *assms*(*2*) **by** *auto*
hence ∃ *y*∈*A*. (*y*,*x*)∈*r* **by** (*unfold predecessors-def*,*auto*)
thus *x*∈*A* **using** ‹*r* ⊆ *A* × *A*› **by** *auto*
qed

lemma *imm-predecessor*:
assumes *tree A r*
and *card* (*predecessors A x r*) = *n+1* **and**
maximum (*predecessors A x r*) *b r*
shows *height A b r* = *n*
proof−
have *transitive-on A r* **and** *r* ⊆ *A* × *A* **and** *irreflexive-on A r*
using ‹*tree A r*›
by (*unfold tree-def*, *unfold strict-part-order-def*, *auto*)
have *x*∈*A* **using** *assms*(*1*) *assms*(*2*) *non-empty-preds-in-tree* **by** *auto*
have *strict-part-order* (*predecessors A x r*) *r*
using ‹*x*∈*A*› ‹*tree A r*› *predecessors-spo*[*of A r*] **by** *auto*
hence *irreflexive-on* (*predecessors A x r*) *r* **and**
transitive-on (*predecessors A x r*) *r*

```
    by(unfold strict-part-order-def, auto)
  have b∈(predecessors A x r)
    using ‹maximum (predecessors A x r) b r› by(unfold maximum-def, auto)
  have total-on (predecessors A x r) r
    using ‹x∈A› ‹tree A r› by(unfold tree-def, auto)
  have card (predecessors A x r)>0  using assms(2) by auto
  hence 1: finite (predecessors A x r)  using card-gt-0-iff by blast
  have  2: b∈(predecessors A x r)
    using assms(3) by (unfold maximum-def,auto)
  hence card ((predecessors A x r)−{b}) = n
    using 1  ‹card (predecessors A x r) = n+1›
    card-Diff-singleton[of b (predecessors A x r) ] by auto
  have (predecessors A b r) = ((predecessors A x r)−{b})
  proof(rule equalityI)
    show (predecessors A b r) ⊆ (predecessors A x r − {b})
    proof
      fix y
      assume  y∈ (predecessors A b r)
      hence y∈A and  (y,b)∈ r by (unfold predecessors-def,auto)
      hence y≠b using ‹irreflexive-on A r› by(unfold irreflexive-on-def,auto)
      have (b,x)∈r using 2 by (unfold predecessors-def,auto)
      hence b∈A  using ‹r ⊆ A × A› by auto
     have (y,x)∈ r using ‹x∈A› ‹y∈A› ‹b∈A›  ‹(y,b)∈ r›  ‹(b,x)∈r› ‹transitive-on
A r›
        by(unfold transitive-on-def, blast)
      show y∈(predecessors A x r − {b})
        using ‹y∈A› ‹(y,x)∈ r› ‹y≠b› by(unfold predecessors-def, auto)
    qed
  next
    show (predecessors A x r − {b}) ⊆ (predecessors A b r)
    proof
      fix y
      assume hip: y∈(predecessors A x r − {b})
      hence y≠b and  y∈A by(unfold predecessors-def, auto)
      have (y,b)∈ r using hip ‹maximum (predecessors A x r) b r›
        by(unfold maximum-def,auto)
      thus y∈ (predecessors A b r) using ‹y∈A›
        by(unfold predecessors-def, auto)
    qed
  qed
  hence 3:  card (predecessors A b r) = card (predecessors A x r − {b})
    by auto
  have finite (predecessors A x r) using ‹x∈A› ‹tree A r› by(unfold tree-def,auto)
  hence card (predecessors A x r − {b}) = card (predecessors A x r)−1
    using 2  card-Suc-Diff1 by auto
  hence card (predecessors A b r) = n
    using 3  ‹card (predecessors A x r) = n+1› by auto
  thus height A b r = n by (unfold height-def, auto)
qed
```

lemma *height*:
  assumes *tree A r* **and** *height A x r = n+1*
  shows ∃ *y*. (*y*,*x*)∈*r* ∧ *height A y r = n*
proof −
  have *1*: *card* (*predecessors A x r*) = *n+1*
    using *assms*(*2*) **by** (*unfold height-def*, *auto*)
  have ∃ *a*. *minimum A a r* **using** ‹*tree A r*› **by**(*unfold tree-def*, *auto*)
  then obtain *a* **where** *a*: *minimum A a r* **by** *auto*
  have *strict-part-order A r* **using** ‹*tree A r*› *tree*[*of A r*] **by** *auto*
  hence *height A a r = 0* **using** *a height-minimum*[*of A r*] **by** *auto*
  hence *x* ≠ *a* **using** *assms*(*2*) **by** *auto*
  have *x*∈*A* **using** ‹*tree A r*› *1 non-empty-preds-in-tree* **by** *auto*
  hence (∃ *b*. *maximum* (*predecessors A x r*) *b r*)
    using ‹*x* ≠ *a*› ‹*tree A r*› *a predecessors-maximum*[*of A r a*] **by** *auto*
  then obtain *b* **where** *b*: (*maximum* (*predecessors A x r*) *b r*) **by** *auto*
  hence (*b*,*x*)∈*r* **by**(*unfold maximum-def*, *unfold predecessors-def*,*auto*)
  thus ∃ *y*. (*y*,*x*)∈*r* ∧ *height A y r = n*
    using ‹*tree A r*› *1 b imm-predecessor*[*of A r*] **by** *auto*
qed

lemma *level*:
  assumes *tree A r* **and** *x* ∈ (*level A r* (*n+1*))
  shows ∃ *y*. (*y*,*x*)∈*r* ∧ *y* ∈ (*level A r n*)
proof−
  have *height A x r = n+1*
    using ‹*x*∈ (*level A r* (*n+1*))› **by** (*unfold level-def*, *auto*)
  hence ∃ *y*. (*y*,*x*)∈*r* ∧ *height A y r = n*
    using ‹*tree A r*› *height*[*of A r*] **by** *auto*
  then obtain *y* **where** *y*: (*y*,*x*)∈*r* ∧ *height A y r = n* **by** *auto*
  have *r* ⊆ *A* × *A* **using** ‹*tree A r*› **by**(*unfold tree-def*,*auto*)
  hence *y*∈*A* **using** *y* **by** *auto*
  hence (*y*,*x*)∈*r* ∧ *y* ∈ (*level A r n*) **using** *y* **by**(*unfold level-def*, *auto*)
  thus *?thesis* **by** *auto*
qed

primrec *set-nodes-at-level* :: *'a set* ⇒ *'a rel* ⇒ *nat* ⇒*'a set* **where**
set-nodes-at-level A r 0 = {*a*. (*minimum A a r*)}
| *set-nodes-at-level A r* (*Suc n*) = (⋃ *a*∈ (*set-nodes-at-level A r n*). *imm-successors A a r*)

lemma *set-nodes-at-level-zero-spo*:
  assumes *strict-part-order A r* **and** *minimum A a r*
  shows (*set-nodes-at-level A r 0*) = {*a*}
proof−
  have *a*∈(*set-nodes-at-level A r 0*) **using** ‹*minimum A a r*› **by** *auto*
  hence *1*: {*a*} ⊆ (*set-nodes-at-level A r 0*) **by** *auto*
  have *2*: (*set-nodes-at-level A r 0*) ⊆ {*a*}
  proof

```
    {fix x
    assume x∈(set-nodes-at-level A r 0)
    hence minimum A x r by auto
    hence x=a using assms spo-uniqueness-min[of A r] by auto
    thus x∈{a} by auto}
  qed
  thus (set-nodes-at-level A r 0) = {a} using 1 2 by auto
qed

lemma height-level:
  assumes strict-part-order A r and minimum A a r
  and x ∈ set-nodes-at-level A r n
  shows height A x r = n
proof−
  have
 ⟦strict-part-order A r; minimum A a r; x ∈ set-nodes-at-level A r n⟧ ⟹
  height A x r = n
  proof(induct n arbitrary: x)
    case 0
    then show height A x r = 0
    proof−
      have minimum A x r using ‹x ∈ set-nodes-at-level A r 0› by auto
      thus height A x r = 0
        using ‹strict-part-order A r› height-minimum[of A r]
        by auto
    qed
  next
    case (Suc n)
    then show ?case
    proof−
      have x∈ (⋃ a ∈ (set-nodes-at-level A r n). (imm-successors A a r))
        using Suc(4) by auto
      then obtain a
        where hip1: a ∈ (set-nodes-at-level A r n) and hip2: x∈ (imm-successors
A a r)
        by auto
      hence 1: height A a r = n using Suc(1−3) by auto
      have height A x r = (height A a r)+1
        using hip2 by(unfold imm-successors-def, auto)
      thus height A x r = Suc n using 1 by auto
    qed
  qed
  thus ?thesis using assms by auto
qed

lemma level-func-vs-level-def:
  assumes tree A r
  shows set-nodes-at-level A r n = level A r n
proof(induct n)
```

```
  have 1: strict-part-order A r and
      2: ∀ x∈A. finite (predecessors A x r)
    using ‹tree A r› tree[of A r] by auto
  have ∃ a. minimum A a r using ‹tree A r› by(unfold tree-def, auto)
  then obtain a where a: minimum A a r by auto
  case 0
  then show set-nodes-at-level A r 0 = level A r 0
  proof−
    have set-nodes-at-level A r 0 = {a} using 1 a set-nodes-at-level-zero-spo[of A
r] by auto
    moreover
    have level A r 0 = {a} using 1 2 a zero-level[of A r] by auto
    ultimately
    show set-nodes-at-level A r 0 = level A r 0 by auto
  qed
  next
    case (Suc n)
    assume set-nodes-at-level A r n = level A r n
    show set-nodes-at-level A r (Suc n) = level A r (Suc n)
    proof(rule equalityI)
      show set-nodes-at-level A r (Suc n) ⊆ level A r (Suc n)
      proof(rule subsetI)
        fix x
        assume hip: x ∈ set-nodes-at-level A r (Suc n) show x ∈ level A r (Suc n)
        proof−
          have
              set-nodes-at-level A r (Suc n) = (⋃ a ∈ (set-nodes-at-level A r n).
(imm-successors A a r))
            by simp
          hence x∈ (⋃ a ∈ (set-nodes-at-level A r n). (imm-successors A a r))
            using hip by auto
          then obtain a where hip1: a ∈ (set-nodes-at-level A r n) and
            hip2:x∈ (imm-successors A a r) by auto
          have (a,x)∈r ∧ height A x r = (height A a r)+1
            using hip2 by(unfold imm-successors-def, auto)
          moreover
          have ∃ b. minimum A b r using ‹tree A r› by(unfold tree-def, auto)
         then obtain b where b: minimum A b r by auto
         have 1: r ⊆ A × A and strict-part-order A r
           using ‹tree A r› by(unfold tree-def, auto)
         hence height A a r = n using b hip1 height-level[of A r] by auto
         ultimately
         have (a,x)∈r ∧ height A x r = n+1 by auto
         hence x∈A ∧ height A x r = n+1 using ‹r ⊆ A × A› by auto
         thus x ∈ level A r (Suc n) by(unfold level-def, auto)
       qed
     qed
  next
    show level A r (Suc n) ⊆ set-nodes-at-level A r (Suc n)
```

proof(*rule subsetI*)
  fix *x*
  assume *hip*: *x* ∈ *level A r* (*Suc n*) **show** *x* ∈ *set-nodes-at-level A r* (*Suc n*)
  proof−
    have *1*: *x*∈*A* ∧ *height A x r* = *n*+*1* **using** *hip* **by**(*unfold level-def*,*auto*)
    hence ∃ *y*. (*y*,*x*)∈*r* ∧ *height A y r* = *n*
    using *assms height*[*of A r*] **by** *auto*
    then obtain *y* **where** *y1*: (*y*,*x*)∈*r* **and** *y2*: *height A y r* = *n* **by** *auto*
    hence *x* ∈ (*imm-successors A y r*)
      using *1* **by**(*unfold imm-successors-def*, *auto*)
    moreover
    have *r* ⊆ *A* × *A* **using** ‹*tree A r*› **by**(*unfold tree-def*, *auto*)
    have *y*∈*A* **using** *y1* ‹*r* ⊆ *A* × *A*› **by** *auto*
    hence *y*∈ *level A r n* **using** *y2* **by**(*unfold level-def*, *auto*)
    hence *y*∈ *set-nodes-at-level A r n* **using** *Suc* **by** *auto*
    ultimately
    show *x* ∈ *set-nodes-at-level A r* (*Suc n*) **by** *auto*
  qed
qed
qed
qed

lemma *pertenece-level*:
assumes *x* ∈ *set-nodes-at-level A r n*
shows *x*∈*A*
proof−
have *x* ∈ *set-nodes-at-level A r n* ⟹ *x*∈*A*
proof(*induct n*)
  case *0*
  show *x* ∈ *A* **using** ‹*x* ∈ *set-nodes-at-level A r 0*› *minimum-def*[*of A x r*] **by** *auto*
next
  case (*Suc n*)
  then show *x* ∈ *A*
  proof−
    have ∃ *a* ∈ (*set-nodes-at-level A r n*). *x*∈ *imm-successors A a r*
      using ‹*x* ∈ *set-nodes-at-level A r* (*Suc n*)› **by** *auto*
    then obtain *a* **where** *a1*: *a* ∈ (*set-nodes-at-level A r n*) **and**
      a2: *x*∈ *imm-successors A a r* **by** *auto*
    show *x* ∈ *A* **using** *a2 imm-successors-def*[*of A a r*] **by** *auto*
  qed
qed
thus *x* ∈ *A* **using** *assms* **by** *auto*
qed

lemma *finiteness-set-nodes-at-levela*:
assumes ∀ *x*∈*A*. *finite* (*imm-successors A x r*) **and** *finite* (*set-nodes-at-level A r n*)
shows *finite* (⋃ *a*∈ (*set-nodes-at-level A r n*). *imm-successors A a r*)

```
proof
  show finite (set-nodes-at-level A r n) using assms(2) by simp
next
  fix x
  assume hip: x ∈ set-nodes-at-level A r n show finite (imm-successors A x r)
  proof−
    have x∈A using hip pertenece-level[of x A r] by auto
    thus finite (imm-successors A x r) using assms(1) by auto
  qed
qed

lemma finiteness-set-nodes-at-level:
  assumes finite (set-nodes-at-level A r 0) and finitely-branching A r
  shows finite (set-nodes-at-level A r n)
proof(induct n)
  case 0
  show finite (set-nodes-at-level A r 0) using assms by auto
next
  case (Suc n)
  then show ?case
  proof −
    have 1: ∀x∈A. finite (imm-successors A x r)
      using assms by (unfold finitely-branching-def, auto)
    hence finite (⋃a∈ (set-nodes-at-level A r n). imm-successors A a r)
      using Suc(1) finiteness-set-nodes-at-levela[of A r] by auto
    thus finite (set-nodes-at-level A r (Suc n)) by auto
  qed
qed

lemma finite-level:
  assumes tree A r and finitely-branching A r
  shows finite (level A r n)
proof−
  have 1: strict-part-order A r using ‹tree A r› tree[of A r] by auto
  have ∃a. minimum A a r using ‹tree A r› tree[of A r] by auto
  then obtain a where minimum A a r by auto
  hence finite (set-nodes-at-level A r 0)
    using 1 set-nodes-at-level-zero-spo[of A r] by auto
  hence finite (set-nodes-at-level A r n)
    using ‹finitely-branching A r› finiteness-set-nodes-at-level[of A r] by auto
  thus ?thesis using ‹tree A r› level-func-vs-level-def[of A r n] by auto
qed

lemma finite-level-a:
  assumes tree A r and ∀n. finite (level A r n)
  shows finitely-branching A r
proof(unfold finitely-branching-def)
  show ∀x∈A. finite (imm-successors A x r)
  proof
```

```
  fix x
  assume x∈A
  show finite (imm-successors A x r) using finitely-branching-def
  proof−
    let ?n = (height A x r)
    have (imm-successors A x r) ⊆ (level A r (?n+1))
      using imm-successors-def[of A x r] level-def[of A r ?n+1] by auto
   thus finite (imm-successors A x r)  using assms(2) by(simp add: finite-subset)

  qed
qed
qed

lemma empty-predec:
  assumes ∀x∈A. (x,y)∉r
  shows predecessors A y r ={}
    using assms by(unfold predecessors-def, auto)

lemma level-element:
 ∀x∈A.∃n. x∈ level A r n
proof
  fix x
  assume hip: x∈A show ∃n. x ∈ level A r n
  proof−
    let ?n = height A x r
    have x∈level A r ?n using ‹x∈A›  by (unfold level-def, auto)
    thus ∃n. x ∈ level A r n by auto
  qed
qed

lemma union-levels:
  shows A =(⋃n. level A r n)
proof(rule equalityI)
  show A ⊆ (⋃n. level A r n)
  proof(rule subsetI)
    fix x
    assume hip: x∈A show x∈(⋃n. level A r n)
    proof−
      have ∃n. x∈ level A r n
        using hip level-element[of A] by auto
      then obtain n where  x∈ level A r n by auto
    thus ?thesis by auto
  qed
qed
next
  show  (⋃n. level A r n) ⊆ A
  proof(rule subsetI)
    fix x
    assume hip:  x ∈ (⋃n. level A r n) show x ∈ A
```

```
    proof−
      obtain n where  x∈ level A r n using hip by auto
      thus x ∈ A by(unfold level-def, auto)
    qed
  qed
qed

lemma path-to-node:
  assumes  tree A r  and  x ∈ (level A r (n+1))
  shows ∀ k.(0≤k ∧ k≤n)⟶ (∃ y. (y,x)∈r ∧ y ∈ (level A r k))
proof−
  have tree A r ⟹ x ∈ (level A r (n+1)) ⟹
  ∀ k.(0≤k ∧ k≤n)⟶ (∃ y. (y,x)∈r ∧ y ∈ (level A r k))
  proof(induction n arbitrary: x)
    have r ⊆ A × A and 1:  strict-part-order A r
    and ∃ a. minimum A a r
    and 2: ∀ x∈A. finite (predecessors A x r)
      using ‹tree A r› tree[of A r] by auto
    case 0
    show  ∀ k. 0 ≤ k ∧ k ≤ 0 ⟶ (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
    proof
      fix k
      show 0 ≤ k ∧ k ≤ 0 ⟶ (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
      proof(rule impI)
        assume hip:  0 ≤ k ∧ k ≤ 0
        show (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
        proof−
          have k=0 using hip  by auto
          thus (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
            using ‹tree A r›  ‹x ∈ (level A r (0 + 1))› level[of A r ]  by auto
        qed
      qed
    qed
    next
      case (Suc n)
      show ∀ k. 0 ≤ k ∧ k ≤ Suc n ⟶ (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
  proof(rule allI, rule impI)
    fix k
    assume hip:  0 ≤ k ∧ k ≤ Suc n
    show  (∃ y. (y, x) ∈ r ∧ y ∈ level A r k)
    proof−
      have (0 ≤ k ∧ k ≤ n) ∨ k = Suc n  using hip by auto
      thus ?thesis
      proof(rule disjE)
        assume hip1:  0 ≤ k ∧ k ≤ n
        have ∃ y. (y,x)∈r ∧ y ∈ (level A r (n+1))
        using ‹tree A r› level  ‹x ∈ level A r (Suc n + 1)› by auto
        then obtain y where y1: (y,x)∈r and y2: y ∈ (level A r (n+1))
          by auto
```

```
        have ∀k. 0 ≤ k ∧ k ≤ n ⟶ (∃z. (z, y) ∈ r ∧ z ∈ level A r k)
          using y2  Suc(1−3) by auto
        hence (∃z. (z, y) ∈ r ∧ z ∈ level A r k)
          using hip1 by auto
        then obtain z where  z1: (z, y) ∈ r and z2: z ∈ (level A r k) by auto
        have  r ⊆ A × A and strict-part-order A r
          using  ‹tree A r› tree by auto
        hence z∈A and  y∈A and x∈A
          using ‹r ⊆ A × A› ‹(z, y) ∈ r› ‹(y,x)∈r› by auto
        have transitive-on A r using ‹strict-part-order A r›
          by(unfold strict-part-order-def, auto)
        hence (z, x) ∈ r using ‹z∈A› ‹y∈A› and ‹x∈A› ‹(z, y) ∈ r› ‹(y,x)∈r›
          by(unfold transitive-on-def, blast)
        thus (∃y. (y, x) ∈ r ∧ y ∈ level A r k)
          using z2 by auto
      next
        assume  k = Suc n
        thus  ∃y. (y,x)∈r ∧ y ∈ (level A r k)
          using ‹tree A r› level ‹x ∈ level A r (Suc n + 1)› by auto
        qed
      qed
    qed
  qed
  thus ?thesis using assms by auto
qed

lemma set-nodes-at-level:
  assumes tree A r
  shows (level A r (n+1))≠ {} ⟶ (∀k.(0≤k ∧ k≤n) ⟶ (level A r k)≠ {})
proof(rule impI)
  assume hip:  (level A r (n+1))≠ {}
    show  (∀k.(0≤k ∧ k≤n) ⟶ (level A r k)≠ {})
    proof−
      have  ∃x. x∈(level A r (n+1)) using hip by auto
      then obtain x where x: x∈(level A r (n+1)) by auto
      thus ?thesis using assms path-to-node[of A r] by blast
    qed
  qed

lemma emptyness-below-height:
  assumes  tree A r
  shows  ((level A r (n+1)) = {}) ⟶ (∀k. k>(n+1) ⟶ (level A r k) = {})
proof(rule ccontr)
  assume hip: ¬ (level A r (n+1) = {} ⟶ (∀k>(n+1). level A r k = {}))
  show False
  proof−
    have ((level A r (n+1)) = {}) ∧ ¬(∀k>(n+1). level A r k = {})
      using hip by auto
    hence 1: (level A r (n+1)) = {} and 2: ∃k>(n+1). (level A r k) ≠ {}
```

```
      by auto
    obtain z where z1: z>(n+1) and z2: (level A r z) ≠ {}
      using 2 by auto
    have z>0 using ‹z>(n+1)› by auto
    hence (level A r ((z−1)+1)) ≠ {}
      using z2 by simp
    hence ∀k.(0≤k ∧ k≤(z−1)) ⟶ (level A r k)≠ {}
      using z2 ‹tree A r› set-nodes-at-level[of A r z−1]
      by auto
    hence (level A r (n+1)) ≠ {}
      using ‹z>(n+1)› by auto
    thus False using 1 by auto
  qed
qed

lemma characterization-nodes-tree-finite-height:
  assumes tree A r and ∀k. k>m ⟶ (level A r k) = {}
  shows A = (⋃n∈{0..m}. level A r n)
proof−
  have a: A = (⋃n. level A r n) using union-levels[of A r] by auto
  have (⋃n. level A r n) = (⋃n∈{0..m}. level A r n)
  proof(rule equalityI)
    show (⋃n. level A r n) ⊆ (⋃n∈{0..m}. level A r n)
    proof(rule subsetI)
      fix x
      assume hip: x∈(⋃n. level A r n)
      show x∈(⋃n∈{0..m}. level A r n)
      proof−
        have ∃n. x∈ level A r n
        using hip level-element[of A] by auto
        then obtain n where n: x∈ level A r n by auto
        have n∈{0..m}
        proof(rule ccontr)
          assume 1: n ∉ {0..m}
          show False
          proof−
            have n > m using 1 by auto
            thus False using assms(2) n by auto
          qed
        qed
        thus x∈(⋃n∈{0..m}. level A r n) using n by auto
      qed
    qed
  next
    show (⋃n∈{0..m}. level A r n) ⊆ (⋃n. level A r n) by auto
  qed
  thus A = (⋃n∈{0..m}. level A r n) using a by auto
qed
```

lemma *finite-tree-if-fin-branches-and-fin-height*:
  assumes *tree A r* **and** *finitely-branching A r*
  and $\exists n. (\forall k. k>n \longrightarrow (level\ A\ r\ k) = \{\})$
  shows *finite A*
proof−
  obtain *m* **where** *m*: $(\forall k. k>m \longrightarrow (level\ A\ r\ k) = \{\})$
    using *assms(3)* **by** *auto*
  hence *1*: $A =(\bigcup n\in\{0..m\}.\ level\ A\ r\ n)$
     using *assms(1) assms(3) characterization-nodes-tree-finite-height[of A r m]*
by *auto*
  have $\forall n.$ *finite* (*level A r n*)
    using *assms(1−2) finite-level* **by** *auto*
  hence $\forall n\in\{0..m\}.$ *finite* (*level A r n*) **by** *auto*
  hence *finite* $(\bigcup n\in\{0..m\}.\ level\ A\ r\ n)$ **by** *auto*
  thus *finite A* **using** *1* **by** *auto*
qed

lemma *all-levels-non-empty*:
  assumes *infinite-tree A r* **and** *finitely-branching A r*
  shows $\forall n.\ level\ A\ r\ n \neq \{\}$
proof(*rule ccontr*)
  assume *hip*: $\neg (\forall n.\ level\ A\ r\ n \neq \{\})$
  show *False*
  proof−
    have *tree A r* **using** ‹*infinite-tree A r*› **by** *auto*
    have $(\exists n.\ level\ A\ r\ n = \{\})$ **using** *hip* **by** *auto*
    then obtain *n* **where** *n*: $level\ A\ r\ n = \{\}$ **by** *auto*
    thus *False*
    proof(*cases n*)
      case *0*
      then show *False*
      proof−
        have $\exists a.$ *minimum A a r* **using** ‹*tree A r*› *tree[of A r]* **by** *auto*
        then obtain *a* **where** *a*: *minimum A a r* **by** *auto*
        have *strict-part-order A r*
        and $\forall x\in A.$ *finite* (*predecessors A x r*)
          using ‹*tree A r*› *tree[of A r]* **by** *auto*
        hence $level\ A\ r\ n = \{a\}$
          using *a* ‹$n=0$› *zero-level[of A r a]* **by** *auto*
        thus *False* **using** ‹$level\ A\ r\ n = \{\}$› **by** *auto*
      qed
      next
        case (*Suc nat*)
        fix *m*
        assume *hip*: $n = Suc\ m$ **show** *False*
        proof−
          have *1*: $level\ A\ r\ (Suc\ m) = \{\}$
            using *hip n* **by** *auto*
        have $(\forall k.\ k>(m+1) \longrightarrow (level\ A\ r\ k) = \{\})$

```
          using ‹tree A r› 1  emptyness-below-height[of A r m] by auto
        hence 1: (∃ n. ∀ k. k>n ⟶ (level A r k) = {}) by auto
        hence 2: finite A
       using ‹tree A r› 1 ‹finitely-branching  A r› finite-tree-if-fin-branches-and-fin-height[of
A r] by auto
        have 3:  ¬ finite A using ‹infinite-tree A r› by auto
        show False using 2 3 by auto
      qed
    qed
  qed
qed

lemma simple-cyclefree:
  assumes tree A r and (x,z)∈r and (y,z)∈r and x≠y
  shows (x,y)∈r ∨ (y,x)∈r
proof−
  have r ⊆ A × A using ‹tree A r› by(unfold tree-def, auto)
  hence x∈A and  y∈A and  z∈A using  ‹(x,z)∈r› and ‹(y,z)∈r› by auto
  hence 1: x ∈ predecessors A z r and 2: y ∈ predecessors A z r
    using assms by(unfold predecessors-def, auto)
  have (total-on (predecessors A  z r) r)
    using ‹tree A r› ‹z∈A› by(unfold tree-def, auto)
  thus ?thesis using 1 2 ‹x≠y›  total-on-def[of predecessors A z r r] by auto
qed

lemma inclusion-predecessors:
  assumes  r ⊆ A × A and strict-part-order A r and (x,y)∈r
  shows (predecessors A x r) ⊂ (predecessors A y r)
proof−
  have irreflexive-on A r and transitive-on A r
    using assms(2) by (unfold strict-part-order-def, auto)
  have 1: (predecessors A x r)⊆ (predecessors A y r)
  proof(rule subsetI)
    fix z
    assume z∈predecessors A x r
    hence z∈A and (z,x)∈r by(unfold predecessors-def, auto)
    have x∈A and y∈A  using ‹(x,y)∈r› ‹r ⊆ A × A› by auto
    hence (z,y)∈r
      using ‹z∈A› ‹y∈A› ‹x∈A› ‹(z,x)∈r› ‹(x,y)∈r› ‹transitive-on A r›
      by (unfold transitive-on-def, blast)
    thus z∈predecessors A y r
      using ‹z∈A› by(unfold predecessors-def, auto)
  qed
  have 2: x∈predecessors A y r
    using ‹r ⊆ A × A› ‹(x,y)∈r› by(unfold predecessors-def, auto)
  have 3:  x∉predecessors A x r
  proof(rule ccontr)
    assume ¬ x ∉ predecessors A x r
    hence x ∈ predecessors A x r by auto
```

hence *x*∈*A* ∧ (*x*,*x*)∈*r*
by(*unfold predecessors-def*, *auto*)
thus *False* **using** ‹*irreflexive-on A r*›
by (*unfold irreflexive-on-def*, *auto*)
qed
have (*predecessors A x r*) ≠ (*predecessors A y r*)
using *2 3* **by** *auto*
thus *?thesis* **using** *1* **by** *auto*
qed

lemma *different-height-finite-pred*:
assumes *r* ⊆ *A* × *A* **and** *strict-part-order A r* **and** (*x*,*y*)∈*r*
and *finite* (*predecessors A y r*)
shows *height A x r* < *height A y r*
proof−
have *card*(*predecessors A x r*) < *card*(*predecessors A y r*)
using *assms inclusion-predecessors*[*of r A x y*] *psubset-card-mono* **by** *auto*
thus *?thesis* **by**(*unfold height-def*, *auto*)
qed

lemma *different-levels-finite-pred*:
assumes *r* ⊆ *A* × *A* **and** *strict-part-order A r* **and** (*x*,*y*)∈*r*
and *x* ∈ (*level A r n*) **and** *y* ∈ (*level A r m*)
and *finite* (*predecessors A y r*)
shows *level A r n* ≠ *level A r m*
proof(*rule ccontr*)
assume ¬ *level A r n* ≠ *level A r m*
hence *level A r n* = *level A r m* **by** *auto*
hence *x* ∈ (*level A r m*) **using** ‹*x* ∈ (*level A r n*)› **by** *auto*
hence *1*: *height A x r*= *m* **by**(*unfold level-def*, *auto*)
have *height A y r*= *m* **using** ‹*y* ∈ (*level A r m*)› **by**(*unfold level-def*, *auto*)
hence *height A x r* = *height A y r* **using** *1* **by** *auto*
thus *False*
using *assms different-height-finite-pred*[*of r A x y*] **by** (*unfold level-def*, *auto*)
qed

lemma *less-level-pred-in-fin-pred*:
assumes *r* ⊆ *A* × *A* **and** *strict-part-order A r*
and *x* ∈ *predecessors A y r* **and** *y* ∈ (*level A r n*)
and *x* ∈ (*level A r m*)
and *finite* (*predecessors A y r*)
shows *m*<*n*
proof−
have (*x*,*y*)∈*r* **using** ‹(*x* ∈ *predecessors A y r*)›
by (*unfold predecessors-def*, *auto*)
thus *?thesis*
using *assms different-height-finite-pred*[*of r A x y*] **by**(*unfold level-def*, *auto*)
qed

```
lemma emptyness-inter-diff-levels-aux:
  assumes tree A r and x∈(predecessors A z r)
  and y∈(predecessors A z r)
  and  x≠y and x ∈ (level A r n) and y ∈ (level A r m)
  shows level A r n ∩ level A r m = {}
proof−
  have (x,y)∈r ∨ (y,x)∈r
    using assms simple-cyclefree[of A] by(unfold predecessors-def, auto)
  thus level A r n ∩ level A r m ={}
  proof(rule disjE)
    assume  (x, y) ∈ r
    have r⊆ A × A and 1: strict-part-order A r
      using  ‹tree A r› by(unfold tree-def,auto)
    hence x∈A and y∈A  and  2: x∈(predecessors A y r)
      using ‹(x, y) ∈ r›  by(unfold predecessors-def, auto)
    have 3: finite (predecessors A y r)
      using ‹y∈A›  ‹tree A r› by(unfold tree-def, auto)
    hence  n<m
      using assms ‹r⊆ A × A› 1 2 3 less-level-pred-in-fin-pred[of r A x y m n]
      by auto
    hence ∃k>0. m=n+k by arith
    then obtain k where k: k>0 and m: m=n+k by auto
    thus ?thesis using uniqueness-level-aux[OF k, of A ]
      by auto
  next
    assume  (y, x) ∈ r
    have r⊆ A × A and 1: strict-part-order A r
      using  ‹tree A r› by(unfold tree-def,auto)
    hence x∈A and y∈A and 2: y∈(predecessors A x r)
      using ‹(y, x) ∈ r›
      by(unfold predecessors-def, auto)
    have 3: finite (predecessors A x r)
      using ‹x∈A› ‹tree A r›
      by(unfold tree-def, auto)
    hence  m<n
      using assms ‹r⊆ A × A› 1 2 3 less-level-pred-in-fin-pred[of r A y x n m]
      by auto
    hence ∃k>0. n=m+k by arith
    then obtain k where k: k>0 and m: n=m+k by auto
    thus ?thesis using  uniqueness-level-aux[OF k, of A] by auto
  qed
qed

lemma emptyness-inter-diff-levels:
  assumes tree A r and (x,z)∈ r and (y,z)∈ r
  and  x≠y and  x ∈ (level A r n) and y ∈ (level A r m)
shows level A r n ∩ level A r m = {}
proof−
  have r ⊆ A × A  using  ‹tree A r› tree by auto
```

```
  hence x∈A and  y∈A  using  ‹r ⊆ A × A›  ‹(x,z) ∈ r›  ‹(y,z)∈r› by auto
  hence x∈(predecessors A z r) and y∈(predecessors A z r)
    using  ‹(x,z)∈ r› and ‹(y,z)∈ r› by(unfold predecessors-def, auto)
  thus ?thesis
    using assms  emptyness-inter-diff-levels-aux[of A r] by blast
qed

primrec disjunction-nodes :: 'a list  ⇒ 'a formula  where
 disjunction-nodes [] = FF
| disjunction-nodes (v#D) = (atom v) ∨. (disjunction-nodes D)

lemma truth-value-disjunction-nodes:
  assumes v∈ set l and t-v-evaluation I (atom v) = Ttrue
  shows t-v-evaluation I (disjunction-nodes l) = Ttrue
proof−
  have v∈ set l ⟹  t-v-evaluation I (atom v) = Ttrue ⟹
  t-v-evaluation I (disjunction-nodes l) = Ttrue
  proof(induct l)
    case Nil
    then show ?case by auto
  next
    case (Cons a l)
    then show  t-v-evaluation I (disjunction-nodes (a # l)) = Ttrue
    proof−
      have v = a ∨ v≠a by auto
      thus  t-v-evaluation I (disjunction-nodes (a # l)) = Ttrue
      proof(rule disjE)
        assume v = a
        hence 1: disjunction-nodes (a#l) = (atom v) ∨. (disjunction-nodes l)
          by auto
        have t-v-evaluation I ((atom v) ∨. (disjunction-nodes l)) = Ttrue
        using Cons(3)  by(unfold t-v-evaluation-def,unfold v-disjunction-def, auto)
        thus ?thesis using 1  by auto
      next
        assume v ≠ a
        hence v∈ set l using Cons(2) by auto
        hence t-v-evaluation I (disjunction-nodes l) = Ttrue
          using Cons(1) Cons(3) by auto
        thus ?thesis
          by(unfold t-v-evaluation-def,unfold v-disjunction-def, auto)
      qed
    qed
  qed
  thus ?thesis using assms by auto
qed

lemma set-set-to-list1:
  assumes tree A r and  finitely-branching A r
  shows set (set-to-list (level A r n)) = (level A r n)
```

using *assms finite-level*[*of A r n*] *set-set-to-list* **by** *auto*

lemma *truth-value-disjunction-formulas*:
assumes *tree A r* **and** *finitely-branching A r*
and $v \in (level\ A\ r\ n) \wedge t\text{-}v\text{-}evaluation\ I\ (atom\ v) = Ttrue$
and $F = disjunction\text{-}nodes(set\text{-}to\text{-}list\ (level\ A\ r\ n))$
shows $t\text{-}v\text{-}evaluation\ I\ F = Ttrue$
proof−
have $set\ (set\text{-}to\text{-}list\ (level\ A\ r\ n)) = (level\ A\ r\ n)$
using *set-set-to-list1 assms*(*1*−*2*) **by** *auto*
hence $v \in set\ (set\text{-}to\text{-}list\ (level\ A\ r\ n))$
using *assms*(*3*) **by** *auto*
thus $t\text{-}v\text{-}evaluation\ I\ F = Ttrue$
using *assms*(*3*−*4*) *truth-value-disjunction-nodes* **by** *auto*
qed

definition $\mathcal{F}$ :: *'a set* ⇒ *'a rel* ⇒ (*'a formula*) *set* **where**
$\mathcal{F}\ A\ r \equiv (\bigcup n.\ \{disjunction\text{-}nodes(set\text{-}to\text{-}list\ (level\ A\ r\ n))\})$

definition $\mathcal{G}$ :: *'a set* ⇒ *'a rel* ⇒ (*'a formula*) *set* **where**
$\mathcal{G}\ A\ r \equiv \{(atom\ u) \rightarrow.\ (atom\ v)\ |u\ v.\ u \in A \wedge v \in A \wedge (v,u) \in r\}$

definition $\mathcal{H}n$ :: *'a set* ⇒ *'a rel* ⇒ *nat* ⇒ (*'a formula*) *set* **where**
$\mathcal{H}n\ A\ r\ n \equiv \{\neg.((atom\ u) \wedge.\ (atom\ v))$
$|u\ v\ .\ u \in (level\ A\ r\ n) \wedge v \in (level\ A\ r\ n) \wedge u \neq v\ \}$
definition $\mathcal{H}$ :: *'a set* ⇒ *'a rel* ⇒ (*'a formula*) *set* **where**
$\mathcal{H}\ A\ r \equiv \bigcup n.\ \mathcal{H}n\ A\ r\ n$

definition $\mathcal{T}$ :: *'a set* ⇒ *'a rel* ⇒ (*'a formula*) *set* **where**
$\mathcal{T}\ A\ r \equiv (\mathcal{F}\ A\ r) \cup (\mathcal{G}\ A\ r) \cup (\mathcal{H}\ A\ r)$

primrec *nodes-formula* :: *'v formula* ⇒ *'v set* **where**
nodes-formula FF = {}
| *nodes-formula TT* = {}
| *nodes-formula* (*atom P*) = {*P*}
| *nodes-formula* (¬. *F*) = *nodes-formula F*
| *nodes-formula* (*F* ∧. *G*) = *nodes-formula F* ∪ *nodes-formula G*
| *nodes-formula* (*F* ∨. *G*) = *nodes-formula F* ∪ *nodes-formula G*
| *nodes-formula* (*F* →.*G*) = *nodes-formula F* ∪ *nodes-formula G*

definition *nodes-set-formulas* :: *'v formula set* ⇒ *'v set* **where**
$nodes\text{-}set\text{-}formulas\ S = (\bigcup F \in S.\ nodes\text{-}formula\ F)$

definition *maximum-height*:: *'v set* ⇒*'v rel* ⇒ *'v formula set* ⇒ *nat* **where**
$maximum\text{-}height\ A\ r\ S = Max\ (\bigcup x \in nodes\text{-}set\text{-}formulas\ S.\ \{height\ A\ x\ r\})$

lemma *node-formula*:
assumes $v \in set\ l$
shows $v \in nodes\text{-}formula\ (disjunction\text{-}nodes\ l)$

```
proof−
  have v ∈ set l ⟹ v ∈ nodes-formula (disjunction-nodes l)
  proof(induct l)
    case Nil
    then show ?case by auto
  next
    case (Cons a l)
    show v ∈ nodes-formula (disjunction-nodes (a # l))
   proof−
     have v = a ∨ v≠a by auto
     thus v ∈ nodes-formula (disjunction-nodes (a # l))
     proof(rule disjE)
       assume v = a
       hence 1: disjunction-nodes (a#l) = (atom v) ∨. (disjunction-nodes l)
         by auto
       have v ∈ nodes-formula ((atom v) ∨. (disjunction-nodes l)) by auto
       thus ?thesis using 1  by auto
     next
       assume v ≠ a
       hence v∈ set l using Cons(2) by auto
       hence v ∈ nodes-formula (disjunction-nodes l)
         using Cons(1) Cons(2) by auto
       thus ?thesis by auto
     qed
   qed
 qed
  thus ?thesis using assms by auto
qed

lemma node-disjunction-formulas:
  assumes  tree A r and  finitely-branching A r and v∈(level A r n)
  and  F = disjunction-nodes(set-to-list (level A r n))
  shows  v ∈ nodes-formula F
proof−
  have set (set-to-list (level A r n)) = (level A r n)
    using set-set-to-list1 assms(1−2) by auto
  hence v∈ set (set-to-list (level A r n))
    using assms(3) by auto
  thus v ∈ nodes-formula F
    using assms(3−4)  node-formula  by auto
qed

fun node-sig-level-max:: 'v set ⇒ 'v rel ⇒ 'v formula set  ⇒ 'v
  where node-sig-level-max A r S =
  (SOME u. u ∈ (level A r ((maximum-height A r S)+1)))

lemma node-level-maximum:
  assumes infinite-tree A r and  finitely-branching A r
  shows (node-sig-level-max A r S) ∈  (level A r ((maximum-height A r S)+1))
```

proof –
  have ∃ *u*. *u* ∈ (*level A r* ((*maximum-height A r S*)+*1*))
    using *assms all-levels-non-empty*[*of A r*] **by** (*unfold level-def*, *auto*)
  then obtain *u* **where** *u*: *u* ∈ (*level A r* ((*maximum-height A r S*)+*1*)) **by** *auto*
  hence (*SOME u*. *u* ∈ (*level A r* ((*maximum-height A r S*)+*1*))) ∈ (*level A r* ((*maximum-height A r S*)+*1*))
    using *someI* **by** *auto*
  thus *?thesis* **by** *auto*
qed

fun *path-interpretation* :: *'v set* ⇒*'v rel* ⇒ *'v* ⇒ (*'v* ⇒ *v-truth*) **where**
path-interpretation A r u = (λ*v*. (*if* (*v*,*u*)∈*r* *then Ttrue else Ffalse*))

lemma *finiteness-nodes-formula*:
finite (*nodes-formula F*) **by**(*induct F*, *auto*)

lemma *finiteness-set-nodes*:
  assumes *finite S*
  shows *finite* (*nodes-set-formulas S*)
  using *assms finiteness-nodes-formula*
  by (*unfold nodes-set-formulas-def*, *auto*)

lemma *maximum1*:
  assumes *finite S* **and** *u* ∈ *nodes-set-formulas S*
  shows (*height A u r*) ≤ (*maximum-height A r S*)
proof –
  have (*height A u r*) ∈ (⋃*x*∈*nodes-set-formulas S*. {*height A x r*})
    using *assms*(*2*) **by** *auto*
  thus (*height A u r*) ≤ (*maximum-height A r S*)
    using ‹*finite S*› *finiteness-set-nodes*[*of S*]
    by(*unfold maximum-height-def*, *auto*)
qed

lemma *value-path-interpretation*:
  assumes *t-v-evaluation* (*path-interpretation A r v*) (*atom u*) = *Ttrue*
  shows (*u*,*v*)∈*r*
proof(*rule ccontr*)
  assume (*u*, *v*) ∉ *r*
  hence *t-v-evaluation* (*path-interpretation A r v*) (*atom u*) = *Ffalse*
    by(*unfold t-v-evaluation-def*, *auto*)
  thus *False* **using** *assms* **by** *auto*
qed

lemma *satisfiable-path*:
  assumes *infinite-tree A r*
  and *finitely-branching A r* **and** *S* ⊆ (*𝒯 A r*)
  and *finite S*
shows *satisfiable S*
proof –

```
let ?m = (maximum-height A r S)+1
let ?level = level A r ?m
let ?u = node-sig-level-max A r S
have 1: tree A r using ‹infinite-tree A r› by auto
have  r ⊆ A × A and strict-part-order A r
  using  ‹tree A r› tree by auto
have transitive-on A r
  using ‹strict-part-order A r›
  by(unfold strict-part-order-def, auto)
have ∃ u. u ∈?level
  using assms(1−2) node-level-maximum by auto
then obtain u where u: u ∈ ?level  by auto
hence levelu:  ?u ∈ ?level
  using someI by auto
hence ?u∈A by(unfold level-def, auto)
have (path-interpretation A r ?u) model S
proof(unfold model-def)
  show ∀ F∈S. t-v-evaluation (path-interpretation A r ?u) F = Ttrue
  proof
    fix F assume F ∈ S
    show  t-v-evaluation (path-interpretation A r ?u) F  = Ttrue
    proof−
      have F ∈ (𝓕 A r) ∪ (𝓖 A r) ∪ (𝓗 A r)
      using ‹S ⊆  𝓣 A r› ‹F ∈ S› assms(2)  by(unfold 𝓣-def,auto)
      hence  F ∈ (𝓕 A r) ∨ F ∈ (𝓖 A r) ∨ F ∈ (𝓗 A r) by auto
      thus ?thesis
      proof(rule disjE)
        assume F ∈ (𝓕 A r)
        hence ∃ n. F = disjunction-nodes(set-to-list (level A r n))
          by(unfold 𝓕-def,auto)
        then obtain n
          where n: F = disjunction-nodes(set-to-list (level A r n))
          by auto
        have ∃ v. v∈(level A r n)
          using  assms(1−2) all-levels-non-empty[of A r] by auto
        then obtain v where v: v ∈ (level A r n) by auto
        hence  v ∈ nodes-formula F
          using n node-disjunction-formulas[OF 1 assms(2) v, of F ]
          by auto
        hence a: v ∈ nodes-set-formulas S
          using ‹F ∈ S›  by(unfold nodes-set-formulas-def, blast)
        hence b: (height A v r) ≤ (maximum-height A r S)
          using ‹finite S›  maximum1[of S v] by auto
        have (height A v r) = n
          using v by(unfold level-def, auto)
        hence  n < ?m
          using ‹finite S› a   maximum1[of S v A r]
          by(unfold maximum-height-def, auto)
        hence (∃ y. (y,?u)∈r ∧ y ∈ (level A r n))
```

```
    using levelu ‹tree A r› path-to-node[of A r]
    by auto
  then obtain y where y1: (y,?u)∈r and y2: y ∈ (level A r n)
    by auto
  hence t-v-evaluation (path-interpretation A r ?u) (atom y) = Ttrue
    by auto
  thus t-v-evaluation (path-interpretation A r ?u) F = Ttrue
    using 1 assms(2) y2 n  truth-value-disjunction-formulas[of A r y]
    by auto
next
  assume  F ∈ 𝒢 A r ∨ F ∈ ℋ A r
  thus t-v-evaluation (path-interpretation A r ?u) F = Ttrue
  proof(rule disjE)
    assume  F ∈ 𝒢 A r
    hence ∃u. ∃v. u∈A ∧ v∈A  ∧ (v,u)∈ r  ∧
          (F = (atom u) →. (atom v))
      by (unfold  𝒢-def, auto)
    then obtain u v where u∈A and v∈A and (v,u)∈ r
    and F: (F = (atom u) →. (atom v)) by auto
    show t-v-evaluation (path-interpretation A r ?u) F = Ttrue
    proof(rule ccontr)
      assume ¬(t-v-evaluation (path-interpretation A r ?u) F = Ttrue)
      hence t-v-evaluation (path-interpretation A r ?u) F = Ffalse
        using Bivaluation by auto
     hence t-v-evaluation (path-interpretation A r ?u) (atom u) =  Ttrue ∧
      t-v-evaluation (path-interpretation A r ?u) (atom v) =  Ffalse
        using F  eval-false-implication by blast
     hence 1: t-v-evaluation (path-interpretation A r ?u) (atom u) =  Ttrue
     and   2: t-v-evaluation (path-interpretation A r ?u) (atom v) =  Ffalse
        by auto
      have (u,?u)∈r using 1 value-path-interpretation by auto
      hence (v,?u)∈ r
        using  ‹u∈A› ‹v∈A› ‹?u∈A› ‹(v,u)∈ r› ‹transitive-on A r›
        by(unfold transitive-on-def, blast)
      hence t-v-evaluation (path-interpretation A r ?u) (atom v) =  Ttrue
        by auto
      thus False using 2 by auto
    qed
  next
    assume  F ∈ ℋ A r
    hence ∃n. F ∈ ℋn A r n by(unfold  ℋ-def, auto)
    then obtain n where  F ∈ ℋn A r n by auto
    hence
    ∃u. ∃v. F = ¬.((atom u) ∧. (atom v)) ∧ u∈(level A r n) ∧
     v∈(level A r n) ∧ u≠v
      by(unfold ℋn-def, auto)
    then obtain u v where F: F = ¬.((atom u) ∧. (atom v))
    and u∈(level A r n) and v∈(level A r n) and u≠v
      by auto
```

```
        show t-v-evaluation (path-interpretation A r ?u) F = Ttrue
        proof(rule ccontr)
          assume t-v-evaluation (path-interpretation A r ?u) F ≠ Ttrue
          hence t-v-evaluation (path-interpretation A r ?u) F = Ffalse
            using Bivaluation by auto
          hence
          t-v-evaluation (path-interpretation A r ?u)((atom u) ∧.
           (atom v)) = Ttrue
            using F NegationValues1 by blast
          hence t-v-evaluation (path-interpretation A r ?u)(atom u) = Ttrue ∧
          t-v-evaluation (path-interpretation A r ?u)(atom v) = Ttrue
            using ConjunctionValues by blast
          hence (u,?u)∈r and (v,?u)∈r
            using value-path-interpretation by auto
          hence a: (level A r n) ∩ (level A r n) = {}
            using ‹tree A r› ‹u∈(level A r n)› ‹v∈(level A r n)› ‹u≠v›
            emptyness-inter-diff-levels[of A r]
            by blast
          have (level A r n) ≠ {}
            using ‹v∈(level A r n)› by auto
          thus False using a by auto
        qed
      qed
    qed
  qed
 qed
qed
thus satisfiable S by(unfold satisfiable-def, auto)
qed

definition 𝓑:: 'a set ⇒ ('a ⇒ v-truth) ⇒ 'a set where
𝓑 A I ≡ {u|u. u∈A ∧ t-v-evaluation I (atom u) = Ttrue}

lemma value-disjunction-list1:
  assumes t-v-evaluation I (disjunction-nodes (a # l)) = Ttrue
  shows t-v-evaluation I (atom a) = Ttrue ∨ t-v-evaluation I (disjunction-nodes
l) = Ttrue
proof−
  have disjunction-nodes (a # l) = (atom a) ∨. (disjunction-nodes l)
    by auto
  hence t-v-evaluation I ((atom a) ∨. (disjunction-nodes l)) = Ttrue
    using assms by auto
  thus ?thesis using DisjunctionValues by blast
qed

lemma value-disjunction-list:
  assumes t-v-evaluation I (disjunction-nodes l) = Ttrue
  shows ∃x. x ∈ set l ∧ t-v-evaluation I (atom x) = Ttrue
proof−
```

have *t-v-evaluation I* (*disjunction-nodes l*) = *Ttrue* ⟹
∃ *x*. *x* ∈ *set l* ∧ *t-v-evaluation I* (*atom x*) = *Ttrue*
proof(*induct l*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a l*)
show ∃ *x*. *x* ∈ *set* (*a* # *l*) ∧ *t-v-evaluation I* (*atom x*) = *Ttrue*
proof−
have *t-v-evaluation I* (*atom a*) = *Ttrue* ∨ *t-v-evaluation I* (*disjunction-nodes l*)=*Ttrue*
using *Cons*(*2*) *value-disjunction-list1*[*of I*] **by** *auto*
thus *?thesis*
proof(*rule disjE*)
assume *t-v-evaluation I* (*atom a*) = *Ttrue*
thus *?thesis* **by** *auto*
next
assume *t-v-evaluation I* (*disjunction-nodes l*) = *Ttrue*
thus *?thesis*
using *Cons* **by** *auto*
qed
qed
qed
thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *intersection-branch-set-nodes-at-level*:
assumes *infinite-tree A r* **and** *finitely-branching A r*
and *I*: ∀ *F* ∈ (ℱ *A r*). *t-v-evaluation I F* = *Ttrue*
shows ∀ *n*. ∃ *x*. *x* ∈ *level A r n* ∧ *x* ∈ (ℬ *A I*) **using** *all-levels-non-empty*
proof−
fix *n*
have ∀ *n*. *t-v-evaluation I* (*disjunction-nodes*(*set-to-list* (*level A r n*))) = *Ttrue*
using *I* **by** (*unfold* ℱ*-def*, *auto*)
hence *1*:
∀ *n*. ∃ *x*. *x* ∈ *set* (*set-to-list* (*level A r n*)) ∧ *t-v-evaluation I* (*atom x*) = *Ttrue*
using *value-disjunction-list* **by** *auto*
have *tree A r*
using ‹*infinite-tree A r*›**by** *auto*
hence ∀ *n*. *set* (*set-to-list* (*level A r n*)) = *level A r n*
using *assms*(*1*−*2*) *set-set-to-list1* **by** *auto*
hence ∀ *n*. ∃ *x*. *x* ∈ *level A r n* ∧ *t-v-evaluation I* (*atom x*) = *Ttrue*
using *1* **by** *auto*
hence ∀ *n*. ∃ *x*. *x* ∈ *level A r n* ∧ *x*∈*A* ∧ *t-v-evaluation I* (*atom x*) = *Ttrue*
by(*unfold level-def*, *auto*)
thus *?thesis* **using** ℬ*-def*[*of A I*] **by** *auto*
qed

lemma *intersection-branch-emptyness-below-height*:

assumes *I*: ∀ *F* ∈ (𝓗 *A r*). *t-v-evaluation I F* = *Ttrue*
and *x*∈(𝓑 *A I*) **and** *y*∈(𝓑 *A I*) **and** *x* ≠ *y* **and** *n*: *x* ∈ *level A r n*
and *m*: *y* ∈ *level A r m*
shows *n* ≠ *m*
proof(*rule ccontr*)
assume ¬ *n* ≠ *m*
hence *n*=*m* **by** *auto*
have *x*∈*A* **and** *y*∈*A* **and** *v1*: *t-v-evaluation I* (*atom x*) = *Ttrue*
and *v2*: *t-v-evaluation I* (*atom y*) = *Ttrue*
using ‹*x*∈(𝓑 *A I*)› ‹*y*∈(𝓑 *A I*)› **by**(*unfold* 𝓑*-def*, *auto*)
have ¬.((*atom x*) ∧. (*atom y*)) ∈ (𝓗*n A r n*)
using ‹*x*∈*A*› ‹*y*∈*A*› ‹*x* ≠ *y*› *n m* ‹*n*=*m*›
by(*unfold* 𝓗*n-def*, *auto*)
hence ¬.((*atom x*) ∧. (*atom y*)) ∈ (𝓗 *A r*)
by(*unfold* 𝓗*-def*, *auto*)
hence *t-v-evaluation I* (¬.((*atom x*) ∧. (*atom y*))) = *Ttrue*
using *I* **by** *auto*
moreover
have *t-v-evaluation I* ((*atom x*) ∧. (*atom y*)) = *Ttrue*
using *v1 v2 v-conjunction-def* **by** *auto*
hence *t-v-evaluation I* (¬.((*atom x*) ∧. (*atom y*))) = *Ffalse*
using *v-negation-def* **by** *auto*
ultimately
show *False* **by** *auto*
qed

lemma *intersection-branch-level*:
assumes *infinite-tree A r* **and** *finitely-branching A r*
and *I*: ∀ *F* ∈ (𝓕 *A r*) ∪ (𝓗 *A r*). *t-v-evaluation I F* = *Ttrue*
shows ∀ *n*. ∃ *u*. (𝓑 *A I*) ∩ *level A r n* = {*u*}
proof
fix *n*
show ∃ *u*. (𝓑 *A I*) ∩ *level A r n* = {*u*}
proof−
have ∃ *u*. *u* ∈ *level A r n* ∧ *u* ∈ (𝓑 *A I*)
using *assms intersection-branch-set-nodes-at-level*[*of A r I*] **by** *auto*
then obtain *u* **where** *u*: *u* ∈ *level A r n* ∧ *u*∈(𝓑 *A I*) **by** *auto*
hence *1*: {*u*} ⊆ (𝓑 *A I*) ∩ *level A r n* **by** *blast*
have *2*: (𝓑 *A I*) ∩ *level A r n* ⊆ {*u*}
proof(*rule subsetI*)
fix *x*
assume *x*∈(𝓑 *A I*) ∩ *level A r n*
hence *2*: *x*∈(𝓑 *A I*) ∧ *x*∈ *level A r n* **by** *auto*
have *u* = *x*
proof(*rule ccontr*)
assume *u* ≠ *x*
hence *n*≠*n*
using *u 2 I intersection-branch-emptyness-below-height*[*of A r*] **by** *blast*
thus *False* **by** *auto*

```
      qed
      thus x∈{u} by auto
    qed
    have (ℬ A I) ∩ level A r n = {u}
      using 1 2 by auto
    thus ∃ u.(ℬ A I) ∩  level A r n = {u}  by auto
  qed
qed

lemma predecessor-in-branch:
  assumes I:  ∀ F ∈ (𝒢 A r). t-v-evaluation I F = Ttrue
  and y∈(ℬ A I)  and  (x,y)∈ r and x∈A and y∈A
shows x∈(ℬ A I)
proof−
  have (atom y) →. (atom x)∈ 𝒢 A r
    using ‹x∈A›  ‹y∈A›  ‹(x, y)∈r› by (unfold 𝒢-def, auto)
  hence t-v-evaluation I ((atom y) →. (atom x)) = Ttrue
    using I by auto
  moreover
  have t-v-evaluation I (atom y) = Ttrue
    using  ‹y∈(ℬ A I)› by(unfold ℬ-def, auto)
  ultimately
  have t-v-evaluation I (atom x) = Ttrue
    using v-implication-def by  auto
  thus  x∈(ℬ A I) using  ‹x∈A›  by(unfold ℬ-def, auto)
qed

lemma is-path:
  assumes  infinite-tree A r and finitely-branching A r
  and I: ∀ F ∈ (𝒯 A r). t-v-evaluation I F = Ttrue
shows path (ℬ A I) A r
proof(unfold path-def)
  let ?B = (ℬ A I)
  have tree A r
  using  ‹infinite-tree A r› by auto
  have ∀ F ∈ (ℱ A r) ∪ (𝒢 A r) ∪ (ℋ A r). t-v-evaluation I F = Ttrue
    using I by(unfold 𝒯-def)
  hence I1:  ∀ F ∈ (ℱ A r). t-v-evaluation I F = Ttrue
  and   I2:  ∀ F ∈ (𝒢 A r). t-v-evaluation I F = Ttrue
  and   I3:  ∀ F ∈ (ℋ A r). t-v-evaluation I F = Ttrue
    by auto
  have 0: sub-linear-order ?B A r
  proof(unfold sub-linear-order-def)
    have 1: ?B ⊆ A  by(unfold ℬ-def, auto)
    have 2: strict-part-order A r
      using ‹tree A r› tree[of A r] by auto
    have total-on ?B r
    proof(unfold total-on-def)
      show ∀ x∈?B. ∀ y∈?B. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
```

```
proof
  fix x
  assume x∈?B
  show ∀ y∈?B. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
  proof
    fix y
    assume y∈?B
    show x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
    proof(rule impI)
      assume x ≠ y
      have x∈A and y∈A and v1: t-v-evaluation I (atom x) = Ttrue
      and v2: t-v-evaluation I (atom y) = Ttrue
        using ‹x∈?B› ‹y∈?B›  by(unfold ℬ-def, auto)
      have (∃ n. x ∈ level A r n) and (∃ m. y ∈ level A r m)
        using ‹x∈A› and ‹y∈A› level-element[of A r]
        by auto
      then obtain n m
      where n: x ∈ level A r n and m: y ∈ level A r m
        by auto
      have n≠m
        using I3 ‹x∈?B› ‹y∈?B› ‹x ≠ y› n m
              intersection-branch-emptyness-below-height[of A r]
        by auto
      hence n<m ∨ m<n by auto
      thus (x, y) ∈ r ∨ (y, x) ∈ r
      proof(rule disjE)
        assume  n < m
        have (x, y) ∈ r
        proof(rule ccontr)
          assume (x, y) ∉ r
          have ∃ z. (z, y)∈r ∧ z ∈ level A r n
            using ‹tree A r› ‹y ∈ level A r m› ‹n < m›
                  path-to-node[of A r y m−1]
            by auto
          then obtain z where z1: (z, y)∈r and z2: z ∈ level A r n
            by auto
          have z∈A using  ‹tree A r› tree z1 by auto
          hence z∈(ℬ A I)
           using I2 ‹y∈A› ‹y∈?B› ‹(z, y)∈r› predecessor-in-branch[of A r I y
z]
            by auto
          have x≠z using ‹(x, y) ∉ r› ‹(z, y)∈r› by auto
          hence n≠n
        using I3 ‹x∈?B› ‹z∈?B› n z2  intersection-branch-emptyness-below-height[of
A r]
            by blast
          thus False by auto
        qed
        thus (x, y) ∈ r ∨ (y, x) ∈ r by auto
```

```
          next
            assume m < n
            have (y, x) ∈ r
            proof(rule ccontr)
              assume (y, x) ∉ r
              have ∃z. (z, x)∈r ∧ z ∈ level A r m
                using ‹tree A r› ‹x ∈ level A r n› ‹m < n›
                      path-to-node[of A r x n−1]
                by auto
                then obtain z where z1: (z, x)∈r and z2: z ∈ level A r m
                by auto
              have z∈A using ‹tree A r› tree z1 by auto
              hence z∈(ℬ A I)
               using I2 ‹x∈A› ‹x∈?B› ‹(z, x)∈r› predecessor-in-branch[of A r I x
z]
                by auto
              have y≠z using ‹(y, x) ∉ r› ‹(z, x)∈r› by auto
              hence m≠m
           using I3 ‹y∈?B› ‹z∈?B› m z2 intersection-branch-emptyness-below-height[of
A r ]
                by blast
              thus False by auto
            qed
            thus (x, y) ∈ r ∨ (y, x) ∈ r by auto
          qed
        qed
      qed
    qed
  qed
  thus 3: ?B ⊆ A ∧ strict-part-order A r ∧ total-on ?B r
    using 1 2 by auto
qed
have 4: (∀C. ?B ⊆ C ∧ sub-linear-order C A r ⟶ ?B = C)
proof
  fix C
  show ?B ⊆ C ∧ sub-linear-order C A r ⟶ ?B = C
  proof(rule impI)
    assume ?B ⊆ C ∧ sub-linear-order C A r
    hence ?B ⊆ C and sub-linear-order C A r by auto
    have C ⊆ ?B
    proof(rule subsetI)
        fix x
        assume x∈ C
      have C ⊆ A
        using ‹sub-linear-order C A r›
       by(unfold sub-linear-order-def, auto)
      hence x∈A using ‹x∈C› by auto
      have ∃n. x∈level A r n
        using ‹x∈A› level-element[of A] by auto
```

```
        then obtain n where n: x∈level A r n by auto
        have ∃ u. (ℬ A I) ∩ level A r n = {u}
          using assms(1,2) I1 I3 intersection-branch-level[of A r]
          by blast
        then obtain u where i: (ℬ A I) ∩ level A r n = {u}
          by auto
        hence u∈A and u: u ∈ level A r n
         by(unfold level-def, auto)
        have x=u
        proof(rule ccontr)
          assume hip: x≠u
          have u∈(ℬ A I) using i by auto
          hence u∈C using ‹?B ⊆ C› by auto
          have total-on C r
            using ‹sub-linear-order C A r› sub-linear-order-def[of C A r]
            by blast
          hence (x,u)∈r ∨ (u,x)∈r
            using hip ‹x∈C› ‹u∈C› ‹sub-linear-order C A r›
            by(unfold total-on-def,auto)
          thus False
          proof(rule disjE)
            assume (x,u)∈r
            have r ⊆ A × A and strict-part-order A r
            and finite (predecessors A u r)
              using ‹u∈A› ‹tree A r› tree[of A r] by auto
            hence  (level A r n) ≠ (level A r n)
              using ‹(x,u)∈r› ‹x ∈ level A r n› ‹u ∈ level A r n›
                    different-levels-finite-pred[of r A ] by blast
            thus False by auto
          next
            assume (u,x)∈r
            have r ⊆ A × A and strict-part-order A r
            and finite (predecessors A x r)
              using ‹x∈A› ‹tree A r› tree[of A r]  by auto
            hence (level A r n) ≠ (level A r n)
              using ‹(u,x)∈r› ‹u ∈ level A r n› ‹x ∈ level A r n›
                    different-levels-finite-pred[of r A ] by blast
            thus False by auto
          qed
        qed
        thus x ∈ ?B using i by auto
      qed
      thus  ?B = C  using ‹?B ⊆ C› by blast
    qed
  qed
  thus sub-linear-order (ℬ A I) A r ∧
        (∀ C. ℬ A I ⊆ C ∧ sub-linear-order C A r ⟶ ℬ A I = C)
    using ‹sub-linear-order (ℬ A I) A r› by auto
qed
```

lemma *surjective-infinite*:
  assumes $\exists f{::}\ 'a \Rightarrow nat.\ \forall n.\ \exists x{\in}A.\ n = f(x)$
  shows *infinite A*
proof(*rule ccontr*)
  assume $\neg$ *infinite A*
  hence *finite A* **by** *auto*
  hence $\exists n.\ \exists g.\ A = g\ `\ \{i{::}nat.\ i < n\}$
    using *finite-imp-nat-seg-image-inj-on*[*of A*] **by** *auto*
  then obtain $n\ g$ **where** g: $A = g\ `\ \{i{::}nat.\ i < n\}$ **by** *auto*
  obtain f **where** $(\forall n.\ \exists x{\in}A.\ n = (f{::}\ 'a \Rightarrow\ nat)(x))$
    using *assms* **by** *auto*
  hence $\forall m.\ \exists k{\in}\{i{::}nat.\ i < n\}.\ m = (f \circ g)(k)$
    using g **by** *auto*
  hence $(UNIV :: nat\ set) = (f \circ g)\ `\ \{i{::}nat.\ i < n\}$
    by *blast*
  hence *finite* $(UNIV :: nat\ set)$
    using *nat-seg-image-imp-finite* **by** *blast*
  thus *False* **by** *auto*
qed

lemma *family-intersection-infinita*:
  fixes $P ::\ nat \Rightarrow\ 'a\ set$
  assumes $\forall n.\ \forall m.\ n \neq m \longrightarrow P\ n \cap P\ m = \{\}$
  and $\forall n.\ (A \cap (P\ n)) \neq \{\}$
  shows *infinite* $(\bigcup n.\ (A \cap (P\ n)))$
proof−
  let $?f = \lambda x.\ SOME\ n.\ x{\in}(A \cap (P\ n))$
  have $\forall n.\ \exists x{\in}(\bigcup n.\ (A \cap (P\ n))).\ n = ?f(x)$
  proof
    fix n
    obtain a **where** a: $a \in (A \cap (P\ n))$ **using** *assms*(*2*) **by** *auto*
    {fix m
    have $a \in (A \cap (P\ m)) \longrightarrow m{=}n$
    proof(*rule impI*)
      assume *hip*: $a \in A \cap P\ m$ **show** $m = n$
      proof(*rule ccontr*)
        assume $m \neq n$
        hence $P\ m \cap P\ n = \{\}$ **using** *assms*(*1*) **by** *auto*
        thus *False* **using** *a hip* **by** *auto*
      qed
    qed}
    hence $\bigwedge m.\ a \in A \cap P\ m \Longrightarrow m = n$ **by** *auto*
    hence *1*: $?f(a) = n$ **using** *a some-equality* **by** *auto*
    have $a{\in}(\bigcup n.\ (A \cap (P\ n)))$ **using** *a* **by** *auto*
    thus $\exists x{\in}\bigcup n.\ A \cap P\ n.\ n = (SOME\ n.\ x \in A \cap P\ n)$ **using** *1* **by** *auto*
  qed
  hence $\exists f{::}\ 'a \Rightarrow\ nat.\ \forall n.\ \exists x{\in}((\bigcup n.\ (A \cap (P\ n)))).\ n = f(x)$
    using *exI* **by** *auto*

thus *?thesis* **using** *surjective-infinite* **by** *auto*
qed

lemma *infinite-path*:
  assumes *infinite-tree A r* **and** *finitely-branching A r*
  and *I*: $\forall F \in (\mathcal{F}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
shows *infinite* ($\mathcal{B}$ *A I*)
proof−
  have *a*: $\forall n. \forall m.\ n \neq m \longrightarrow$ *level A r n* $\cap$ *level A r m* = {}
    using *uniqueness-level*[*of* - - *A r*] **by** *auto*
  have $\forall n.$ $\mathcal{B}$ *A I* $\cap$ *level A r n* $\neq$ {}
    using ‹*infinite-tree A r*›
      ‹*finitely-branching A r*› *I* *intersection-branch-set-nodes-at-level*[*of A r*]
    by *blast*
  hence *infinite* ($\bigcup n.$ ($\mathcal{B}$ *A I*) $\cap$ *level A r n*)
    using *family-intersection-infinita a* **by** *auto*
  thus *infinite* ($\mathcal{B}$ *A I*)**by** *auto*
qed

theorem *Koenig-Lemma*:
  assumes *infinite-tree* (*A*::′*nodes*:: *countable set*) *r*
  and *finitely-branching A r*
  shows $\exists B.$ *infinite-path B A r*
proof−
  have *satisfiable* ($\mathcal{T}$ *A r*)
  proof−
    have $\forall$ *S*. *S* $\subseteq$ ($\mathcal{T}$ *A r*) $\wedge$ (*finite S*) $\longrightarrow$ *satisfiable S*
      using ‹*infinite-tree A r*› ‹*finitely-branching A r*› *satisfiable-path*
      by *auto*
    thus *satisfiable* ($\mathcal{T}$ *A r*)
      using *Compactness-Theorem*[*of* ($\mathcal{T}$ *A r*)] **by** *auto*
  qed
  hence $\exists I.$ ($\forall F \in (\mathcal{T}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*)
    by(*unfold satisfiable-def*, *unfold model-def*, *auto*)
  then obtain *I* **where** *I*: $\forall F \in (\mathcal{T}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
    by *auto*
  hence $\forall F \in (\mathcal{F}\ A\ r) \cup (\mathcal{G}\ A\ r) \cup (\mathcal{H}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
    by(*unfold* $\mathcal{T}$-*def*)
  hence *I1*: $\forall F \in (\mathcal{F}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
  and *I2*: $\forall F \in (\mathcal{G}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
  and *I3*: $\forall F \in (\mathcal{H}\ A\ r)$. *t-v-evaluation I F* = *Ttrue*
    by *auto*
  let *?B* = ($\mathcal{B}$ *A I*)
  have *infinite-path ?B A r*
  proof(*unfold infinite-path-def*)
    show *path ?B A r* $\wedge$ *infinite ?B*
    proof(*rule conjI*)
      show *path ?B A r*
        using ‹*infinite-tree A r*› ‹*finitely-branching A r*› *I is-path*[*of A r*]

```
        by auto
      show infinite (ℬ A I)
        using ‹infinite-tree A r› ‹finitely-branching A r› I1 infinite-path
      by auto
    qed
  qed
  thus ∃B. infinite-path B A r by auto
qed

end
```

References

[1] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, second edition, 1996.

[2] F. F. Serrano Suárez. *Formalización en Isar de la Meta-Lógica de Primer Orden.* PhD thesis, Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Sevilla, Spain, 2012. https://idus.us.es/handle/11441/57780. In Spanish.

[3] R. M. Smullyan. *First-Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete. 2. Folge.* Springer-Verlag, Berlin, 1968. Also available as a Dover Publications Inc., 1994.

[4] F. F. S. Suárez, M. Ayala-Rincón, and T. A. de Lima. Hall's Theorem for Enumerable Families of Finite Sets. In *Proceedings 15th International Conference on Intelligent Computer Mathematics, CICM*, volume 13467 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2022.

[5] F. F. S. Suárez, M. Ayala-Rincón, and T. A. de Lima. Formalisation of Hall's Theorem for Countable Infinite Graphs. In *Proceedings 18th Colombian Conference on Computing, CCC.* Springer, 2024.