Formalizing a Seligman-Style Tableau System for Hybrid Logic

Asta Halkjær From

March 17, 2025

Abstract

This work is a formalization of soundness and completeness proofs for a Seligman-style tableau system for hybrid logic. The completeness result is obtained via a synthetic approach using maximally consistent sets of tableau blocks. The formalization differs from previous work [1, 2] in a few ways. First, to avoid the need to backtrack in the construction of a tableau, the formalized system has no unnamed initial segment, and therefore no Name rule. Second, I show that the full Bridge rule is admissible in the system. Third, I start from rules restricted to only extend the branch with new formulas, including only witnessing diamonds that are not already witnessed, and show that the unrestricted rules are admissible. Similarly, I start from simpler versions of the @-rules and show that these are sufficient. The GoTo rule is restricted using a notion of potential such that each application consumes potential and potential is earned through applications of the remaining rules. I show that if a branch can be closed then it can be closed starting from a single unit. Finally, Nom is restricted by a fixed set of allowed nominals. The resulting system should be terminating.

Preamble

The formalization was part of the author's MSc thesis in Computer Science and Engineering at the Technical University of Denmark (DTU).

Supervisors:

- Jørgen Villadsen
- Alexander Birch Jensen (co-supervisor)
- Patrick Blackburn (Roskilde University, external supervisor)

Contents

theory *Hybrid-Logic* **imports** *HOL−Library.Countable* **begin**

1 Syntax

datatype (′*a*, ′*b*) *fm*
  = *Pro* ′*a*
  | *Nom* ′*b*
  | *Neg* ‹(′*a*, ′*b*) *fm*› (‹¬ -› [*40*] *40*)
  | *Dis* ‹(′*a*, ′*b*) *fm*› ‹(′*a*, ′*b*) *fm*› (**infixr** ‹∨› *30*)
  | *Dia* ‹(′*a*, ′*b*) *fm*› (‹◇ -› *10*)
  | *Sat* ′*b* ‹(′*a*, ′*b*) *fm*› (‹**@** - -› *10*)

We can give other connectives as abbreviations.

abbreviation *Top* (‹⊤›) **where**
  ‹⊤ ≡ (*undefined* ∨ ¬ *undefined*)›

abbreviation *Con* (**infixr** ‹∧› *35*) **where**
  ‹*p* ∧ *q* ≡ ¬ (¬ *p* ∨ ¬ *q*)›

abbreviation *Imp* (**infixr** ‹⟶› *25*) **where**
  ‹*p* ⟶ *q* ≡ ¬ (*p* ∧ ¬ *q*)›

abbreviation *Box* (‹□ -› *10*) **where**
  ‹□ *p* ≡ ¬ (◇ ¬ *p*)›

primrec *nominals* :: ‹(′*a*, ′*b*) *fm* ⇒ ′*b set*› **where**
  ‹*nominals* (*Pro x*) = {}›
| ‹*nominals* (*Nom i*) = {*i*}›
| ‹*nominals* (¬ *p*) = *nominals p*›
| ‹*nominals* (*p* ∨ *q*) = *nominals p* ∪ *nominals q*›
| ‹*nominals* (◇ *p*) = *nominals p*›
| ‹*nominals* (**@** *i p*) = {*i*} ∪ *nominals p*›

primrec *sub* :: ‹(′*b* ⇒ ′*c*) ⇒ (′*a*, ′*b*) *fm* ⇒ (′*a*, ′*c*) *fm*› **where**
  ‹*sub* - (*Pro x*) = *Pro x*›
| ‹*sub f* (*Nom i*) = *Nom* (*f i*)›
| ‹*sub f* (¬ *p*) = (¬ *sub f p*)›
| ‹*sub f* (*p* ∨ *q*) = (*sub f p* ∨ *sub f q*)›
| ‹*sub f* (◇ *p*) = (◇ *sub f p*)›
| ‹*sub f* (**@** *i p*) = (**@** (*f i*) (*sub f p*))›

lemma *sub-nominals*: ‹*nominals* (*sub f p*) = *f* ‘ *nominals p*›
  by (*induct p*) *auto*

lemma *sub-id*: ‹*sub id p* = *p*›
  by (*induct p*) *simp-all*

lemma *sub-upd-fresh*: ‹*i* ∉ *nominals p* ⟹ *sub* (*f*(*i* := *j*)) *p* = *sub f p*›
  by (*induct p*) *auto*

2 Semantics

Type variable $'w$ stands for the set of worlds and $'a$ for the set of propositional symbols. The accessibility relation is given by R and the valuation by V. The mapping from nominals to worlds is an extra argument g to the semantics.

```
datatype ('w, 'a) model =
  Model (R: ‹'w ⇒ 'w set›) (V: ‹'w ⇒ 'a ⇒ bool›)
```

```
primrec semantics
  :: ‹('w, 'a) model ⇒ ('b ⇒ 'w) ⇒ 'w ⇒ ('a, 'b) fm ⇒ bool›
  (‹-, -, - ⊨ -› [50, 50, 50] 50) where
  ‹(M, -, w ⊨ Pro x) = V M w x›
| ‹(-, g, w ⊨ Nom i) = (w = g i)›
| ‹(M, g, w ⊨ ¬ p) = (¬ M, g, w ⊨ p)›
| ‹(M, g, w ⊨ (p ∨ q)) = ((M, g, w ⊨ p) ∨ (M, g, w ⊨ q))›
| ‹(M, g, w ⊨ ◇ p) = (∃ v ∈ R M w. M, g, v ⊨ p)›
| ‹(M, g, - ⊨ @ i p) = (M, g, g i ⊨ p)›
```

```
lemma ‹M, g, w ⊨ ⊤›
  by simp
```

```
lemma semantics-fresh:
  ‹i ∉ nominals p ⟹ (M, g, w ⊨ p) = (M, g(i := v), w ⊨ p)›
  by (induct p arbitrary: w) auto
```

2.1 Examples

```
abbreviation is-named :: ‹('w, 'b) model ⇒ bool› where
  ‹is-named M ≡ ∀ w. ∃ a. V M a = w›
```

```
abbreviation reflexive :: ‹('w, 'b) model ⇒ bool› where
  ‹reflexive M ≡ ∀ w. w ∈ R M w›
```

```
abbreviation irreflexive :: ‹('w, 'b) model ⇒ bool› where
  ‹irreflexive M ≡ ∀ w. w ∉ R M w›
```

```
abbreviation symmetric :: ‹('w, 'b) model ⇒ bool› where
  ‹symmetric M ≡ ∀ v w. w ∈ R M v ⟷ v ∈ R M w›
```

```
abbreviation asymmetric :: ‹('w, 'b) model ⇒ bool› where
  ‹asymmetric M ≡ ∀ v w. ¬ (w ∈ R M v ∧ v ∈ R M w)›
```

```
abbreviation transitive :: ‹('w, 'b) model ⇒ bool› where
  ‹transitive M ≡ ∀ v w x. w ∈ R M v ∧ x ∈ R M w ⟶ x ∈ R M v›
```

```
abbreviation universal :: ‹('w, 'b) model ⇒ bool› where
  ‹universal M ≡ ∀ v w. v ∈ R M w›
```

lemma ‹*irreflexive M* ⟹ *M*, *g*, *w* ⊨ **@** *i* ¬ (◊ *Nom i*)›
proof −
  assume ‹*irreflexive M*›
  then have ‹*g i* ∉ *R M* (*g i*)›
    by *simp*
  then have ‹¬ *M*, *g*, *g i* ⊨ ◊ *Nom i*›
    by *simp*
  then have ‹*M*, *g*, *g i* ⊨ ¬ (◊ *Nom i*)›
    by *simp*
  then show ‹*M*, *g*, *w* ⊨ **@** *i* ¬ (◊ *Nom i*)›
    by *simp*
qed

We can automatically show some characterizations of frames by pure axioms.

lemma ‹*irreflexive M* = (∀ *g w*. *M*, *g*, *w* ⊨ **@** *i* ¬ (◊ *Nom i*))›
  by *auto*

lemma ‹*asymmetric M* = (∀ *g w*. *M*, *g*, *w* ⊨ **@** *i* (□ ¬ (◊ *Nom i*)))›
  by *auto*

lemma ‹*universal M* = (∀ *g w*. *M*, *g*, *w* ⊨ ◊ *Nom i*)›
  by *auto*

3 Tableau

A block is defined as a list of formulas paired with an opening nominal. The opening nominal is not necessarily in the list. A branch is a list of blocks.

type-synonym ('*a*, '*b*) *block* = ‹('*a*, '*b*) *fm list* × '*b*›
type-synonym ('*a*, '*b*) *branch* = ‹('*a*, '*b*) *block list*›

abbreviation *member-list* :: ‹'*a* ⇒ '*a list* ⇒ *bool*› (‹- ∈. -› [*51*, *51*] *50*) **where**
  ‹*x* ∈. *xs* ≡ *x* ∈ *set xs*›

The predicate *on* presents the opening nominal as appearing on the block.

primrec *on* :: ‹('*a*, '*b*) *fm* ⇒ ('*a*, '*b*) *block* ⇒ *bool*› (‹- *on* -› [*51*, *51*] *50*) **where**
  ‹*p on* (*ps*, *i*) = (*p* ∈. *ps* ∨ *p* = *Nom i*)›

syntax
  -Ballon :: ‹*pttrn* ⇒ '*a set* ⇒ *bool* ⇒ *bool*› (‹(*3*∀(-/*on*-)./ -)› [*0*, *0*, *10*] *10*)
  -Bexon :: ‹*pttrn* ⇒ '*a set* ⇒ *bool* ⇒ *bool*› (‹(*3*∃(-/*on*-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
  -Ballon ⇌ *All* **and**
  -Bexon ⇌ *Ex*

translations
  ∀ *p on A*. *P* ⇀ ∀ *p*. *p on A* ⟶ *P*

∃ p on A. P ⇀ ∃ p. p on A ∧ P

abbreviation *list-nominals* :: ‹(′a, ′b) fm list ⇒ ′b set› **where**
‹list-nominals ps ≡ (⋃ p ∈ set ps. nominals p)›

primrec *block-nominals* :: ‹(′a, ′b) block ⇒ ′b set› **where**
‹block-nominals (ps, i) = {i} ∪ list-nominals ps›

definition *branch-nominals* :: ‹(′a, ′b) branch ⇒ ′b set› **where**
‹branch-nominals branch ≡ (⋃ block ∈ set branch. block-nominals block)›

abbreviation *at-in-branch* :: ‹(′a, ′b) fm ⇒ ′b ⇒ (′a, ′b) branch ⇒ bool› **where**
‹at-in-branch p a branch ≡ ∃ ps. (ps, a) ∈. branch ∧ p on (ps, a)›

notation *at-in-branch* (‹- at - in -› [51, 51, 51] 50)

definition *new* :: ‹(′a, ′b) fm ⇒ ′b ⇒ (′a, ′b) branch ⇒ bool› **where**
‹new p a branch ≡ ¬ p at a in branch›

definition *witnessed* :: ‹(′a, ′b) fm ⇒ ′b ⇒ (′a, ′b) branch ⇒ bool› **where**
‹witnessed p a branch ≡ ∃ i. (@ i p) at a in branch ∧ (◇ Nom i) at a in branch›

A branch has a closing tableau iff it is contained in the following inductively defined set. In that case I call the branch closeable. The first argument on the left of the turnstile, A, is a fixed set of nominals restricting Nom. This set rules out the copying of nominals and accessibility formulas introduced by DiaP. The second argument is "potential", used to restrict the GoTo rule.

inductive *STA* :: ‹′b set ⇒ nat ⇒ (′a, ′b) branch ⇒ bool› (‹-, - ⊢ -› [50, 50, 50] 50)
for *A* :: ‹′b set› **where**
Close:
‹p at i in branch ⟹ (¬ p) at i in branch ⟹
A, n ⊢ branch›
| *Neg*:
‹(¬ ¬ p) at a in (ps, a) # branch ⟹
new p a ((ps, a) # branch) ⟹
A, Suc n ⊢ (p # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *DisP*:
‹(p ∨ q) at a in (ps, a) # branch ⟹
new p a ((ps, a) # branch) ⟹ new q a ((ps, a) # branch) ⟹
A, Suc n ⊢ (p # ps, a) # branch ⟹ A, Suc n ⊢ (q # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *DisN*:
‹(¬ (p ∨ q)) at a in (ps, a) # branch ⟹
new (¬ p) a ((ps, a) # branch) ∨ new (¬ q) a ((ps, a) # branch) ⟹
A, Suc n ⊢ ((¬ q) # (¬ p) # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *DiaP*:

‹(◇ p) at a in (ps, a) # branch ⟹
i ∉ A ∪ branch-nominals ((ps, a) # branch) ⟹
∄ a. p = Nom a ⟹ ¬ witnessed p a ((ps, a) # branch) ⟹
A, Suc n ⊢ ((@ i p) # (◇ Nom i) # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *DiaN*:
‹(¬ (◇ p)) at a in (ps, a) # branch ⟹
(◇ Nom i) at a in (ps, a) # branch ⟹
new (¬ (@ i p)) a ((ps, a) # branch) ⟹
A, Suc n ⊢ ((¬ (@ i p)) # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *SatP*:
‹(@ a p) at b in (ps, a) # branch ⟹
new p a ((ps, a) # branch) ⟹
A, Suc n ⊢ (p # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *SatN*:
‹(¬ (@ a p)) at b in (ps, a) # branch ⟹
new (¬ p) a ((ps, a) # branch) ⟹
A, Suc n ⊢ ((¬ p) # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›
| *GoTo*:
‹i ∈ branch-nominals branch ⟹
A, n ⊢ ([], i) # branch ⟹
A, Suc n ⊢ branch›
| *Nom*:
‹p at b in (ps, a) # branch ⟹ Nom a at b in (ps, a) # branch ⟹
∀ i. p = Nom i ∨ p = (◇ Nom i) ⟶ i ∈ A ⟹
new p a ((ps, a) # branch) ⟹
A, Suc n ⊢ (p # ps, a) # branch ⟹
A, n ⊢ (ps, a) # branch›

abbreviation *STA-ex-potential* :: ‹'b set ⇒ ('a, 'b) branch ⇒ bool› (‹- ⊢ -› [50, 50] 50) **where**
‹A ⊢ branch ≡ ∃ n. A, n ⊢ branch›

lemma *STA-Suc*: ‹A, n ⊢ branch ⟹ A, Suc n ⊢ branch›
by (*induct n branch rule*: *STA.induct*) (*simp-all add*: *STA.intros*)

A verified derivation in the calculus.

lemma
fixes *i*
defines ‹p ≡ ¬ (@ i (Nom i))›
shows ‹A, Suc n ⊢ [([p], a)]›
proof −
have ‹i ∈ branch-nominals [([p], a)]›
unfolding *p-def branch-nominals-def* **by** *simp*
then have *?thesis* **if** ‹A, n ⊢ [([], i), ([p], a)]›
using *that GoTo* **by** *fast*

moreover have ‹*new* (¬ *Nom i*) *i* [([], *i*), ([*p*], *a*)]›
  unfolding *p-def new-def* **by** *auto*
moreover have ‹(¬ (**@** *i* (*Nom i*))) *at a in* [([], *i*), ([*p*], *a*)]›
  unfolding *p-def* **by** *fastforce*
ultimately have *?thesis* **if** ‹*A*, *Suc n* ⊢ [([¬ *Nom i*], *i*), ([*p*], *a*)]›
  using *that SatN* **by** *fast*
then show *?thesis*
  by (*meson Close list.set-intros*(*1*) *on.simps*)
qed

4 Soundness

An *i*-block is satisfied by a model *M* and assignment *g* if all formulas on the block are true under *M* at the world *g i* A branch is satisfied by a model and assignment if all blocks on it are.

primrec *block-sat* :: ‹(*'w*, *'a*) *model* ⇒ (*'b* ⇒ *'w*) ⇒ (*'a*, *'b*) *block* ⇒ *bool*›
  (‹-, - $\models_B$ -› [*50*, *50*] *50*) **where**
  ‹(*M*, *g* $\models_B$ (*ps*, *i*)) = (∀ *p on* (*ps*, *i*). *M*, *g*, *g i* ⊨ *p*)›

abbreviation *branch-sat* ::
  ‹(*'w*, *'a*) *model* ⇒ (*'b* ⇒ *'w*) ⇒ (*'a*, *'b*) *branch* ⇒ *bool*›
  (‹-, - $\models_\Theta$ -› [*50*, *50*] *50*) **where**
  ‹*M*, *g* $\models_\Theta$ *branch* ≡ ∀ (*ps*, *i*) ∈ *set branch*. *M*, *g* $\models_B$ (*ps*, *i*)›

lemma *block-nominals*:
  ‹*p on block* ⟹ *i* ∈ *nominals p* ⟹ *i* ∈ *block-nominals block*›
  by (*induct block*) *auto*

lemma *block-sat-fresh*:
  assumes ‹*M*, *g* $\models_B$ *block*› ‹*i* ∉ *block-nominals block*›
  shows ‹*M*, *g*(*i* := *v*) $\models_B$ *block*›
  using *assms*
proof (*induct block*)
  case (*Pair ps a*)
  then have ‹∀ *p on* (*ps*, *a*). *i* ∉ *nominals p*›
    using *block-nominals* **by** *fast*
  moreover have ‹*i* ≠ *a*›
    using *calculation* **by** *simp*
  ultimately have ‹∀ *p on* (*ps*, *a*). *M*, *g*(*i* := *v*), (*g*(*i* := *v*)) *a* ⊨ *p*›
    using *Pair semantics-fresh* **by** *fastforce*
  then show *?case*
    by (*meson block-sat.simps*)
qed

lemma *branch-sat-fresh*:
  assumes ‹*M*, *g* $\models_\Theta$ *branch*› ‹*i* ∉ *branch-nominals branch*›
  shows ‹*M*, *g*(*i* := *v*) $\models_\Theta$ *branch*›
  using *assms* **using** *block-sat-fresh* **unfolding** *branch-nominals-def* **by** *fast*

If a branch has a derivation then it cannot be satisfied.

```
lemma soundness': ‹A, n ⊢ branch ⟹ M, g ⊨Θ branch ⟹ False›
proof (induct n branch arbitrary: g rule: STA.induct)
  case (Close p i branch)
  then have ‹M, g, g i ⊨ p› ‹M, g, g i ⊨ ¬ p›
    by fastforce+
  then show ?case
    by simp
next
  case (Neg p a ps branch)
  have ‹M, g, g a ⊨ p›
    using Neg(1, 5) by fastforce
  then have ‹M, g ⊨Θ (p # ps, a) # branch›
    using Neg(5) by simp
  then show ?case
    using Neg(4) by blast
next
  case (DisP p q a ps branch)
  consider ‹M, g, g a ⊨ p› | ‹M, g, g a ⊨ q›
    using DisP(1, 8) by fastforce
  then consider
    ‹M, g ⊨Θ (p # ps, a) # branch› |
    ‹M, g ⊨Θ (q # ps, a) # branch›
    using DisP(8) by auto
  then show ?case
    using DisP(5, 7) by metis
next
  case (DisN p q a ps branch)
  have ‹M, g, g a ⊨ ¬ p› ‹M, g, g a ⊨ ¬ q›
    using DisN(1, 5) by fastforce+
  then have ‹M, g ⊨Θ ((¬ q) # (¬ p) # ps, a) # branch›
    using DisN(5) by simp
  then show ?case
    using DisN(4) by blast
next
  case (DiaP p a ps branch i)
  then have *: ‹M, g ⊨B (ps, a)›
    by simp

  have ‹i ∉ nominals p›
    using DiaP(1−2) unfolding branch-nominals-def by fastforce

  have ‹M, g, g a ⊨ ◊ p›
    using DiaP(1, 7) by fastforce
  then obtain v where ‹v ∈ R M (g a)› ‹M, g, v ⊨ p›
    by auto
  then have ‹M, g(i := v), v ⊨ p›
    using ‹i ∉ nominals p› semantics-fresh by metis
  then have ‹M, g(i := v), g a ⊨ @ i p›
```

```
    by simp
  moreover have ‹M, g(i := v), g a ⊨ ◇ Nom i›
    using ‹v ∈ R M (g a)› by simp
  moreover have ‹M, g(i := v) ⊨Θ (ps, a) # branch›
    using DiaP(2, 7) branch-sat-fresh by fast
  moreover have ‹i ∉ block-nominals (ps, a)›
    using DiaP(2) unfolding branch-nominals-def by simp
  then have ‹∀ p on (ps, a). M, g(i := v), g a ⊨ p›
    using ∗ semantics-fresh by fastforce
  ultimately have
    ‹M, g(i := v) ⊨Θ ((@ i p) # (◇ Nom i) # ps, a) # branch›
    by auto
  then show ?case
    using DiaP by blast
next
  case (DiaN p a ps branch i)
  have ‹M, g, g a ⊨ ¬ (◇ p)› ‹M, g, g a ⊨ ◇ Nom i›
    using DiaN(1−2, 6) by fastforce+
  then have ‹M, g, g a ⊨ ¬ (@ i p)›
    by simp
  then have ‹M, g ⊨Θ ((¬ (@ i p)) # ps, a) # branch›
    using DiaN(6) by simp
  then show ?thesis
    using DiaN(5) by blast
next
  case (SatP a p b ps branch)
  have ‹M, g, g a ⊨ p›
    using SatP(1, 5) by fastforce
  then have ‹M, g ⊨Θ (p # ps, a) # branch›
    using SatP(5) by simp
  then show ?case
    using SatP(4) by blast
next
  case (SatN a p b ps branch)
  have ‹M, g, g a ⊨ ¬ p›
    using SatN(1, 5) by fastforce
  then have ‹M, g ⊨Θ ((¬ p) # ps, a) # branch›
    using SatN(5) by simp
  then show ?case
    using SatN(4) by blast
next
  case (GoTo i branch)
  then show ?case
    by auto
next
  case (Nom p b ps a branch)
  have ‹M, g, g b ⊨ p› ‹M, g, g b ⊨ Nom a›
    using Nom(1−2, 7) by fastforce+
  moreover have ‹M, g ⊨B (ps, a)›
```

```
    using Nom(7) by simp
  ultimately have ‹M, g ⊨B (p # ps, a)›
    by simp
  then have ‹M, g ⊨Θ (p # ps, a) # branch›
    using Nom(7) by simp
  then show ?case
    using Nom(6) by blast
qed

lemma block-sat: ‹∀ p on block. M, g, w ⊨ p ⟹ M, g ⊨B block›
  by (induct block) auto

lemma branch-sat:
  assumes ‹∀ (ps, i) ∈ set branch. ∀ p on (ps, i). M, g, w ⊨ p›
  shows ‹M, g ⊨Θ branch›
  using assms block-sat by fast

lemma soundness:
  assumes ‹A, n ⊢ branch›
  shows ‹∃ block ∈ set branch. ∃ p on block. ¬ M, g, w ⊨ p›
  using assms soundness' branch-sat by fast

corollary ‹¬ A, n ⊢ []›
  using soundness by fastforce

theorem soundness-fresh:
  assumes ‹A, n ⊢ [([¬ p], i)]› ‹i ∉ nominals p›
  shows ‹M, g, w ⊨ p›
proof −
  from assms(1) have ‹M, g, g i ⊨ p› for g
    using soundness by fastforce
  then have ‹M, g(i := w), (g(i := w)) i ⊨ p›
    by blast
  then have ‹M, g(i := w), w ⊨ p›
    by simp
  then have ‹M, g(i := g i), w ⊨ p›
    using assms(2) semantics-fresh by metis
  then show ?thesis
    by simp
qed
```

5 No Detours

We only need to spend initial potential when we apply GoTo twice in a row. Otherwise another rule will have been applied in-between that justifies the GoTo. Therefore, by filtering out detours we can close any closeable branch starting from a single unit of potential.

```
primrec nonempty :: ‹('a, 'b) block ⇒ bool› where
```

```
  ‹nonempty (ps, i) = (ps ≠ [])›

lemma nonempty-Suc:
  assumes
    ‹A, n ⊢ (ps, a) # filter nonempty left @ right›
    ‹q at a in (ps, a) # filter nonempty left @ right› ‹q ≠ Nom a›
  shows ‹A, Suc n ⊢ filter nonempty ((ps, a) # left) @ right›
proof (cases ps)
  case Nil
  then have ‹a ∈ branch-nominals (filter nonempty left @ right)›
    unfolding branch-nominals-def using assms(2−3) by fastforce
  then show ?thesis
    using assms(1) Nil GoTo by auto
next
  case Cons
  then show ?thesis
    using assms(1) STA-Suc by auto
qed

lemma STA-nonempty:
  ‹A, n ⊢ left @ right ⟹ A, Suc m ⊢ filter nonempty left @ right›
proof (induct n ‹left @ right› arbitrary: left right rule: STA.induct)
  case (Close p i n)
  have ‹(¬ p) at i in filter nonempty left @ right›
    using Close(2) by fastforce
  moreover from this have ‹p at i in filter nonempty left @ right›
    using Close(1) by fastforce
  ultimately show ?case
    using STA.Close by fast
next
  case (Neg p a ps branch n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ (p # ps, a) # branch›
      using Neg(4) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using Neg(1−2) STA.Neg by fast
    then show ?thesis
      using Nil Neg(5) STA-Suc by auto
  next
    case (Cons - left′)
    then have ‹A, Suc m ⊢ (p # ps, a) # filter nonempty left′ @ right›
      using Neg(4)[where left=‹- # left′›] Neg(5) by fastforce
    moreover have ∗: ‹(¬ ¬ p) at a in (ps, a) # filter nonempty left′ @ right›
      using Cons Neg(1, 5) by fastforce
    moreover have ‹new p a ((ps, a) # filter nonempty left′ @ right)›
      using Cons Neg(2, 5) unfolding new-def by auto
    ultimately have ‹A, m ⊢ (ps, a) # filter nonempty left′ @ right›
```

```
      using STA.Neg by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left′) @ right›
      using ∗ nonempty-Suc by fast
    then show ?thesis
      using Cons Neg(5) by auto
  qed
next
  case (DisP p q a ps branch n)
  then show ?case
  proof (cases left)
    case Nil
     then have ‹A, Suc m ⊢ (p # ps, a) # branch› ‹A, Suc m ⊢ (q # ps, a) #
branch›
      using DisP(5, 7) by fastforce+
    then have ‹A, m ⊢ (ps, a) # branch›
      using DisP(1−3) STA.DisP by fast
    then show ?thesis
      using Nil DisP(8) STA-Suc by auto
  next
    case (Cons - left′)
    then have
      ‹A, Suc m ⊢ (p # ps, a) # filter nonempty left′ @ right›
      ‹A, Suc m ⊢ (q # ps, a) # filter nonempty left′ @ right›
      using DisP(5, 7)[where left=‹- # left′›] DisP(8) by fastforce+
    moreover have ∗: ‹(p ∨ q) at a in (ps, a) # filter nonempty left′ @ right›
      using Cons DisP(1, 8) by fastforce
    moreover have
      ‹new p a ((ps, a) # filter nonempty left′ @ right)›
      ‹new q a ((ps, a) # filter nonempty left′ @ right)›
      using Cons DisP(2−3, 8) unfolding new-def by auto
    ultimately have ‹A, m ⊢ (ps, a) # filter nonempty left′ @ right›
      using STA.DisP by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left′) @ right›
      using ∗ nonempty-Suc by fast
    then show ?thesis
      using Cons DisP(8) by auto
  qed
next
  case (DisN p q a ps branch n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ ((¬ q) # (¬ p) # ps, a) # branch›
      using DisN(4) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using DisN(1−2) STA.DisN by fast
    then show ?thesis
      using Nil DisN(5) STA-Suc by auto
  next
```

```
      case (Cons - left')
      then have ‹A, Suc m ⊢ ((¬ q) # (¬ p) # ps, a) # filter nonempty left' @
right›
        using DisN(4)[where left=‹- # left'›] DisN(5) by fastforce
      moreover have *: ‹(¬ (p ∨ q)) at a in (ps, a) # filter nonempty left' @ right›
        using Cons DisN(1, 5) by fastforce
      moreover consider
        ‹new (¬ p) a ((ps, a) # filter nonempty left' @ right)› |
        ‹new (¬ q) a ((ps, a) # filter nonempty left' @ right)›
        using Cons DisN(2, 5) unfolding new-def by auto
      ultimately have ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
        using STA.DisN by metis
      then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
        using * nonempty-Suc by fast
      then show ?thesis
        using Cons DisN(5) by auto
    qed
next
  case (DiaP p a ps branch i n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ ((@ i p) # (◇ Nom i) # ps, a) # branch›
      using DiaP(6) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using DiaP(1−4) STA.DiaP by fast
    then show ?thesis
      using Nil DiaP(7) STA-Suc by auto
  next
    case (Cons - left')
    then have ‹A, Suc m ⊢ ((@ i p) # (◇ Nom i) # ps, a) # filter nonempty left'
@ right›
      using DiaP(6)[where left=‹- # left'›] DiaP(7) by fastforce
    moreover have *: ‹(◇ p) at a in (ps, a) # filter nonempty left' @ right›
      using Cons DiaP(1, 7) by fastforce
    moreover have ‹i ∉ A ∪ branch-nominals ((ps, a) # filter nonempty left' @
right)›
      using Cons DiaP(2, 7) unfolding branch-nominals-def by auto
    moreover have ‹¬ witnessed p a ((ps, a) # filter nonempty left' @ right)›
      using Cons DiaP(4, 7) unfolding witnessed-def by auto
    ultimately have ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
      using DiaP(3) STA.DiaP by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
      using * nonempty-Suc by fast
    then show ?thesis
      using Cons DiaP(7) by auto
  qed
next
  case (DiaN p a ps branch i n)
```

```
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ ((¬ (@ i p)) # ps, a) # branch›
      using DiaN(5) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using DiaN(1−3) STA.DiaN by fast
    then show ?thesis
      using Nil DiaN(6) STA-Suc by auto
  next
    case (Cons - left')
    then have ‹A, Suc m ⊢ ((¬ (@ i p)) # ps, a) # filter nonempty left' @ right›
      using DiaN(5)[where left=‹- # left'›] DiaN(6) by fastforce
    moreover have ∗: ‹(¬ (◇ p)) at a in (ps, a) # filter nonempty left' @ right›
      using Cons DiaN(1, 6) by fastforce
    moreover have ∗: ‹(◇ Nom i) at a in (ps, a) # filter nonempty left' @ right›
      using Cons DiaN(2, 6) by fastforce
    moreover have ‹new (¬ (@ i p)) a ((ps, a) # filter nonempty left' @ right)›
      using Cons DiaN(3, 6) unfolding new-def by auto
    ultimately have ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
      using STA.DiaN by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
      using ∗ nonempty-Suc by fast
    then show ?thesis
      using Cons DiaN(6) by auto
  qed
next
  case (SatP a p b ps branch n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ (p # ps, a) # branch›
      using SatP(4) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using SatP(1−2) STA.SatP by fast
    then show ?thesis
      using Nil SatP(5) STA-Suc by auto
  next
    case (Cons - left')
    then have ‹A, Suc m ⊢ (p # ps, a) # filter nonempty left' @ right›
      using SatP(4)[where left=‹- # left'›] SatP(5) by fastforce
    moreover have ‹(@ a p) at b in (ps, a) # filter nonempty left' @ right›
      using Cons SatP(1, 5) by fastforce
    moreover have ‹new p a ((ps, a) # filter nonempty left' @ right)›
      using Cons SatP(2, 5) unfolding new-def by auto
    ultimately have ∗: ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
      using STA.SatP by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
    proof (cases ps)
```

```
      case Nil
      then have ‹a ∈ branch-nominals (filter nonempty left' @ right)›
        unfolding branch-nominals-def using SatP(1, 5) Cons by fastforce
      then show ?thesis
        using * Nil GoTo by fastforce
    next
      case Cons
      then show ?thesis
        using * STA-Suc by auto
    qed
    then show ?thesis
      using Cons SatP(5) by auto
  qed
next
  case (SatN a p b ps branch n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ ((¬ p) # ps, a) # branch›
      using SatN(4) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using SatN(1−2) STA.SatN by fast
    then show ?thesis
      using Nil SatN(5) STA-Suc by auto
  next
    case (Cons - left')
    then have ‹A, Suc m ⊢ ((¬ p) # ps, a) # filter nonempty left' @ right›
      using SatN(4)[where left=‹- # left'›] SatN(5) by fastforce
    moreover have ‹(¬ (@ a p)) at b in (ps, a) # filter nonempty left' @ right›
      using Cons SatN(1, 5) by fastforce
    moreover have ‹new (¬ p) a ((ps, a) # filter nonempty left' @ right)›
      using Cons SatN(2, 5) unfolding new-def by auto
    ultimately have *: ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
      using STA.SatN by fast
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
    proof (cases ps)
      case Nil
      then have ‹a ∈ branch-nominals (filter nonempty left' @ right)›
        unfolding branch-nominals-def using SatN(1, 5) Cons by fastforce
      then show ?thesis
        using * Nil GoTo by fastforce
    next
      case Cons
      then show ?thesis
        using * STA-Suc by auto
    qed
    then show ?thesis
      using Cons SatN(5) by auto
  qed
```

```
next
  case (GoTo i n)
  show ?case
    using GoTo(3)[where left=‹([], i) # left›] by simp
next
  case (Nom p b ps a branch n)
  then show ?case
  proof (cases left)
    case Nil
    then have ‹A, Suc m ⊢ (p # ps, a) # branch›
      using Nom(6) by fastforce
    then have ‹A, m ⊢ (ps, a) # branch›
      using Nom(1−4) STA.Nom by metis
    then show ?thesis
      using Nil Nom(7) STA-Suc by auto
  next
    case (Cons - left')
    then have ‹A, Suc m ⊢ (p # ps, a) # filter nonempty left' @ right›
      using Nom(6)[where left=‹- # left'›] Nom(7) by fastforce
    moreover have
      ‹p at b in (ps, a) # filter nonempty left' @ right› and a:
      ‹Nom a at b in (ps, a) # filter nonempty left' @ right›
      using Cons Nom(1−2, 7) by simp-all (metis empty-iff empty-set)+
    moreover have ‹new p a ((ps, a) # filter nonempty left' @ right)›
      using Cons Nom(4, 7) unfolding new-def by auto
    ultimately have ∗: ‹A, m ⊢ (ps, a) # filter nonempty left' @ right›
      using Nom(3) STA.Nom by metis
    then have ‹A, Suc m ⊢ filter nonempty ((ps, a) # left') @ right›
    proof (cases ps)
      case Nil
      moreover have ‹a ≠ b›
        using Nom(1, 4) unfolding new-def by blast
      ultimately have ‹a ∈ branch-nominals (filter nonempty left' @ right)›
        using a unfolding branch-nominals-def by fastforce
      then show ?thesis
        using ∗ Nil GoTo by auto
    next
      case Cons
      then show ?thesis
        using ∗ STA-Suc by auto
    qed
    then show ?thesis
      using Cons Nom(7) by auto
  qed
qed

theorem STA-potential: ‹A, n ⊢ branch ⟹ A, Suc m ⊢ branch›
  using STA-nonempty[where left=‹[]›] by auto
```

corollary *STA-one*: ‹*A*, *n* ⊢ *branch* ⟹ *A*, *1* ⊢ *branch*›
  using *STA-potential* **by** *auto*

5.1 Free GoTo

The above result allows us to prove a version of GoTo that works "for free."

lemma *GoTo′*:
  assumes ‹*A*, *Suc n* ⊢ ([], *i*) # *branch*› ‹*i* ∈ *branch-nominals branch*›
  shows ‹*A*, *Suc n* ⊢ *branch*›
  using *assms GoTo STA-potential* **by** *fast*

6 Indexed Mapping

This section contains some machinery for showing admissible rules.

6.1 Indexing

We use pairs of natural numbers to index into the branch. The first component specifies the block and the second specifies the formula on that block. We index from the back to ensure that indices are stable under the addition of new formulas and blocks.

primrec *rev-nth* :: ‹*′a list* ⇒ *nat* ⇒ *′a option*› (**infixl** ‹!.› *100*) **where**
  ‹[] !. *v* = *None*›
| ‹(*x* # *xs*) !. *v* = (*if length xs* = *v then Some x else xs* !. *v*)›

lemma *rev-nth-last*: ‹*xs* !. *0* = *Some x* ⟹ *last xs* = *x*›
  by (*induct xs*) *auto*

lemma *rev-nth-zero*: ‹(*xs* @ [*x*]) !. *0* = *Some x*›
  by (*induct xs*) *auto*

lemma *rev-nth-snoc*: ‹(*xs* @ [*x*]) !. *Suc v* = *Some y* ⟹ *xs* !. *v* = *Some y*›
  by (*induct xs*) *auto*

lemma *rev-nth-Suc*: ‹(*xs* @ [*x*]) !. *Suc v* = *xs* !. *v*›
  by (*induct xs*) *auto*

lemma *rev-nth-bounded*: ‹*v* < *length xs* ⟹ ∃ *x*. *xs* !. *v* = *Some x*›
  by (*induct xs*) *simp-all*

lemma *rev-nth-Cons*: ‹*xs* !. *v* = *Some y* ⟹ (*x* # *xs*) !. *v* = *Some y*›
proof (*induct xs arbitrary*: *v rule*: *rev-induct*)
  case (*snoc a xs*)
  then show *?case*
  proof (*induct v*)
    case (*Suc v*)

    then have ‹*xs* !. *v* = *Some y*›
      using *rev-nth-snoc* **by** *fast*
    then have ‹(*x* # *xs*) !. *v* = *Some y*›
      using *Suc*(*2*) **by** *blast*
    then show *?case*
      using *Suc*(*3*) **by** *auto*
  qed *simp*
qed *simp*

lemma *rev-nth-append*: ‹*xs* !. *v* = *Some y* ⟹ (*ys* @ *xs*) !. *v* = *Some y*›
  using *rev-nth-Cons*[**where** *xs*=‹- @ *xs*›] **by** (*induct ys*) *simp-all*

lemma *rev-nth-mem*: ‹*block* ∈. *branch* ⟷ (∃ *v*. *branch* !. *v* = *Some block*)›
proof
  assume ‹*block* ∈. *branch*›
  then show ‹∃ *v*. *branch* !. *v* = *Some block*›
  proof (*induct branch*)
    case (*Cons block′ branch*)
    then show *?case*
    proof (*cases* ‹*block* = *block′*›)
      case *False*
      then have ‹∃ *v*. *branch* !. *v* = *Some block*›
        using *Cons* **by** *simp*
      then show *?thesis*
        using *rev-nth-Cons* **by** *fast*
    qed *auto*
  qed *simp*
next
  assume ‹∃ *v*. *branch* !. *v* = *Some block*›
  then show ‹*block* ∈. *branch*›
  proof (*induct branch*)
    case (*Cons block′ branch*)
    then show *?case*
      by *simp* (*metis option.sel*)
  qed *simp*
qed

lemma *rev-nth-on*: ‹*p on* (*ps*, *i*) ⟷ (∃ *v*. *ps* !. *v* = *Some p*) ∨ *p* = *Nom i*›
  by (*simp add*: *rev-nth-mem*)

lemma *rev-nth-Some*: ‹*xs* !. *v* = *Some y* ⟹ *v* < *length xs*›
proof (*induct xs arbitrary*: *v rule*: *rev-induct*)
  case (*snoc x xs*)
  then show *?case*
    by (*induct v*) (*simp-all*, *metis rev-nth-snoc*)
qed *simp*

lemma *index-Cons*:
  assumes ‹((*ps*, *a*) # *branch*) !. *v* = *Some* (*qs*, *b*)› ‹*qs* !. *v′* = *Some q*›

shows ‹∃ *qs'*. ((*p* # *ps*, *a*) # *branch*) !. *v* = *Some* (*qs'*, *b*) ∧ *qs'* !. *v'* = *Some q*›
proof −
have
‹((*p* # *ps*, *a*) # *branch*) !. *v* = *Some* (*qs*, *b*) ∨
((*p* # *ps*, *a*) # *branch*) !. *v* = *Some* (*p* # *qs*, *b*)›
using *assms*(*1*) **by** *auto*
moreover have ‹*qs* !. *v'* = *Some q*› ‹(*p* # *qs*) !. *v'* = *Some q*›
using *assms*(*2*) *rev-nth-Cons* **by** *fast+*
ultimately show *?thesis*
by *fastforce*
qed

6.2 Mapping

primrec *mapi* :: ‹(*nat* ⇒ *'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list*› **where**
‹*mapi f* [] = []›
| ‹*mapi f* (*x* # *xs*) = *f* (*length xs*) *x* # *mapi f xs*›

primrec *mapi-block* ::
‹(*nat* ⇒ (*'a*, *'b*) *fm* ⇒ (*'a*, *'b*) *fm*) ⇒ ((*'a*, *'b*) *block* ⇒ (*'a*, *'b*) *block*)› **where**
‹*mapi-block f* (*ps*, *i*) = (*mapi f ps*, *i*)›

definition *mapi-branch* ::
‹(*nat* ⇒ *nat* ⇒ (*'a*, *'b*) *fm* ⇒ (*'a*, *'b*) *fm*) ⇒ ((*'a*, *'b*) *branch* ⇒ (*'a*, *'b*) *branch*)›
where
‹*mapi-branch f branch* ≡ *mapi* (λ*v*. *mapi-block* (*f v*)) *branch*›

abbreviation *mapper* ::
‹((*'a*, *'b*) *fm* ⇒ (*'a*, *'b*) *fm*) ⇒
(*nat* × *nat*) *set* ⇒ *nat* ⇒ *nat* ⇒ (*'a*, *'b*) *fm* ⇒ (*'a*, *'b*) *fm*› **where**
‹*mapper f xs v v' p* ≡ (*if* (*v*, *v'*) ∈ *xs then f p else p*)›

lemma *mapi-block-add-oob*:
assumes ‹*length ps* ≤ *v'*›
shows
‹*mapi-block* (*mapper f* ({(*v*, *v'*)} ∪ *xs*) *v*) (*ps*, *i*) =
mapi-block (*mapper f xs v*) (*ps*, *i*)›
using *assms* **by** (*induct ps*) *simp-all*

lemma *mapi-branch-add-oob*:
assumes ‹*length branch* ≤ *v*›
shows
‹*mapi-branch* (*mapper f* ({(*v*, *v'*)} ∪ *xs*)) *branch* =
mapi-branch (*mapper f xs*) *branch*›
unfolding *mapi-branch-def* **using** *assms* **by** (*induct branch*) *simp-all*

lemma *mapi-branch-head-add-oob*:
‹*mapi-branch* (*mapper f* ({(*length branch*, *length ps*)} ∪ *xs*)) ((*ps*, *a*) # *branch*)
=

mapi-branch (*mapper f xs*) ((*ps*, *a*) # *branch*)›
using *mapi-branch-add-oob*[**where** *branch*=*branch*] **unfolding** *mapi-branch-def*
using *mapi-block-add-oob*[**where** *ps*=*ps*] **by** *simp*

lemma *mapi-branch-mem*:
assumes ‹(*ps*, *i*) ∈. *branch*›
shows ‹∃ *v*. (*mapi* (*f v*) *ps*, *i*) ∈. *mapi-branch f branch*›
unfolding *mapi-branch-def* **using** *assms* **by** (*induct branch*) *auto*

lemma *rev-nth-mapi-branch*:
assumes ‹*branch* !. *v* = *Some* (*ps*, *a*)›
shows ‹(*mapi* (*f v*) *ps*, *a*) ∈. *mapi-branch f branch*›
unfolding *mapi-branch-def* **using** *assms*
by (*induct branch*) (*simp-all*, *metis mapi-block.simps option.inject*)

lemma *rev-nth-mapi-block*:
assumes ‹*ps* !. *v'* = *Some p*›
shows ‹*f v' p on* (*mapi f ps*, *a*)›
using *assms* **by** (*induct ps*) (*simp-all*, *metis option.sel*)

lemma *mapi-append*:
‹*mapi f* (*xs* @ *ys*) = *mapi* (λ*v*. *f* (*v* + *length ys*)) *xs* @ *mapi f ys*›
by (*induct xs*) *simp-all*

lemma *mapi-block-id*: ‹*mapi-block* (*mapper f* {} *v*) (*ps*, *i*) = (*ps*, *i*)›
by (*induct ps*) *auto*

lemma *mapi-branch-id*: ‹*mapi-branch* (*mapper f* {}) *branch* = *branch*›
unfolding *mapi-branch-def* **using** *mapi-block-id* **by** (*induct branch*) *auto*

lemma *length-mapi*: ‹*length* (*mapi f xs*) = *length xs*›
by (*induct xs*) *auto*

lemma *mapi-rev-nth*:
assumes ‹*xs* !. *v* = *Some x*›
shows ‹*mapi f xs* !. *v* = *Some* (*f v x*)›
using *assms*
proof (*induct xs arbitrary*: *v*)
case (*Cons y xs*)
have ∗: ‹*mapi f* (*y* # *xs*) = *f* (*length xs*) *y* # *mapi f xs*›
by *simp*
show *?case*
proof (*cases* ‹*v* = *length xs*›)
case *True*
then have ‹*mapi f* (*y* # *xs*) !. *v* = *Some* (*f* (*length xs*) *y*)›
using *length-mapi* ∗ **by** (*metis rev-nth.simps*(*2*))
then show *?thesis*
using *Cons.prems True* **by** *auto*
next

case *False*
then show *?thesis*
using ∗ *Cons length-mapi* **by** (*metis rev-nth.simps*(*2*))
qed
qed *simp*

7 Duplicate Formulas

7.1 Removable indices

abbreviation ‹*proj* ≡ *Equiv-Relations.proj*›

definition *all-is* :: ‹(′*a*, ′*b*) *fm* ⇒ (′*a*, ′*b*) *fm list* ⇒ *nat set* ⇒ *bool*› **where**
‹*all-is p ps xs* ≡ ∀ *v* ∈ *xs*. *ps* !. *v* = *Some p*›

definition *is-at* :: ‹(′*a*, ′*b*) *fm* ⇒ ′*b* ⇒ (′*a*, ′*b*) *branch* ⇒ *nat* ⇒ *nat* ⇒ *bool*› **where**
‹*is-at p i branch v v*′ ≡ ∃ *ps*. *branch* !. *v* = *Some* (*ps*, *i*) ∧ *ps* !. *v*′ = *Some p*›

This definition is slightly complicated by the inability to index the opening nominal.

definition *is-elsewhere* :: ‹(′*a*, ′*b*) *fm* ⇒ ′*b* ⇒ (′*a*, ′*b*) *branch* ⇒ (*nat* × *nat*) *set* ⇒ *bool*› **where**
‹*is-elsewhere p i branch xs* ≡ ∃ *w w*′ *ps*. (*w*, *w*′) ∉ *xs* ∧
branch !. *w* = *Some* (*ps*, *i*) ∧ (*p* = *Nom i* ∨ *ps* !. *w*′ = *Some p*)›

definition *Dup* :: ‹(′*a*, ′*b*) *fm* ⇒ ′*b* ⇒ (′*a*, ′*b*) *branch* ⇒ (*nat* × *nat*) *set* ⇒ *bool*›
where
‹*Dup p i branch xs* ≡ ∀ (*v*, *v*′) ∈ *xs*.
is-at p i branch v v′ ∧ *is-elsewhere p i branch xs*›

lemma *Dup-all-is*:
assumes ‹*Dup p i branch xs*› ‹*branch* !. *v* = *Some* (*ps*, *a*)›
shows ‹*all-is p ps* (*proj xs v*)›
using *assms* **unfolding** *Dup-def is-at-def all-is-def proj-def* **by** *auto*

lemma *Dup-branch*:
‹*Dup p i branch xs* ⟹ *Dup p i* (*extra* @ *branch*) *xs*›
unfolding *Dup-def is-at-def is-elsewhere-def* **using** *rev-nth-append* **by** *fast*

lemma *Dup-block*:
assumes ‹*Dup p i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹*Dup p i* ((*ps*′ @ *ps*, *a*) # *branch*) *xs*›
unfolding *Dup-def*
proof *safe*
fix *v v*′
assume ‹(*v*, *v*′) ∈ *xs*›
then show ‹*is-at p i* ((*ps*′ @ *ps*, *a*) # *branch*) *v v*′›
using *assms rev-nth-append* **unfolding** *Dup-def is-at-def* **by** *fastforce*
next

```
  fix v v′
  assume ‹(v, v′) ∈ xs›
  then obtain w w′ qs where
    ‹(w, w′) ∉ xs› ‹((ps, a) # branch) !. w = Some (qs, i)›
    ‹p = Nom i ∨ qs !. w′ = Some p›
    using assms unfolding Dup-def is-elsewhere-def by blast
  then have
    ‹∃ qs. ((ps′ @ ps, a) # branch) !. w = Some (qs, i) ∧
     (p = Nom i ∨ qs !. w′ = Some p)›
    using rev-nth-append by fastforce
  then show ‹is-elsewhere p i ((ps′ @ ps, a) # branch) xs›
    unfolding is-elsewhere-def using ‹(w, w′) ∉ xs› by blast
qed

definition only-touches :: ‹′b ⇒ (′a, ′b) branch ⇒ (nat × nat) set ⇒ bool› where
  ‹only-touches i branch xs ≡ ∀ (v, v′) ∈ xs. ∀ ps a. branch !. v = Some (ps, a) ⟶
i = a›

lemma Dup-touches: ‹Dup p i branch xs ⟹ only-touches i branch xs›
  unfolding Dup-def is-at-def only-touches-def by auto

lemma only-touches-opening:
  assumes ‹only-touches i branch xs› ‹(v, v′) ∈ xs› ‹branch !. v = Some (ps, a)›
  shows ‹i = a›
  using assms unfolding only-touches-def is-at-def by auto

lemma Dup-head:
  ‹Dup p i ((ps, a) # branch) xs ⟹ Dup p i ((q # ps, a) # branch) xs›
  using Dup-block[where ps′=‹[-]›] by simp

lemma Dup-head-oob′:
  assumes ‹Dup p i ((ps, a) # branch) xs›
  shows ‹(length branch, k + length ps) ∉ xs›
  using assms rev-nth-Some unfolding Dup-def is-at-def by fastforce

lemma Dup-head-oob:
  assumes ‹Dup p i ((ps, a) # branch) xs›
  shows ‹(length branch, length ps) ∉ xs›
  using assms Dup-head-oob′[where k=0] by fastforce
```

7.2 Omitting formulas

```
primrec omit :: ‹nat set ⇒ (′a, ′b) fm list ⇒ (′a, ′b) fm list› where
  ‹omit xs [] = []›
| ‹omit xs (p # ps) = (if length ps ∈ xs then omit xs ps else p # omit xs ps)›

primrec omit-block :: ‹nat set ⇒ (′a, ′b) block ⇒ (′a, ′b) block› where
  ‹omit-block xs (ps, a) = (omit xs ps, a)›
```

definition *omit-branch* :: ‹(*nat* × *nat*) *set* ⇒ ('*a*, '*b*) *branch* ⇒ ('*a*, '*b*) *branch*›
where
‹*omit-branch xs branch* ≡ *mapi* (λ*v*. *omit-block* (*proj xs v*)) *branch*›

lemma *omit-mem*: ‹*ps* !. *v* = *Some p* ⟹ *v* ∉ *xs* ⟹ *p* ∈. *omit xs ps*›
proof (*induct ps*)
  case (*Cons q ps*)
  then show *?case*
    by (*cases* ‹*v* = *length ps*›) *simp-all*
qed *simp*

lemma *omit-id*: ‹*omit* {} *ps* = *ps*›
  by (*induct ps*) *auto*

lemma *omit-block-id*: ‹*omit-block* {} *block* = *block*›
  using *omit-id* **by** (*cases block*) *simp*

lemma *omit-branch-id*: ‹*omit-branch* {} *branch* = *branch*›
  unfolding *omit-branch-def proj-def* **using** *omit-block-id*
  by (*induct branch*) *fastforce+*

lemma *omit-branch-mem-diff-opening*:
  assumes ‹*only-touches i branch xs*› ‹(*ps*, *a*) ∈. *branch*› ‹*i* ≠ *a*›
  shows ‹(*ps*, *a*) ∈. *omit-branch xs branch*›
proof −
  obtain *v* **where** *v*: ‹*branch* !. *v* = *Some* (*ps*, *a*)›
    using *assms*(*2*) *rev-nth-mem* **by** *fast*
  then have ‹*omit-branch xs branch* !. *v* = *Some* (*omit* (*proj xs v*) *ps*, *a*)›
    unfolding *omit-branch-def* **by** (*simp add*: *mapi-rev-nth*)
  then have ∗: ‹(*omit* (*proj xs v*) *ps*, *a*) ∈. *omit-branch xs branch*›
    using *rev-nth-mem* **by** *fast*
  moreover have ‹*proj xs v* = {}›
    unfolding *proj-def* **using** *assms*(*1*, *3*) *v only-touches-opening* **by** *fast*
  then have ‹*omit* (*proj xs v*) *ps* = *ps*›
    using *omit-id* **by** *auto*
  ultimately show *?thesis*
    by *simp*
qed

lemma *Dup-omit-branch-mem-same-opening*:
  assumes ‹*Dup p i branch xs*› ‹*p at i in branch*›
  shows ‹*p at i in omit-branch xs branch*›
proof −
  obtain *ps* **where** *ps*: ‹(*ps*, *i*) ∈. *branch*› ‹*p on* (*ps*, *i*)›
    using *assms*(*2*) **by** *blast*
  then obtain *v* **where** *v*: ‹*branch* !. *v* = *Some* (*ps*, *i*)›
    using *rev-nth-mem* **by** *fast*
  then have ‹*omit-branch xs branch* !. *v* = *Some* (*omit* (*proj xs v*) *ps*, *i*)›
    unfolding *omit-branch-def* **by** (*simp add*: *mapi-rev-nth*)

then have ∗: ‹(*omit* (*proj xs v*) *ps*, *i*) ∈. *omit-branch xs branch*›
  using *rev-nth-mem* **by** *fast*

consider
  v′ **where** ‹*ps* !. *v*′ = *Some p*› ‹(*v*, *v*′) ∈ *xs*› |
  v′ **where** ‹*ps* !. *v*′ = *Some p*› ‹(*v*, *v*′) ∉ *xs*› |
  ‹*p* = *Nom i*›
  using *ps v rev-nth-mem* **by** *fastforce*
then show *?thesis*
proof *cases*
  case (*1 v*′)
  then obtain *qs w w*′ **where** *qs*:
    ‹(*w*, *w*′) ∉ *xs*› ‹*branch* !. *w* = *Some* (*qs*, *i*)› ‹*p* = *Nom i* ∨ *qs* !. *w*′ = *Some p*›
    using *assms*(*1*) **unfolding** *Dup-def is-elsewhere-def* **by** *blast*
  then have ‹*omit-branch xs branch* !. *w* = *Some* (*omit* (*proj xs w*) *qs*, *i*)›
    unfolding *omit-branch-def* **by** (*simp add*: *mapi-rev-nth*)
  then have ‹(*omit* (*proj xs w*) *qs*, *i*) ∈. *omit-branch xs branch*›
    using *rev-nth-mem* **by** *fast*
  moreover have ‹*p on* (*omit* (*proj xs w*) *qs*, *i*)›
    unfolding *proj-def* **using** *qs*(*1*, *3*) *omit-mem* **by** *fastforce*
  ultimately show *?thesis*
    by *blast*
next
  case (*2 v*′)
  then show *?thesis*
    using ∗ *omit-mem* **unfolding** *proj-def*
    by (*metis Image-singleton-iff on.simps*)
next
  case *3*
  then show *?thesis*
    using ∗ **by** *auto*
qed
qed

lemma *omit-del*:
  assumes ‹*p* ∈. *ps*› ‹*p* ∉ *set* (*omit xs ps*)›
  shows ‹∃ *v*. *ps* !. *v* = *Some p* ∧ *v* ∈ *xs*›
  using *assms omit-mem rev-nth-mem* **by** *metis*

lemma *omit-all-is*:
  assumes ‹*all-is p ps xs*› ‹*q* ∈. *ps*› ‹*q* ∉ *set* (*omit xs ps*)›
  shows ‹*q* = *p*›
  using *assms omit-del* **unfolding** *all-is-def* **by** *fastforce*

definition *all-is-branch* :: ‹(′*a*, ′*b*) *fm* ⇒ ′*b* ⇒ (′*a*, ′*b*) *branch* ⇒ (*nat* × *nat*) *set* ⇒ *bool*› **where**
  ‹*all-is-branch p i branch xs* ≡ ∀ (*v*, *v*′) ∈ *xs*. *v* < *length branch* ⟶ *is-at p i branch v v*′›

lemma *all-is-branch*:
‹*all-is-branch p i branch xs* ⟹ *branch* !. *v* = *Some* (*ps*, *a*) ⟹ *all-is p ps* (*proj xs v*)›
unfolding *all-is-branch-def is-at-def all-is-def proj-def* **using** *rev-nth-Some* **by** *fastforce*

lemma *Dup-all-is-branch*: ‹*Dup p i branch xs* ⟹ *all-is-branch p i branch xs*›
unfolding *all-is-branch-def Dup-def* **by** *fast*

lemma *omit-branch-mem-diff-formula*:
assumes ‹*all-is-branch p i branch xs*› ‹*q at i in branch*› ‹*p* ≠ *q*›
shows ‹*q at i in omit-branch xs branch*›
proof −
obtain *ps* **where** *ps*: ‹(*ps*, *i*) ∈. *branch*› ‹*q on* (*ps*, *i*)›
using *assms*(*2*) **by** *blast*
then obtain *v* **where** *v*: ‹*branch* !. *v* = *Some* (*ps*, *i*)›
using *rev-nth-mem* **by** *fast*
then have ‹*omit-branch xs branch* !. *v* = *Some* (*omit* (*proj xs v*) *ps*, *i*)›
unfolding *omit-branch-def* **by** (*simp add*: *mapi-rev-nth*)
then have ∗: ‹(*omit* (*proj xs v*) *ps*, *i*) ∈. *omit-branch xs branch*›
using *rev-nth-mem* **by** *fast*
moreover have ‹*all-is p ps* (*proj xs v*)›
using *assms*(*1*) *v all-is-branch* **by** *fast*
then have ‹*q on* (*omit* (*proj xs v*) *ps*, *i*)›
using *ps assms*(*3*) *omit-all-is* **by** *auto*
ultimately show *?thesis*
by *blast*
qed

lemma *Dup-omit-branch-mem*:
assumes ‹*Dup p i branch xs*› ‹*q at a in branch*›
shows ‹*q at a in omit-branch xs branch*›
using *assms omit-branch-mem-diff-opening Dup-touches Dup-omit-branch-mem-same-opening omit-branch-mem-diff-formula Dup-all-is-branch* **by** *fast*

lemma *omit-set*: ‹*set* (*omit xs ps*) ⊆ *set ps*›
by (*induct ps*) *auto*

lemma *on-omit*: ‹*p on* (*omit xs ps*, *i*) ⟹ *p on* (*ps*, *i*)›
using *omit-set* **by** *auto*

lemma *all-is-set*:
assumes ‹*all-is p ps xs*›
shows ‹{*p*} ∪ *set* (*omit xs ps*) = {*p*} ∪ *set ps*›
using *assms omit-all-is omit-set* **unfolding** *all-is-def* **by** *fast*

lemma *all-is-list-nominals*:
assumes ‹*all-is p ps xs*›
shows ‹*nominals p* ∪ *list-nominals* (*omit xs ps*) = *nominals p* ∪ *list-nominals*

ps›
  using *assms all-is-set* **by** *fastforce*

lemma *all-is-block-nominals*:
  assumes ‹*all-is p ps xs*›
  shows ‹*nominals p* ∪ *block-nominals* (*omit xs ps*, *i*) = *nominals p* ∪ *block-nominals* (*ps*, *i*)›
  using *assms* **by** (*simp add*: *all-is-list-nominals*)

lemma *all-is-branch-nominals′*:
  assumes ‹*all-is-branch p i branch xs*›
  shows
    ‹*nominals p* ∪ *branch-nominals* (*omit-branch xs branch*) =
     nominals p ∪ *branch-nominals branch*›
proof −
  have ‹∀ (*v*, *v′*) ∈ *xs*. *v* < *length branch* ⟶ *is-at p i branch v v′*›
    using *assms* **unfolding** *all-is-branch-def is-at-def* **by** *auto*
  then show *?thesis*
  proof (*induct branch*)
    case *Nil*
    then show *?case*
      unfolding *omit-branch-def* **by** *simp*
  next
    case (*Cons block branch*)
    then show *?case*
    proof (*cases block*)
      case (*Pair ps a*)
      have ‹∀ (*v*, *v′*) ∈ *xs*. *v* < *length branch* ⟶ *is-at p i branch v v′*›
        using *Cons*(*2*) *rev-nth-Cons* **unfolding** *is-at-def* **by** *auto*
      then have
        ‹*nominals p* ∪ *branch-nominals* (*omit-branch xs branch*) =
         nominals p ∪ *branch-nominals branch*›
        using *Cons*(*1*) **by** *blast*
      then have
        ‹*nominals p* ∪ *branch-nominals* (*omit-branch xs* ((*ps*, *a*) # *branch*)) =
         nominals p ∪ *block-nominals* (*omit* (*proj xs* (*length branch*)) *ps*, *a*) ∪
          branch-nominals branch›
        unfolding *branch-nominals-def omit-branch-def* **by** *auto*
      moreover have ‹*all-is p ps* (*proj xs* (*length branch*))›
        using *Cons*(*2*) *Pair* **unfolding** *proj-def all-is-def is-at-def* **by** *auto*
      then have
        ‹*nominals p* ∪ *block-nominals* (*omit* (*proj xs* (*length branch*)) *ps*, *a*) =
         nominals p ∪ *block-nominals* (*ps*, *a*)›
        using *all-is-block-nominals* **by** *fast*
      then have
        ‹*nominals p* ∪ *block-nominals* (*omit-block* (*proj xs* (*length branch*)) (*ps*, *a*))
=
         nominals p ∪ *block-nominals* (*ps*, *a*)›
        by *simp*

ultimately have
‹*nominals p* ∪ *branch-nominals* (*omit-branch xs* ((*ps*, *a*) # *branch*)) =
nominals p ∪ *block-nominals* (*ps*, *a*) ∪ *branch-nominals branch*›
by *auto*
then show *?thesis*
unfolding *branch-nominals-def* **using** *Pair* **by** *auto*
qed
qed
qed

lemma *Dup-branch-nominals*:
assumes ‹*Dup p i branch xs*›
shows ‹*branch-nominals* (*omit-branch xs branch*) = *branch-nominals branch*›
proof (*cases* ‹*xs* = {}›)
case *True*
then show *?thesis*
using *omit-branch-id* **by** *metis*
next
case *False*
with *assms* **obtain** *ps w w′* **where**
‹(*w*, *w′*) ∉ *xs*› ‹*branch* !. *w* = *Some* (*ps*, *i*)› ‹*p* = *Nom i* ∨ *ps* !. *w′* = *Some p*›
unfolding *Dup-def is-elsewhere-def* **by** *fast*
then have ∗: ‹(*ps*, *i*) ∈. *branch*› ‹*p on* (*ps*, *i*)›
using *rev-nth-mem rev-nth-on* **by** *fast*+
then have ‹*nominals p* ⊆ *branch-nominals branch*›
unfolding *branch-nominals-def* **using** *block-nominals* **by** *fast*
moreover obtain *ps′* **where**
‹(*ps′*, *i*) ∈. *omit-branch xs branch*› ‹*p on* (*ps′*, *i*)›
using *assms* ∗ *Dup-omit-branch-mem* **by** *fast*
then have ‹*nominals p* ⊆ *branch-nominals* (*omit-branch xs branch*)›
unfolding *branch-nominals-def* **using** *block-nominals* **by** *fast*
moreover have
‹*nominals p* ∪ *branch-nominals* (*omit-branch xs branch*) =
nominals p ∪ *branch-nominals branch*›
using *assms all-is-branch-nominals′ Dup-all-is-branch* **by** *fast*
ultimately show *?thesis*
by *blast*
qed

lemma *omit-branch-mem-dual*:
assumes ‹*p at i in omit-branch xs branch*›
shows ‹*p at i in branch*›
proof −
obtain *ps* **where** *ps*: ‹(*ps*, *i*) ∈. *omit-branch xs branch*› ‹*p on* (*ps*, *i*)›
using *assms*(*1*) **by** *blast*
then obtain *v* **where** *v*: ‹*omit-branch xs branch* !. *v* = *Some* (*ps*, *i*)›
using *rev-nth-mem* **unfolding** *omit-branch-def* **by** *fast*
then have ‹*v* < *length* (*omit-branch xs branch*)›
using *rev-nth-Some* **by** *fast*

```
  then have ‹v < length branch›
    unfolding omit-branch-def using length-mapi by metis
  then obtain ps' i' where ps': ‹branch !. v = Some (ps', i')›
    using rev-nth-bounded by (metis surj-pair)
  then have ‹omit-branch xs branch !. v = Some (omit (proj xs v) ps', i')›
    unfolding omit-branch-def by (simp add: mapi-rev-nth)
  then have ‹ps = omit (proj xs v) ps'› ‹i = i'›
    using v by simp-all
  then have ‹p on (ps', i)›
    using ps omit-set by auto
  moreover have ‹(ps', i) ∈. branch›
    using ps' ‹i = i'› rev-nth-mem by fast
  ultimately show ?thesis
    using ‹ps = omit (proj xs v) ps'› by blast
qed

lemma witnessed-omit-branch:
  assumes ‹witnessed p a (omit-branch xs branch)›
  shows ‹witnessed p a branch›
proof −
  obtain ps qs i where
    ps: ‹(ps, a) ∈. omit-branch xs branch› ‹(@ i p) on (ps, a)› and
    qs: ‹(qs, a) ∈. omit-branch xs branch› ‹(◇ Nom i) on (qs, a)›
    using assms unfolding witnessed-def by blast
  from ps obtain ps' where
    ‹(ps', a) ∈. branch› ‹(@ i p) on (ps', a)›
    using omit-branch-mem-dual by fast
  moreover from qs obtain qs' where
    ‹(qs', a) ∈. branch› ‹(◇ Nom i) on (qs', a)›
    using omit-branch-mem-dual by fast
  ultimately show ?thesis
    unfolding witnessed-def by blast
qed

lemma new-omit-branch:
  assumes ‹new p a branch›
  shows ‹new p a (omit-branch xs branch)›
  using assms omit-branch-mem-dual unfolding new-def by fast

lemma omit-oob:
  assumes ‹length ps ≤ v›
  shows ‹omit ({v} ∪ xs) ps = omit xs ps›
  using assms by (induct ps) simp-all

lemma omit-branch-oob:
  assumes ‹length branch ≤ v›
  shows ‹omit-branch ({(v, v')} ∪ xs) branch = omit-branch xs branch›
  using assms
proof (induct branch)
```

```
  case Nil
  then show ?case
    unfolding omit-branch-def by simp
next
  case (Cons block branch)
  let ?xs = ‹({(v, v')} ∪ xs)›
  show ?case
  proof (cases block)
    case (Pair ps a)
    then have
      ‹omit-branch ?xs ((ps, a) # branch) =
        (omit (proj ?xs (length branch)) ps, a) # omit-branch xs branch›
      using Cons unfolding omit-branch-def by simp
    moreover have ‹proj ?xs (length branch) = proj xs (length branch)›
      using Cons(2) unfolding proj-def by auto
    ultimately show ?thesis
      unfolding omit-branch-def by simp
  qed
qed
```

7.3 Induction

```
lemma STA-Dup:
  assumes ‹A, n ⊢ branch› ‹Dup q i branch xs›
  shows ‹A, n ⊢ omit-branch xs branch›
  using assms
proof (induct n branch)
  case (Close p i' branch n)
  have ‹p at i' in omit-branch xs branch›
    using Close(1, 3) Dup-omit-branch-mem by fast
  moreover have ‹(¬ p) at i' in omit-branch xs branch›
    using Close(2, 3) Dup-omit-branch-mem by fast
  ultimately show ?case
    using STA.Close by fast
next
  case (Neg p a ps branch n)
  have ‹A, Suc n ⊢ omit-branch xs ((p # ps, a) # branch)›
    using Neg(4−) Dup-head by fast
  moreover have ‹(length branch, length ps) ∉ xs›
    using Neg(5) Dup-head-oob by fast
  ultimately have
   ‹A, Suc n ⊢ (p # omit (proj xs (length branch)) ps, a) # omit-branch xs branch›
    unfolding omit-branch-def proj-def by simp
  moreover have ‹(¬ ¬ p) at a in omit-branch xs ((ps, a) # branch)›
    using Neg(1, 5) Dup-omit-branch-mem by fast
  moreover have ‹new p a (omit-branch xs ((ps, a) # branch))›
    using Neg(2) new-omit-branch by fast
  ultimately show ?case
    by (simp add: omit-branch-def STA.Neg)
```

next
case (*DisP p q a ps branch n*)
have
‹*A*, *Suc n* ⊢ *omit-branch xs* ((*p* # *ps*, *a*) # *branch*)›
‹*A*, *Suc n* ⊢ *omit-branch xs* ((*q* # *ps*, *a*) # *branch*)›
using *DisP*(*4*−) *Dup-head* **by** *fast+*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *DisP*(*8*) *Dup-head-oob* **by** *fast*
ultimately have
‹*A*, *Suc n* ⊢ (*p* # *omit* (*proj xs* (*length branch*)) *ps*, *a*) # *omit-branch xs branch*›
‹*A*, *Suc n* ⊢ (*q* # *omit* (*proj xs* (*length branch*)) *ps*, *a*) # *omit-branch xs branch*›
unfolding *omit-branch-def proj-def* **by** *simp-all*
moreover have ‹(*p* ∨ *q*) *at a in omit-branch xs* ((*ps*, *a*) # *branch*)›
using *DisP*(*1*, *8*) *Dup-omit-branch-mem* **by** *fast*
moreover have ‹*new p a* (*omit-branch xs* ((*ps*, *a*) # *branch*))›
using *DisP*(*2*) *new-omit-branch* **by** *fast*
moreover have ‹*new q a* (*omit-branch xs* ((*ps*, *a*) # *branch*))›
using *DisP*(*3*) *new-omit-branch* **by** *fast*
ultimately show *?case*
by (*simp add*: *omit-branch-def STA.DisP*)
next
case (*DisN p q a ps branch n*)
have ‹*A*, *Suc n* ⊢ *omit-branch xs* (((¬ *q*) # (¬ *p*) # *ps*, *a*) # *branch*)›
using *DisN*(*4*−) *Dup-block*[**where** *ps′*=‹[-, -]›] **by** *fastforce*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *DisN*(*5*) *Dup-head-oob* **by** *fast*
moreover have ‹(*length branch*, *1* + *length ps*) ∉ *xs*›
using *DisN*(*5*) *Dup-head-oob′* **by** *fast*
ultimately have
‹*A*, *Suc n* ⊢ ((¬ *q*) # (¬ *p*) # *omit* (*proj xs* (*length branch*)) *ps*, *a*) #
omit-branch xs branch›
unfolding *omit-branch-def proj-def* **by** *simp*
moreover have ‹(¬ (*p* ∨ *q*)) *at a in omit-branch xs* ((*ps*, *a*) # *branch*)›
using *DisN*(*1*, *5*) *Dup-omit-branch-mem* **by** *fast*
moreover have
‹*new* (¬ *p*) *a* (*omit-branch xs* ((*ps*, *a*) # *branch*)) ∨
new (¬ *q*) *a* (*omit-branch xs* ((*ps*, *a*) # *branch*))›
using *DisN*(*2*) *new-omit-branch* **by** *fast*
ultimately show *?case*
by (*simp add*: *omit-branch-def STA.DisN*)
next
case (*DiaP p a ps branch i n*)
have ‹*A*, *Suc n* ⊢ *omit-branch xs* (((**@** *i p*) # (◇ *Nom i*) # *ps*, *a*) # *branch*)›
using *DiaP*(*4*−) *Dup-block*[**where** *ps′*=‹[-, -]›] **by** *fastforce*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *DiaP*(*7*) *Dup-head-oob* **by** *fast*
moreover have ‹(*length branch*, *1*+ *length ps*) ∉ *xs*›
using *DiaP*(*7*) *Dup-head-oob′* **by** *fast*
ultimately have

‹A, Suc n ⊢ ((@ i p) # (◇ Nom i) # omit (proj xs (length branch)) ps, a) #
omit-branch xs branch›
unfolding omit-branch-def proj-def **by** simp
moreover have ‹(◇ p) at a in omit-branch xs ((ps, a) # branch)›
using DiaP(1, 7) Dup-omit-branch-mem **by** fast
moreover have ‹i ∉ A ∪ branch-nominals (omit-branch xs ((ps, a) # branch))›
using DiaP(2, 7) Dup-branch-nominals **by** fast
moreover have ‹¬ witnessed p a (omit-branch xs ((ps, a) # branch))›
using DiaP(4) witnessed-omit-branch **by** fast
ultimately show ?case
using DiaP(3) **by** (simp add: omit-branch-def STA.DiaP)
next
case (DiaN p a ps branch i n)
have ‹A, Suc n ⊢ omit-branch xs (((¬ (@ i p)) # ps, a) # branch)›
using DiaN(4−) Dup-head **by** fast
moreover have ‹(length branch, length ps) ∉ xs›
using DiaN(6) Dup-head-oob **by** fast
ultimately have
‹A, Suc n ⊢ ((¬ (@ i p)) # omit (proj xs (length branch)) ps, a) #
omit-branch xs branch›
unfolding omit-branch-def proj-def **by** simp
moreover have ‹(¬ (◇ p)) at a in omit-branch xs ((ps, a) # branch)›
using DiaN(1, 6) Dup-omit-branch-mem **by** fast
moreover have ‹(◇ Nom i) at a in omit-branch xs ((ps, a) # branch)›
using DiaN(2, 6) Dup-omit-branch-mem **by** fast
moreover have ‹new (¬ (@ i p)) a (omit-branch xs ((ps, a) # branch))›
using DiaN(3) new-omit-branch **by** fast
ultimately show ?case
by (simp add: omit-branch-def STA.DiaN)
next
case (SatP a p b ps branch n)
have ‹A, Suc n ⊢ omit-branch xs ((p # ps, a) # branch)›
using SatP(4−) Dup-head **by** fast
moreover have ‹(length branch, length ps) ∉ xs›
using SatP(5) Dup-head-oob **by** fast
ultimately have
‹A, Suc n ⊢ (p # omit (proj xs (length branch)) ps, a) # omit-branch xs branch›
unfolding omit-branch-def proj-def **by** simp
moreover have ‹(@ a p) at b in omit-branch xs ((ps, a) # branch)›
using SatP(1, 5) Dup-omit-branch-mem **by** fast
moreover have ‹new p a (omit-branch xs ((ps, a) # branch))›
using SatP(2) new-omit-branch **by** fast
ultimately show ?case
by (simp add: omit-branch-def STA.SatP)
next
case (SatN a p b ps branch n)
have ‹A, Suc n ⊢ omit-branch xs (((¬ p) # ps, a) # branch)›
using SatN(4−) Dup-head **by** fast
moreover have ‹(length branch, length ps) ∉ xs›

using *SatN*(*5*) *Dup-head-oob* **by** *fast*
ultimately have
‹*A*, *Suc n* ⊢ ((¬ *p*) # *omit* (*proj xs* (*length branch*)) *ps*, *a*) # *omit-branch xs branch*›
unfolding *omit-branch-def proj-def* **by** *simp*
moreover have ‹(¬ (@ *a p*)) *at b in omit-branch xs* ((*ps*, *a*) # *branch*)›
using *SatN*(*1*, *5*) *Dup-omit-branch-mem* **by** *fast*
moreover have ‹*new* (¬ *p*) *a* (*omit-branch xs* ((*ps*, *a*) # *branch*))›
using *SatN*(*2*) *new-omit-branch* **by** *fast*
ultimately show *?case*
by (*simp add*: *omit-branch-def STA.SatN*)
next
case (*GoTo i branch n*)
then have ‹*A*, *n* ⊢ *omit-branch xs* (([], *i*) # *branch*)›
using *Dup-branch*[**where** *extra*=‹[([], *i*)]›] **by** *fastforce*
then have ‹*A*, *n* ⊢ ([], *i*) # *omit-branch xs branch*›
unfolding *omit-branch-def* **by** *simp*
moreover have ‹*i* ∈ *branch-nominals* (*omit-branch xs branch*)›
using *GoTo*(*1*, *4*) *Dup-branch-nominals* **by** *fast*
ultimately show *?case*
unfolding *omit-branch-def* **by** (*simp add*: *STA.GoTo*)
next
case (*Nom p b ps a branch n*)
have ‹*A*, *Suc n* ⊢ *omit-branch xs* ((*p* # *ps*, *a*) # *branch*)›
using *Nom*(*4*−) *Dup-head* **by** *fast*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *Nom*(*7*) *Dup-head-oob* **by** *fast*
ultimately have
‹*A*, *Suc n* ⊢ (*p* # *omit* (*proj xs* (*length branch*)) *ps*, *a*) # *omit-branch xs branch*›
unfolding *omit-branch-def proj-def* **by** *simp*
moreover have ‹*p at b in omit-branch xs* ((*ps*, *a*) # *branch*)›
using *Nom*(*1*, *7*) *Dup-omit-branch-mem* **by** *fast*
moreover have ‹*Nom a at b in omit-branch xs* ((*ps*, *a*) # *branch*)›
using *Nom*(*2*, *7*) *Dup-omit-branch-mem* **by** *fast*
moreover have ‹*new p a* (*omit-branch xs* ((*ps*, *a*) # *branch*))›
using *Nom*(*4*) *new-omit-branch* **by** *fast*
ultimately show *?case*
using *Nom*(*3*) **by** (*simp add*: *omit-branch-def STA.Nom*)
qed

theorem *Dup*:
assumes ‹*A*, *n* ⊢ (*p* # *ps*, *a*) # *branch*› ‹¬ *new p a* ((*ps*, *a*) # *branch*)›
shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
proof −
obtain *qs* **where** *qs*:
‹(*qs*, *a*) ∈. (*ps*, *a*) # *branch*› ‹*p on* (*qs*, *a*)›
using *assms*(*2*) **unfolding** *new-def* **by** *blast*

let *?xs* = ‹{(*length branch*, *length ps*)}›

```
  have *: ‹is-at p a ((p # ps, a) # branch) (length branch) (length ps)›
    unfolding is-at-def by simp

  have ‹Dup p a ((p # ps, a) # branch) ?xs›
  proof (cases ‹p = Nom a›)
    case True
    moreover have ‹((p # ps, a) # branch) !. length branch = Some (p # ps, a)›
      by simp
    moreover have ‹p on (p # ps, a)›
      by simp
    ultimately have ‹is-elsewhere p a ((p # ps, a) # branch) ?xs›
      unfolding is-elsewhere-def using assms(2) rev-nth-Some
      by (metis (mono-tags, lifting) Pair-inject less-le singletonD)
    then show ?thesis
      unfolding Dup-def using * by blast
  next
    case false: False
    then show ?thesis
    proof (cases ‹ps = qs›)
      case True
      then obtain w' where w': ‹qs !. w' = Some p›
        using qs(2) false rev-nth-mem by fastforce
      then have ‹(p # ps) !. w' = Some p›
        using True rev-nth-Cons by fast
      moreover have ‹((p # ps, a) # branch) !. length branch = Some (p # ps,
a)›
        by simp
      moreover have ‹(length branch, w') ∉ ?xs›
        using True w' rev-nth-Some by fast
      ultimately have ‹is-elsewhere p a ((p # ps, a) # branch) ?xs›
        unfolding is-elsewhere-def by fast
      then show ?thesis
        unfolding Dup-def using * by fast
    next
      case False
      then obtain w where w: ‹branch !. w = Some (qs, a)›
        using qs(1) rev-nth-mem by fastforce
      moreover obtain w' where w': ‹qs !. w' = Some p›
        using qs(2) false rev-nth-mem by fastforce
      moreover have ‹(w, w') ∉ ?xs›
        using rev-nth-Some w by fast
      ultimately have ‹is-elsewhere p a ((p # ps, a) # branch) ?xs›
        unfolding is-elsewhere-def using rev-nth-Cons by fast
      then show ?thesis
        unfolding Dup-def using * by fast
    qed
  qed
```

then have ‹*A*, *n* ⊢ *omit-branch ?xs* ((*p* # *ps*, *a*) # *branch*)›
  using *assms*(*1*) *STA-Dup* **by** *fast*
then have ‹*A*, *n* ⊢ (*omit* (*proj ?xs* (*length branch*)) *ps*, *a*) # *omit-branch ?xs branch*›
  unfolding *omit-branch-def proj-def* **by** *simp*
moreover have ‹*omit-branch ?xs branch* = *omit-branch* {} *branch*›
  using *omit-branch-oob* **by** *auto*
then have ‹*omit-branch ?xs branch* = *branch*›
  using *omit-branch-id* **by** *simp*
moreover have ‹*proj ?xs* (*length branch*) = {*length ps*}›
  unfolding *proj-def* **by** *blast*
then have ‹*omit* (*proj ?xs* (*length branch*)) *ps* = *omit* {} *ps*›
  using *omit-oob* **by** *auto*
then have ‹*omit* (*proj ?xs* (*length branch*)) *ps* = *ps*›
  using *omit-id* **by** *simp*
ultimately show *?thesis*
  by *simp*
qed

7.4 Unrestricted rules

lemma *STA-add*: ‹*A*, *n* ⊢ *branch* ⟹ *A*, *m* + *n* ⊢ *branch*›
  using *STA-Suc* **by** (*induct m*) *auto*

lemma *STA-le*: ‹*A*, *n* ⊢ *branch* ⟹ *n* ≤ *m* ⟹ *A*, *m* ⊢ *branch*›
  using *STA-add* **by** (*metis le-add-diff-inverse2*)

lemma *Neg′*:
  assumes
    ‹(¬ ¬ *p*) *at a in* (*ps*, *a*) # *branch*›
    ‹*A*, *n* ⊢ (*p* # *ps*, *a*) # *branch*›
  shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
  using *assms Neg Dup STA-Suc* **by** *metis*

lemma *DisP′*:
  assumes
    ‹(*p* ∨ *q*) *at a in* (*ps*, *a*) # *branch*›
    ‹*A*, *n* ⊢ (*p* # *ps*, *a*) # *branch*› ‹*A*, *n* ⊢ (*q* # *ps*, *a*) # *branch*›
  shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
proof (*cases* ‹*new p a* ((*ps*, *a*) # *branch*) ∧ *new q a* ((*ps*, *a*) # *branch*)›)
  case *True*
  moreover have ‹*A*, *Suc n* ⊢ (*p* # *ps*, *a*) # *branch*› ‹*A*, *Suc n* ⊢ (*q* # *ps*, *a*) # *branch*›
    using *assms*(*2*−*3*) *STA-Suc* **by** *fast*+
  ultimately show *?thesis*
    using *assms*(*1*) *DisP* **by** *fast*
next
  case *False*
  then show *?thesis*

```
    using assms Dup by fast
qed

lemma DisP'':
  assumes
    ‹(p ∨ q) at a in (ps, a) # branch›
    ‹A, n ⊢ (p # ps, a) # branch› ‹A, m ⊢ (q # ps, a) # branch›
  shows ‹A, max n m ⊢ (ps, a) # branch›
proof (cases ‹n ≤ m›)
  case True
  then have ‹A, m ⊢ (p # ps, a) # branch›
    using assms(2) STA-le by blast
  then show ?thesis
    using assms True by (simp add: DisP' max.absorb2)
next
  case False
  then have ‹A, n ⊢ (q # ps, a) # branch›
    using assms(3) STA-le by fastforce
  then show ?thesis
    using assms False by (simp add: DisP' max.absorb1)
qed

lemma DisN':
  assumes
    ‹(¬ (p ∨ q)) at a in (ps, a) # branch›
    ‹A, n ⊢ ((¬ q) # (¬ p) # ps, a) # branch›
  shows ‹A, n ⊢ (ps, a) # branch›
proof (cases ‹new (¬ q) a ((ps, a) # branch) ∨ new (¬ p) a ((ps, a) # branch)›)
  case True
  then show ?thesis
    using assms DisN STA-Suc by fast
next
  case False
  then show ?thesis
    using assms Dup
    by (metis (no-types, lifting) list.set-intros(1−2) new-def on.simps set-ConsD)
qed

lemma DiaP':
  assumes
    ‹(◇ p) at a in (ps, a) # branch›
    ‹i ∉ A ∪ branch-nominals ((ps, a) # branch)›
    ‹∄ a. p = Nom a›
    ‹¬ witnessed p a ((ps, a) # branch)›
    ‹A, n ⊢ ((@ i p) # (◇ Nom i) # ps, a) # branch›
  shows ‹A, n ⊢ (ps, a) # branch›
  using assms DiaP STA-Suc by fast

lemma DiaN':
```

assumes
‹(¬ (◊ *p*)) *at a in* (*ps*, *a*) # *branch*›
‹(◊ *Nom i*) *at a in* (*ps*, *a*) # *branch*›
‹*A*, *n* ⊢ ((¬ (**@** *i p*)) # *ps*, *a*) # *branch*›
shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
using *assms DiaN Dup STA-Suc* **by** *fast*

lemma *SatP'*:
assumes
‹(**@** *a p*) *at b in* (*ps*, *a*) # *branch*›
‹*A*, *n* ⊢ (*p* # *ps*, *a*) # *branch*›
shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
using *assms SatP Dup STA-Suc* **by** *fast*

lemma *SatN'*:
assumes
‹(¬ (**@** *a p*)) *at b in* (*ps*, *a*) # *branch*›
‹*A*, *n* ⊢ ((¬ *p*) # *ps*, *a*) # *branch*›
shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
using *assms SatN Dup STA-Suc* **by** *fast*

lemma *Nom'*:
assumes
‹*p at b in* (*ps*, *a*) # *branch*›
‹*Nom a at b in* (*ps*, *a*) # *branch*›
‹∀ *i*. *p* = *Nom i* ∨ *p* = (◊ *Nom i*) ⟶ *i* ∈ *A*›
‹*A*, *n* ⊢ (*p* # *ps*, *a*) # *branch*›
shows ‹*A*, *n* ⊢ (*ps*, *a*) # *branch*›
proof (*cases* ‹*new p a* ((*ps*, *a*) # *branch*)›)
case *True*
moreover have ‹*A*, *Suc n* ⊢ (*p* # *ps*, *a*) # *branch*›
using *assms*(*4*) *STA-Suc* **by** *blast*
ultimately show *?thesis*
using *assms*(*1*−*3*) *Nom* **by** *metis*
next
case *False*
then show *?thesis*
using *assms Dup* **by** *fast*
qed

8 Substitution

lemma *finite-nominals*: ‹*finite* (*nominals p*)›
by (*induct p*) *simp-all*

lemma *finite-block-nominals*: ‹*finite* (*block-nominals block*)›
using *finite-nominals* **by** (*induct block*) *auto*

lemma *finite-branch-nominals*: ‹*finite* (*branch-nominals branch*)›

unfolding *branch-nominals-def* **by** (*induct branch*) (*auto simp*: *finite-block-nominals*)

abbreviation *sub-list* :: ‹('*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *fm list* ⇒ ('*a*, '*c*) *fm list*› **where**
‹*sub-list f ps* ≡ *map* (*sub f*) *ps*›

primrec *sub-block* :: ‹('*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *block* ⇒ ('*a*, '*c*) *block*› **where**
‹*sub-block f* (*ps*, *i*) = (*sub-list f ps*, *f i*)›

definition *sub-branch* :: ‹('*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *branch* ⇒ ('*a*, '*c*) *branch*› **where**
‹*sub-branch f blocks* ≡ *map* (*sub-block f*) *blocks*›

lemma *sub-block-mem*: ‹*p on block* ⟹ *sub f p on sub-block f block*›
by (*induct block*) *auto*

lemma *sub-branch-mem*:
assumes ‹(*ps*, *i*) ∈. *branch*›
shows ‹(*sub-list f ps*, *f i*) ∈. *sub-branch f branch*›
unfolding *sub-branch-def* **using** *assms image-iff* **by** *fastforce*

lemma *sub-block-nominals*: ‹*block-nominals* (*sub-block f block*) = *f* ‘ *block-nominals block*›
by (*induct block*) (*auto simp*: *sub-nominals*)

lemma *sub-branch-nominals*:
‹*branch-nominals* (*sub-branch f branch*) = *f* ‘ *branch-nominals branch*›
unfolding *branch-nominals-def sub-branch-def*
by (*induct branch*) (*auto simp*: *sub-block-nominals*)

lemma *sub-list-id*: ‹*sub-list id ps* = *ps*›
using *sub-id* **by** (*induct ps*) *auto*

lemma *sub-block-id*: ‹*sub-block id block* = *block*›
using *sub-list-id* **by** (*induct block*) *auto*

lemma *sub-branch-id*: ‹*sub-branch id branch* = *branch*›
unfolding *sub-branch-def* **using** *sub-block-id* **by** (*induct branch*) *auto*

lemma *sub-block-upd-fresh*:
assumes ‹*i* ∉ *block-nominals block*›
shows ‹*sub-block* (*f*(*i* := *j*)) *block* = *sub-block f block*›
using *assms* **by** (*induct block*) (*auto simp add*: *sub-upd-fresh*)

lemma *sub-branch-upd-fresh*:
assumes ‹*i* ∉ *branch-nominals branch*›
shows ‹*sub-branch* (*f*(*i* := *j*)) *branch* = *sub-branch f branch*›
using *assms* **unfolding** *branch-nominals-def sub-branch-def*
by (*induct branch*) (*auto simp*: *sub-block-upd-fresh*)

lemma *sub-comp*: ‹*sub f* (*sub g p*) = *sub* (*f o g*) *p*›

by (*induct p*) *simp-all*

lemma *sub-list-comp*: ‹*sub-list f* (*sub-list g ps*) = *sub-list* (*f o g*) *ps*›
  using *sub-comp* **by** (*induct ps*) *auto*

lemma *sub-block-comp*: ‹*sub-block f* (*sub-block g block*) = *sub-block* (*f o g*) *block*›
  using *sub-list-comp* **by** (*induct block*) *simp-all*

lemma *sub-branch-comp*:
  ‹*sub-branch f* (*sub-branch g branch*) = *sub-branch* (*f o g*) *branch*›
  unfolding *sub-branch-def* **using** *sub-block-comp* **by** (*induct branch*) *fastforce+*

lemma *swap-id*: ‹(*id*(*i* := *j*, *j* := *i*)) *o* (*id*(*i* := *j*, *j* := *i*)) = *id*›
  by *auto*

lemma *at-in-sub-branch*:
  assumes ‹*p at i in* (*ps*, *a*) # *branch*›
  shows ‹*sub f p at f i in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
  using *assms sub-branch-mem* **by** *fastforce*

lemma *sub-still-allowed*:
  assumes ‹∀ *i*. *p* = *Nom i* ∨ *p* = (◊ *Nom i*) ⟶ *i* ∈ *A*›
  shows ‹*sub f p* = *Nom i* ∨ *sub f p* = (◊ *Nom i*) ⟶ *i* ∈ *f* ‘ *A*›
proof *safe*
  assume ‹*sub f p* = *Nom i*›
  then obtain *i'* **where** *i'*: ‹*p* = *Nom i'*› ‹*f i'* = *i*›
    by (*cases p*) *simp-all*
  then have ‹*i'* ∈ *A*›
    using *assms* **by** *fast*
  then show ‹*i* ∈ *f* ‘ *A*›
    using *i'* **by** *fast*
next
  assume ‹*sub f p* = (◊ *Nom i*)›
  then obtain *i'* **where** *i'*: ‹*p* = (◊ *Nom i'*)› ‹*f i'* = *i*›
  proof (*induct p*)
    case (*Dia q*)
    then show *?case*
      by (*cases q*) *simp-all*
  qed *simp-all*
  then have ‹*i'* ∈ *A*›
    using *assms* **by** *fast*
  then show ‹*i* ∈ *f* ‘ *A*›
    using *i'* **by** *fast*
qed

If a branch has a closing tableau then so does any branch obtained by renaming nominals as long as the substitution leaves some nominals free. This is always the case for substitutions that do not change the type of nominals. Since some formulas on the renamed branch may no longer be new, they do not contribute any potential and so we existentially quantify over the potential needed to close the new branch. We assume that the set of allowed nominals A is finite such that we can obtain a free nominal.

lemma *STA-sub'*:
  fixes *f* :: ‹′b ⇒ ′c›
  assumes ‹⋀(*f* :: ′b ⇒ ′c) *i A*. *finite A* ⟹ *i* ∉ *A* ⟹ ∃ *j*. *j* ∉ *f* ‘ *A*›
    ‹*finite A*› ‹*A*, *n* ⊢ *branch*›
  shows ‹*f* ‘ *A* ⊢ *sub-branch f branch*›
  using *assms*(*3*−)
proof (*induct n branch arbitrary*: *f rule*: *STA.induct*)
  case (*Close p i branch n*)
  have ‹*sub f p at f i in sub-branch f branch*›
    using *Close*(*1*) *sub-branch-mem* **by** *fastforce*
  moreover have ‹(¬ *sub f p*) *at f i in sub-branch f branch*›
    using *Close*(*2*) *sub-branch-mem* **by** *force*
  ultimately show *?case*
    using *STA.Close* **by** *fast*
next
  case (*Neg p a ps branch n f*)
  then have ‹*f* ‘ *A* ⊢ (*sub f p* # *sub-list f ps*, *f a*) # *sub-branch f branch*›
    unfolding *sub-branch-def* **by** *simp*
  moreover have ‹(¬ ¬ *sub f p*) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
   using *Neg*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*3*))
  ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
    using *Neg'* **by** *fast*
  then show *?case*
    unfolding *sub-branch-def* **by** *simp*
next
  case (*DisP p q a ps branch n*)
  then have
    ‹*f* ‘ *A* ⊢ (*sub f p* # *sub-list f ps*, *f a*) # *sub-branch f branch*›
    ‹*f* ‘ *A* ⊢ (*sub f q* # *sub-list f ps*, *f a*) # *sub-branch f branch*›
    unfolding *sub-branch-def* **by** *simp-all*
  moreover have ‹(*sub f p* ∨ *sub f q*) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
   using *DisP*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*4*))
  ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
    using *DisP''* **by** *fast*
  then show *?case*
    unfolding *sub-branch-def* **by** *simp*
next
  case (*DisN p q a ps branch n*)
  then have ‹*f* ‘ *A* ⊢ ((¬ *sub f q*) # (¬ *sub f p*) # *sub-list f ps*, *f a*) # *sub-branch f branch*›
    unfolding *sub-branch-def* **by** *simp*
  moreover have ‹(¬ (*sub f p* ∨ *sub f q*)) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
   using *DisN*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*3*−*4*))
  ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›

using *DisN′* **by** *fast*
then show *?case*
unfolding *sub-branch-def* **by** *simp*
next
case (*DiaP p a ps branch i n*)
have ‹$i \notin A$›
using *DiaP*(*2*) **by** *simp*

show *?case*
proof (*cases* ‹*witnessed* (*sub f p*) (*f a*) (*sub-branch f* ((*ps*, *a*) # *branch*))›)
case *True*
then obtain i' **where**
rs: ‹(**@** i' (*sub f p*)) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*› **and**
ts: ‹(◇ *Nom* i') *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
unfolding *sub-branch-def witnessed-def* **by** *auto*
from *rs* **have** *rs′*:
‹(**@** i' (*sub f p*)) *at f a in* ((◇ *Nom* i') # *sub-list f ps*, *f a*) # *sub-branch f branch*›
by *fastforce*

let *?f* = ‹*f*($i := i'$)›
let *?branch* = ‹*sub-branch ?f branch*›
have ‹*sub-branch ?f* ((*ps*, *a*) # *branch*) = *sub-branch f* ((*ps*, *a*) # *branch*)›
using *DiaP*(*2*) *sub-branch-upd-fresh* **by** *fast*
then have ∗∗: ‹*sub-list ?f ps* = *sub-list f ps*› ‹*?f a* = *f a*› ‹*?branch* = *sub-branch f branch*›
unfolding *sub-branch-def* **by** *simp-all*

have *p*: ‹*sub ?f p* = *sub f p*›
using *DiaP*(*1*−*2*) *sub-upd-fresh* **unfolding** *branch-nominals-def* **by** *fastforce*

have ‹*?f* ‘ *A* ⊢ *sub-branch ?f* (((**@** *i p*) # (◇ *Nom i*) # *ps*, *a*) # *branch*)›
using *DiaP*(*6*) **by** *blast*
then have
‹*?f* ‘ *A* ⊢ ((**@** (*?f i*) (*sub ?f p*)) # (◇ *Nom* (*?f i*)) # *sub-list ?f ps*, *?f a*) # *?branch*›
unfolding *sub-branch-def* **by** *fastforce*
then have
‹*?f* ‘ *A* ⊢ ((**@** i' (*sub f p*)) # (◇ *Nom* i') # *sub-list f ps*, *f a*) # *sub-branch f branch*›
using *p* ∗∗ **by** *simp*
then have ‹*?f* ‘ *A* ⊢ ((◇ *Nom* i') # *sub-list f ps*, *f a*) # *sub-branch f branch*›
using *rs′* **by** (*meson Dup new-def*)
then have ‹*?f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *ts* **by** (*meson Dup new-def*)
moreover have ‹*?f* ‘ *A* = *f* ‘ *A*›
using ‹$i \notin A$› **by** *auto*
ultimately show *?thesis*
unfolding *sub-branch-def* **by** *auto*

next
  case *False*
  have ‹*finite* (*branch-nominals* ((*ps*, *a*) # *branch*))›
    by (*simp add*: *finite-branch-nominals*)
  then have ‹*finite* (*A* ∪ *branch-nominals* ((*ps*, *a*) # *branch*))›
    using ‹*finite A*› **by** *simp*
  then obtain *j* **where** ∗: ‹*j* ∉ *f* ‘ (*A* ∪ *branch-nominals* ((*ps*, *a*) # *branch*))›
    using *DiaP*(*2*) *assms* **by** *metis*
  then have ‹*j* ∉ *f* ‘ *A*›
    by *blast*

  let *?f* = ‹*f*(*i* := *j*)›
  let *?branch* = ‹*sub-branch ?f branch*›
  have ∗∗: ‹*sub-branch ?f* ((*ps*, *a*) # *branch*) = *sub-branch f* ((*ps*, *a*) # *branch*)›
    using *DiaP*(*2*) *sub-branch-upd-fresh* **by** *fast*
  then have ∗∗∗: ‹*sub-list ?f ps* = *sub-list f ps*› ‹*?f a* = *f a*› ‹*?branch* = *sub-branch f branch*›
    unfolding *sub-branch-def* **by** *simp-all*
  moreover have *p*: ‹*sub ?f p* = *sub f p*›
    using *DiaP*(*1*−*2*) *sub-upd-fresh* **unfolding** *branch-nominals-def* **by** *fastforce*
  ultimately have ‹¬ *witnessed* (*sub ?f p*) (*?f a*) (*sub-branch ?f* ((*ps*, *a*) # *branch*))›
    using *False* ∗∗ **by** *simp*
  then have *w*: ‹¬ *witnessed* (*sub ?f p*) (*?f a*) ((*sub-list ?f ps*, *?f a*) # *?branch*)›
    unfolding *sub-branch-def* **by** *simp*

  have *f*: ‹*?f* ‘ *A* = *f* ‘ *A*›
    using ‹*i* ∉ *A*› **by** *auto*

  have ‹*?f* ‘ *A* ⊢ *sub-branch ?f* (((**@** *i p*) # (◇ *Nom i*) # *ps*, *a*) # *branch*)›
    using *DiaP*(*6*) **by** *blast*
  then have ‹*f* ‘ *A* ⊢ ((**@** (*?f i*) (*sub ?f p*)) # (◇ *Nom* (*?f i*)) # *sub-list ?f ps*, *?f a*) # *?branch*›
    unfolding *sub-branch-def* **using** *f* **by** *simp*
  moreover have ‹*sub ?f* (◇ *p*) *at ?f a in* (*sub-list ?f ps*, *?f a*) # *sub-branch ?f branch*›
    using *DiaP*(*1*) *at-in-sub-branch* **by** *fast*
  then have ‹(◇ *sub ?f p*) *at ?f a in* (*sub-list ?f ps*, *?f a*) # *sub-branch ?f branch*›
    by *simp*
  moreover have ‹∄ *a*. *sub ?f p* = *Nom a*›
    using *DiaP*(*3*) **by** (*cases p*) *simp-all*
  moreover have ‹*j* ∉ *f* ‘ (*branch-nominals* ((*ps*, *a*) # *branch*))›
    using ∗ **by** *blast*
  then have ‹*?f i* ∉ *branch-nominals* ((*sub-list ?f ps*, *?f a*) # *?branch*)›
    using ∗∗ *sub-branch-nominals* **unfolding** *sub-branch-def*
    by (*metis fun-upd-same list*.*simps*(*9*) *sub-block*.*simps*)
  ultimately have ‹*f* ‘ *A* ⊢ (*sub-list ?f ps*, *?f a*) # *?branch*›
    using *w DiaP′* ‹*j* ∉ *f* ‘ *A*› **by** (*metis Un-iff fun-upd-same*)
  then show *?thesis*

using ∗∗∗ **unfolding** *sub-branch-def* **by** *simp*
qed
next
case (*DiaN p a ps branch i n*)
then have ‹*f* ‘ *A* ⊢ ((¬ (**@** (*f i*) (*sub f p*))) # *sub-list f ps*, *f a*) # *sub-branch f branch*›
unfolding *sub-branch-def* **by** *simp*
moreover have ‹(¬ (◇ *sub f p*)) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *DiaN*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*3*, *5*))
moreover have ‹(◇ *Nom* (*f i*)) *at f a in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *DiaN*(*2*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*2*, *5*))
ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *DiaN′* **by** *fast*
then show *?case*
unfolding *sub-branch-def* **by** *simp*
next
case (*SatP a p b ps branch n*)
then have ‹*f* ‘ *A* ⊢ (*sub f p* # *sub-list f ps*, *f a*) # *sub-branch f branch*›
unfolding *sub-branch-def* **by** *simp*
moreover have ‹(**@** (*f a*) (*sub f p*)) *at f b in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *SatP*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*6*))
ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *SatP′* **by** *fast*
then show *?case*
unfolding *sub-branch-def* **by** *simp*
next
case (*SatN a p b ps branch n*)
then have ‹*f* ‘ *A* ⊢ ((¬ *sub f p*) # *sub-list f ps*, *f a*) # *sub-branch f branch*›
unfolding *sub-branch-def* **by** *simp*
moreover have ‹(¬ (**@** (*f a*) (*sub f p*))) *at f b in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *SatN*(*1*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*3*, *6*))
ultimately have ‹*f* ‘ *A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
using *SatN′* **by** *fast*
then show *?case*
unfolding *sub-branch-def* **by** *simp*
next
case (*GoTo i branch n*)
then have ‹*f* ‘ *A* ⊢ ([], *f i*) # *sub-branch f branch*›
unfolding *sub-branch-def* **by** *simp*
moreover have ‹*f i* ∈ *branch-nominals* (*sub-branch f branch*)›
using *GoTo*(*1*) *sub-branch-nominals* **by** *fast*
ultimately show *?case*
using *STA.GoTo* **by** *fast*

next
  case (*Nom p b ps a branch n*)
  then have ‹*f ‘ A* ⊢ *sub-branch f* ((*p* # *ps*, *a*) # *branch*)›
    by *blast*
  then have ‹*f ‘ A* ⊢ (*sub f p* # *sub-list f ps*, *f a*) # *sub-branch f branch*›
    unfolding *sub-branch-def* **by** *simp*
  moreover have ‹*sub f p at f b in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
    using *Nom*(*1*) *at-in-sub-branch* **by** *fast*
  moreover have ‹*Nom* (*f a*) *at f b in* (*sub-list f ps*, *f a*) # *sub-branch f branch*›
   using *Nom*(*2*) *at-in-sub-branch* **by** (*metis* (*no-types*, *opaque-lifting*) *sub.simps*(*2*))
  moreover have ‹∀ *i*. *sub f p* = *Nom i* ∨ *sub f p* = (◇ *Nom i*) ⟶ *i* ∈ *f ‘ A*›
    using *Nom*(*3*) *sub-still-allowed* **by** *metis*
  ultimately have ‹*f ‘ A* ⊢ (*sub-list f ps*, *f a*) # *sub-branch f branch*›
    using *Nom′* **by** *metis*
  then show *?case*
    unfolding *sub-branch-def* **by** *simp*
qed

lemma *ex-fresh-gt*:
  fixes *f* :: ‹′*b* ⇒ ′*c*›
  assumes ‹∃ *g* :: ′*c* ⇒ ′*b*. *surj g*› ‹*finite A*› ‹*i* ∉ *A*›
  shows ‹∃ *j*. *j* ∉ *f ‘ A*›
proof (*rule ccontr*)
  assume ‹∄ *j*. *j* ∉ *f ‘ A*›
  moreover obtain *g* :: ‹′*c* ⇒ ′*b*› **where** ‹*surj g*›
    using *assms*(*1*) **by** *blast*
  ultimately show *False*
    using *assms*(*2*−*3*)
    by (*metis UNIV-I UNIV-eq-I card-image-le card-seteq finite-imageI image-comp subsetI*)
qed

corollary *STA-sub-gt*:
  fixes *f* :: ‹′*b* ⇒ ′*c*›
  assumes ‹∃ *g* :: ′*c* ⇒ ′*b*. *surj g*› ‹*A* ⊢ *branch*›
    ‹*finite A*› ‹∀ *i* ∈ *branch-nominals branch*. *f i* ∈ *f ‘ A* ⟶ *i* ∈ *A*›
  shows ‹*f ‘ A* ⊢ *sub-branch f branch*›
  using *assms ex-fresh-gt STA-sub′* **by** *metis*

corollary *STA-sub-inf*:
  fixes *f* :: ‹′*b* ⇒ ′*c*›
  assumes ‹*infinite* (*UNIV* :: ′*c set*)› ‹*A* ⊢ *branch*›
    ‹*finite A*› ‹∀ *i* ∈ *branch-nominals branch*. *f i* ∈ *f ‘ A* ⟶ *i* ∈ *A*›
  shows ‹*f ‘ A* ⊢ *sub-branch f branch*›
proof −
  have ‹*finite A* ⟹ ∃ *j*. *j* ∉ *f ‘ A*› **for** *A* **and** *f* :: ‹′*b* ⇒ ′*c*›
    using *assms*(*1*) *ex-new-if-finite* **by** *blast*
  then show *?thesis*
    using *assms*(*2*−) *STA-sub′* **by** *metis*

qed

corollary *STA-sub*:
  fixes f :: ‹′*b* ⇒ ′*b*›
  assumes ‹*A* ⊢ *branch*› ‹*finite A*›
  shows ‹*f* ‘ *A* ⊢ *sub-branch f branch*›
proof −
  have ‹*finite A* ⟹ *i* ∉ *A* ⟹ ∃ *j*. *j* ∉ *f* ‘ *A*› **for** *i A* **and** *f* :: ‹′*b* ⇒ ′*b*›
    by (*metis card-image-le card-seteq finite-imageI subsetI*)
  then show *?thesis*
    using *assms STA-sub′* **by** *metis*
qed

8.1 Unrestricted (◇) rule

lemma *DiaP′′*:
  assumes
    ‹(◇ *p*) *at a in* (*ps*, *a*) # *branch*›
    ‹*i* ∉ *A* ∪ *branch-nominals* ((*ps*, *a*) # *branch*)› ‹∄ *a*. *p* = *Nom a*›
    ‹*finite A*›
    ‹*A* ⊢ ((**@** *i p*) # (◇ *Nom i*) # *ps*, *a*) # *branch*›
  shows ‹*A* ⊢ (*ps*, *a*) # *branch*›
proof (*cases* ‹*witnessed p a* ((*ps*, *a*) # *branch*)›)
  case *True*
  then obtain *i′* **where**
    rs: ‹(**@** *i′ p*) *at a in* (*ps*, *a*) # *branch*› **and**
    ts: ‹(◇ *Nom i′*) *at a in* (*ps*, *a*) # *branch*›
    unfolding *witnessed-def* **by** *blast*
  then have *rs′*:
    ‹(**@** *i′ p*) *at a in* ((◇ *Nom i′*) # *ps*, *a*) # *branch*›
    by *fastforce*

  let *?f* = ‹*id*(*i* := *i′*)›

  have ‹*?f* ‘ *A* ⊢ *sub-branch ?f* (((**@** *i p*) # (◇ *Nom i*) # *ps*, *a*) # *branch*)›
    using *assms*(*4*−*5*) *STA-sub* **by** *blast*
  then have ‹*?f* ‘ *A* ⊢ ((**@** *i′* (*sub ?f p*)) # (◇ *Nom i′*) # *sub-list ?f ps*, *?f a*) #
      sub-branch ?f branch›
    unfolding *sub-branch-def* **by** *simp*
  moreover have ‹*i* ∉ *nominals p*› ‹*i* ∉ *list-nominals ps*› ‹*i* ≠ *a*› ‹*i* ∉ *branch-nominals branch*›
    using *assms*(*1*−*3*) **unfolding** *branch-nominals-def* **by** *fastforce+*
  then have ‹*sub ?f p* = *p*›
    by (*simp add*: *sub-id sub-upd-fresh*)
  moreover have ‹*sub-list ?f ps* = *ps*›
    using ‹*i* ∉ *list-nominals ps*› **by** (*simp add*: *map-idI sub-id sub-upd-fresh*)
  moreover have ‹*?f a* = *a*›
    using ‹*i* ≠ *a*› **by** *simp*
  moreover have ‹*sub-branch ?f branch* = *branch*›

using ‹$i \notin$ *branch-nominals branch*› **by** (*simp add*: *sub-branch-id sub-branch-upd-fresh*)
ultimately have ‹*?f* ‘ $A \vdash$ ((**@** i' p) # (◊ *Nom* i') # *ps*, *a*) # *branch*›
by *simp*
then have ‹*?f* ‘ $A \vdash$ ((◊ *Nom* i') # *ps*, *a*) # *branch*›
using rs' **by** (*meson Dup new-def*)
then have ‹*?f* ‘ $A \vdash$ (*ps*, *a*) # *branch*›
using *ts* **by** (*meson Dup new-def*)
moreover have ‹*?f* ‘ $A = A$›
using *assms*(*2*) **by** *auto*
ultimately show *?thesis*
by *simp*
next
case *False*
then show *?thesis*
using *assms* $DiaP'$ *STA-Suc* **by** *fast*
qed

9 Structural Properties

lemma *block-nominals-branch*:
assumes ‹*block* ∈. *branch*›
shows ‹*block-nominals block* ⊆ *branch-nominals branch*›
unfolding *branch-nominals-def* **using** *assms* **by** *blast*

lemma *sub-block-fresh*:
assumes ‹$i \notin$ *branch-nominals branch*› ‹*block* ∈. *branch*›
shows ‹*sub-block* (f(i := j)) *block* = *sub-block f block*›
using *assms block-nominals-branch sub-block-upd-fresh* **by** *fast*

lemma *list-down-induct* [*consumes 1*, *case-names Start Cons*]:
assumes ‹$\forall y \in$ *set ys*. *Q y*› ‹*P* (*ys* @ *xs*)›
‹$\bigwedge$*y xs*. *Q y* $\Longrightarrow$ *P* (*y* # *xs*) $\Longrightarrow$ *P xs*›
shows ‹*P xs*›
using *assms* **by** (*induct ys*) *auto*

If the last block on a branch has opening nominal *a* and the last formulas on that block occur on another block alongside nominal *a*, then we can drop those formulas.

lemma *STA-drop-prefix*:
assumes ‹*set ps* ⊆ *set qs*› ‹(*qs*, *a*) ∈. *branch*› ‹*A*, $n \vdash$ (*ps* @ ps', *a*) # *branch*›
shows ‹*A*, $n \vdash$ (ps', *a*) # *branch*›
proof −
have ‹$\forall p \in$ *set ps*. *p on* (*qs*, *a*)›
using *assms*(*1*) **by** *auto*
then show *?thesis*
proof (*induct* ps' *rule*: *list-down-induct*)
case *Start*
then show *?case*

```
      using assms(3) .
  next
    case (Cons p ps)
    then show ?case
      using assms(2) by (meson Dup new-def list.set-intros(2))
  qed
qed
```

We can drop a block if it is subsumed by another block.

```
lemma STA-drop-block:
  assumes
    ‹set ps ⊆ set ps'› ‹(ps', a) ∈. branch›
    ‹A, n ⊢ (ps, a) # branch›
  shows ‹A, Suc n ⊢ branch›
  using assms
proof (induct branch)
  case Nil
  then show ?case
    by simp
next
  case (Cons block branch)
  then show ?case
  proof (cases block)
    case (Pair qs b)
    then have ‹A, n ⊢ ([], a) # (qs, b) # branch›
       using Cons(2−4) STA-drop-prefix[where branch=‹(qs, b) # branch›] by
simp
    moreover have ‹a ∈ branch-nominals ((qs, b) # branch)›
      unfolding branch-nominals-def using Cons(3) Pair by fastforce
    ultimately have ‹A, Suc n ⊢ (qs, b) # branch›
      by (simp add: GoTo)
    then show ?thesis
      using Pair Dup by fast
  qed
qed

lemma STA-drop-block':
  assumes ‹A, n ⊢ (ps, a) # branch› ‹(ps, a) ∈. branch›
  shows ‹A, Suc n ⊢ branch›
  using assms STA-drop-block by fastforce

lemma sub-branch-image: ‹set (sub-branch f branch) = sub-block f ‘ set branch›
  unfolding sub-branch-def by simp

lemma sub-block-repl:
  assumes ‹j ∉ block-nominals block›
  shows ‹i ∉ block-nominals (sub-block (id(i := j, j := i)) block)›
  using assms by (simp add: image-iff sub-block-nominals)
```

```
lemma sub-branch-repl:
  assumes ‹j ∉ branch-nominals branch›
  shows ‹i ∉ branch-nominals (sub-branch (id(i := j, j := i)) branch)›
  using assms by (simp add: image-iff sub-branch-nominals)
```

If a finite set of blocks has a closing tableau then so does any finite superset.

```
lemma STA-struct:
  fixes branch :: ‹('a, 'b) branch›
  assumes
    inf: ‹infinite (UNIV :: 'b set)› and fin: ‹finite A› and
    ‹A, n ⊢ branch› ‹set branch ⊆ set branch'›
  shows ‹A ⊢ branch'›
  using assms(3−)
proof (induct n branch arbitrary: branch' rule: STA.induct)
  case (Close p i branch n)
  then show ?case
    using STA.Close by fast
next
  case (Neg p a ps branch n)
  have ‹A ⊢ (p # ps, a) # branch'›
    using Neg(4−) by (simp add: subset-code(1))
  moreover have ‹(¬ ¬ p) at a in (ps, a) # branch'›
    using Neg(1, 5) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using Neg' by fast
  moreover have ‹(ps, a) ∈. branch'›
    using Neg(5) by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (DisP p q a ps branch n)
  have ‹A ⊢ (p # ps, a) # branch'› ‹A ⊢ (q # ps, a) # branch'›
    using DisP(5, 7−) by (simp-all add: subset-code(1))
  moreover have ‹(p ∨ q) at a in (ps, a) # branch'›
    using DisP(1, 8) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using DisP'' by fast
  moreover have ‹(ps, a) ∈. branch'›
    using DisP(8) by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (DisN p q a ps branch n)
  have ‹A ⊢ ((¬ q) # (¬ p) # ps, a)# branch'›
    using DisN(4−) by (simp add: subset-code(1))
  moreover have ‹(¬ (p ∨ q)) at a in (ps, a) # branch'›
    using DisN(1, 5) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using DisN' by fast
```

moreover have ‹$(ps, a) \in.$ *branch'*›
  using *DisN(5)* **by** *simp*
ultimately show *?case*
  using *STA-drop-block'* **by** *fast*
next
  case (*DiaP p a ps branch i n*)
  have ‹*finite* ($A \cup$ *branch-nominals branch'*)›
    using *fin* **by** (*simp add*: *finite-branch-nominals*)
  then obtain *j* **where** *j*: ‹$j \notin A \cup$ *branch-nominals branch'*›
    using *assms ex-new-if-finite* **by** *blast*
  then have *j'*: ‹$j \notin$ *branch-nominals* ((*ps*, *a*) # *branch*)›
    using *DiaP(7)* **unfolding** *branch-nominals-def* **by** *blast*

  let *?f* = ‹*id*($i := j$, $j := i$)›
  let *?branch'* = ‹*sub-branch ?f branch'*›
  have *branch'*: ‹*sub-branch ?f ?branch'* = *branch'*›
    using *sub-branch-comp sub-branch-id swap-id* **by** *metis*

  have ‹$i \notin$ *branch-nominals* ((*ps*, *a*) # *branch*)›
    using *DiaP(2)* **by** *blast*
  then have *branch*: ‹*sub-branch ?f* ((*ps*, *a*) # *branch*) = (*ps*, *a*) # *branch*›
    using *DiaP(2) j' sub-branch-id sub-branch-upd-fresh* **by** *metis*
  moreover have
    ‹*set* (*sub-branch ?f* ((*ps*, *a*) # *branch*)) $\subseteq$ *set ?branch'*›
    using *DiaP(7) sub-branch-image* **by** *blast*
  ultimately have $*$: ‹*set* ((*ps*, *a*) # *branch*) $\subseteq$ *set ?branch'*›
    unfolding *sub-branch-def* **by** *auto*

  have ‹$i \notin$ *block-nominals* (*ps*, *a*)›
    using *DiaP* **unfolding** *branch-nominals-def* **by** *simp*
  moreover have ‹$i \notin$ *branch-nominals ?branch'*›
    using *j sub-branch-repl* **by** *fast*
  ultimately have *i*: ‹$i \notin$ *branch-nominals* ((*ps*, *a*) # *?branch'*)›
    unfolding *branch-nominals-def* **by** *simp*

  have ‹*?f* ‘ *A* = *A*›
    using *DiaP(2) j* **by** *auto*

  have ‹$A \vdash$ ((**@** *i p*) # ($\Diamond$ *Nom i*) # *ps*, *a*) # *?branch'*›
    using *DiaP(6)* $*$
   by (*metis* (*no-types*, *lifting*) *subset-code(1) insert-mono list.set(2) set-subset-Cons*)
  moreover have ‹($\Diamond$ *p*) *at a in* (*ps*, *a*) # *?branch'*›
    using *DiaP(1, 7)* $*$ **by** (*meson set-subset-Cons subset-code(1)*)
  ultimately have ‹$A \vdash$ (*ps*, *a*) # *?branch'*›
    using *inf DiaP(2−3) fin i DiaP''* **by** (*metis Un-iff*)
  then have ‹*?f* ‘ $A \vdash$ *sub-branch ?f* ((*ps*, *a*) # *?branch'*)›
    using *STA-sub fin* **by** *blast*
  then have ‹$A \vdash$ (*ps*, *a*) # *branch'*›
    using ‹*?f* ‘ *A* = *A*› *branch' branch* **unfolding** *sub-branch-def* **by** *simp*

```
  moreover have ‹(ps, a) ∈. branch'›
    using ‹set ((ps, a) # branch) ⊆ set branch'› by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (DiaN p a ps branch i n)
  have ‹A ⊢ ((¬ (@ i p)) # ps, a) # branch'›
    using DiaN(5−) by (simp add: subset-code(1))
  moreover have
    ‹(¬ (◇ p)) at a in (ps, a) # branch'›
    ‹(◇ Nom i) at a in (ps, a) # branch'›
    using DiaN(1−2, 6) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using DiaN' by fast
  moreover have ‹(ps, a) ∈. branch'›
    using DiaN(6) by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (SatP a p b ps branch n)
  have ‹A ⊢ (p # ps, a) # branch'›
    using SatP(4−) by (simp add: subset-code(1))
  moreover have ‹(@ a p) at b in (ps, a) # branch'›
    using SatP(1, 5) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using SatP' by fast
  moreover have ‹(ps, a) ∈. branch'›
    using SatP(5) by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (SatN a p b ps branch n)
  have ‹A ⊢ ((¬ p) # ps, a) # branch'›
    using SatN(4−) by (simp add: subset-code(1))
  moreover have ‹(¬ (@ a p)) at b in (ps, a) # branch'›
    using SatN(1, 5) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using SatN' by fast
  moreover have ‹(ps, a) ∈. branch'›
    using SatN(5) by simp
  ultimately show ?case
    using STA-drop-block' by fast
next
  case (GoTo i branch n)
  then have ‹A ⊢ ([], i) # branch'›
    by (simp add: subset-code(1))
  moreover have ‹i ∈ branch-nominals branch'›
    using GoTo(1, 4) unfolding branch-nominals-def by auto
  ultimately show ?case
```

```
    using GoTo(2) STA.GoTo by fast
next
  case (Nom p b ps a branch n)
  have ‹A ⊢ (p # ps, a) # branch'›
    using Nom(6−) by (simp add: subset-code(1))
  moreover have ‹p at b in (ps, a) # branch'›
    using Nom(1, 7) by auto
  moreover have ‹Nom a at b in (ps, a) # branch'›
    using Nom(2, 7) by auto
  ultimately have ‹A ⊢ (ps, a) # branch'›
    using Nom(3) Nom' by metis
  moreover have ‹(ps, a) ∈. branch'›
    using Nom(7) by simp
  ultimately show ?case
    using STA-drop-block' by fast
qed
```

If a branch has a closing tableau then we can replace the formulas of the last block on that branch with any finite superset and still obtain a closing tableau.

```
lemma STA-struct-block:
  fixes branch :: ‹('a, 'b) branch›
  assumes
    inf: ‹infinite (UNIV :: 'b set)› and fin: ‹finite A› and
    ‹A, n ⊢ (ps, a) # branch› ‹set ps ⊆ set ps'›
  shows ‹A ⊢ (ps', a) # branch›
  using assms(3−)
proof (induct n ‹(ps, a) # branch› arbitrary: ps ps' rule: STA.induct)
  case (Close p i n ts ts')
  then have ‹p at i in (ts', a) # branch› ‹(¬ p) at i in (ts', a) # branch›
    by auto
  then show ?case
    using STA.Close by fast
next
  case (Neg p ps n)
  then have ‹(¬ ¬ p) at a in (ps', a) # branch›
    by auto
  moreover have ‹A ⊢ (p # ps', a) # branch›
    using Neg(4−) by (simp add: subset-code(1))
  ultimately show ?case
    using Neg' by fast
next
  case (DisP p q ps n)
  then have ‹(p ∨ q) at a in (ps', a) # branch›
    by auto
  moreover have ‹A ⊢ (p # ps', a) # branch› ‹A ⊢ (q # ps', a) # branch›
    using DisP(5, 7−) by (simp-all add: subset-code(1))
  ultimately show ?case
    using DisP'' by fast
```

```
next
  case (DisN p q ps n)
  then have ‹(¬ (p ∨ q)) at a in (ps', a) # branch›
    by auto
  moreover have ‹A ⊢ ((¬ q) # (¬ p) # ps', a) # branch›
    using DisN(4-) by (simp add: subset-code(1))
  ultimately show ?case
    using DisN' by fast
next
  case (DiaP p ps i n)
  have ‹finite (A ∪ branch-nominals ((ps', a) # branch))›
    using fin finite-branch-nominals by blast
  then obtain j where j: ‹j ∉ A ∪ branch-nominals ((ps', a) # branch)›
    using assms ex-new-if-finite by blast
  then have j': ‹j ∉ block-nominals (ps, a)›
    using DiaP.prems unfolding branch-nominals-def by auto

  let ?f = ‹id(i := j, j := i)›
  let ?ps' = ‹sub-list ?f ps'›
  have ps': ‹sub-list ?f ?ps' = ps'›
    using sub-list-comp sub-list-id swap-id by metis

  have ‹i ∉ block-nominals (ps, a)›
    using DiaP(1-2) unfolding branch-nominals-def by simp
  then have ps: ‹sub-block ?f (ps, a) = (ps, a)›
    using j' sub-block-id sub-block-upd-fresh by metis
  moreover have ‹set (sub-list ?f ps) ⊆ set (sub-list ?f ps')›
    using ‹set ps ⊆ set ps'› by auto
  ultimately have *: ‹set ps ⊆ set ?ps'›
    by simp

  have ‹i ∉ branch-nominals branch›
    using DiaP unfolding branch-nominals-def by simp
  moreover have ‹j ∉ branch-nominals branch›
    using j unfolding branch-nominals-def by simp
  ultimately have branch: ‹sub-branch ?f branch = branch›
    using sub-branch-id sub-branch-upd-fresh by metis

  have ‹i ≠ a› ‹j ≠ a›
    using DiaP j unfolding branch-nominals-def by simp-all
  then have ‹?f a = a›
    by simp
  moreover have ‹j ∉ block-nominals (ps', a)›
    using j unfolding branch-nominals-def by simp
  ultimately have ‹i ∉ block-nominals (?ps', a)›
    using sub-block-repl[where block=‹(ps', a)› and i=i and j=j] by simp

  have ‹?f ` A = A›
    using DiaP(2) j by auto
```

```
  have ‹(◇ p) at a in (?ps′, a) # branch›
    using DiaP(1) ∗ by fastforce
  moreover have ‹A ⊢ ((@ i p) # (◇ Nom i) # ?ps′, a) # branch›
    using ∗ DiaP(6) fin by (simp add: subset-code(1))
  moreover have ‹i ∉ A ∪ branch-nominals ((?ps′, a) # branch)›
    using DiaP(2) ‹i ∉ block-nominals (?ps′, a)› unfolding branch-nominals-def
by simp
  ultimately have ‹A ⊢ (?ps′, a) # branch›
    using DiaP(3) fin DiaP′′ by metis
  then have ‹?f ‘ A ⊢ sub-branch ?f ((?ps′, a) # branch)›
    using STA-sub fin by blast
  then have ‹A ⊢ (sub-list ?f ?ps′, ?f a) # sub-branch ?f branch›
    unfolding sub-branch-def using ‹?f ‘ A = A› by simp
  then show ?case
    using ‹?f a = a› ps′ branch by simp
next
  case (DiaN p ps i n)
  then have
    ‹(¬ (◇ p)) at a in (ps′, a) # branch›
    ‹(◇ Nom i) at a in (ps′, a) # branch›
    by auto
  moreover have ‹A ⊢ ((¬ (@ i p)) # ps′, a) # branch›
    using DiaN(5−) by (simp add: subset-code(1))
  ultimately show ?case
    using DiaN′ by fast
next
  case (SatP p b ps n)
  then have ‹(@ a p) at b in (ps′, a) # branch›
    by auto
  moreover have ‹A ⊢ (p # ps′, a) # branch›
    using SatP(4−) by (simp add: subset-code(1))
  ultimately show ?case
    using SatP′ by fast
next
  case (SatN p b ps n)
  then have ‹(¬ (@ a p)) at b in (ps′, a) # branch›
    by auto
  moreover have ‹A ⊢ ((¬ p) # ps′, a) # branch›
    using SatN(4−) by (simp add: subset-code(1))
  ultimately show ?case
    using SatN′ by fast
next
  case (GoTo i n ps)
  then have ‹A, Suc n ⊢ (ps, a) # branch›
    using STA.GoTo by fast
  then obtain m where ‹A, m ⊢ (ps, a) # (ps′, a) # branch›
    using inf fin STA-struct[where branch′=‹(ps, a) # - # -›] by fastforce
  then have ‹A, Suc m ⊢ (ps′, a) # branch›
```

```
    using GoTo(4) by (simp add: STA-drop-block[where a=a])
  then show ?case
    by blast
next
  case (Nom p b ps n)
  have ‹p at b in (ps', a) # branch›
    using Nom(1, 7) by auto
  moreover have ‹Nom a at b in (ps', a) # branch›
    using Nom(2, 7) by auto
  moreover have ‹A ⊢ (p # ps', a) # branch›
    using Nom(6−) by (simp add: subset-code(1))
  ultimately show ?case
    using Nom(3) Nom' by metis
qed
```

10 Bridge

We define a *descendants k i branch* relation on sets of indices. The sets are built on the index of a ◊ *Nom k* on an *i*-block in *branch* and can be extended by indices of formula occurrences that can be thought of as descending from that ◊ *Nom k* by application of either the (¬ ◊) or *Nom* rule.

We show that if we have nominals *j* and *k* on the same block in a closeable branch, then the branch obtained by the following transformation is also closeable: For every index *v*, if the formula at *v* is ◊ *Nom k*, replace it by ◊ *Nom j* and if it is ¬ (**@** *k p*) replace it by ¬ (**@** *j p*). There are no other cases.

From this transformation we can show admissibility of the Bridge rule under the assumption that *j* is an allowed nominal.

10.1 Replacing

```
abbreviation bridge' :: ‹'b ⇒ 'b ⇒ ('a, 'b) fm ⇒ ('a, 'b) fm› where
  ‹bridge' k j p ≡ case p of
    (◊ Nom k') ⇒ (if k = k' then (◊ Nom j) else (◊ Nom k'))
  | (¬ (@ k' q)) ⇒ (if k = k' then (¬ (@ j q)) else (¬ (@ k' q)))
  | p ⇒ p›

abbreviation bridge ::
  ‹'b ⇒ 'b ⇒ (nat × nat) set ⇒ nat ⇒ nat ⇒ ('a, 'b) fm ⇒ ('a, 'b) fm› where
  ‹bridge k j ≡ mapper (bridge' k j)›

lemma bridge-on-Nom:
  ‹Nom i on (ps, a) ⟹ Nom i on (mapi (bridge k j xs v) ps, a)›
  by (induct ps) auto

lemma bridge'-nominals:
```

‹*nominals* (*bridge′ k j p*) ∪ {*k*, *j*} = *nominals p* ∪ {*k*, *j*}›
proof (*induct p*)
case (*Neg p*)
then show *?case* **by** (*cases p*) *auto*
next
case (*Dia p*)
then show *?case* **by** (*cases p*) *auto*
qed *auto*

lemma *bridge-nominals*:
‹*nominals* (*bridge k j xs v v′ p*) ∪ {*k*, *j*} = *nominals p* ∪ {*k*, *j*}›
proof (*cases* ‹(*v*, *v′*) ∈ *xs*›)
case *True*
then have ‹*nominals* (*bridge k j xs v v′ p*) = *nominals* (*bridge′ k j p*)›
by *simp*
then show *?thesis*
using *bridge′-nominals* **by** *metis*
qed *simp*

lemma *bridge-block-nominals*:
‹*block-nominals* (*mapi-block* (*bridge k j xs v*) (*ps*, *a*)) ∪ {*k*, *j*} =
block-nominals (*ps*, *a*) ∪ {*k*, *j*}›
proof (*induct ps*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons p ps*)
have ‹*?case* ⟷
(*nominals* (*bridge k j xs v* (*length ps*) *p*)) ∪
(*block-nominals* (*mapi-block* (*bridge k j xs v*) (*ps*, *a*)) ∪ {*k*, *j*}) =
(*nominals p*) ∪ (*block-nominals* (*ps*, *a*) ∪ {*k*, *j*})›
by *simp*
also have ‹. . . ⟷
(*nominals* (*bridge k j xs v* (*length ps*) *p*) ∪ {*k*, *j*}) ∪
(*block-nominals* (*mapi-block* (*bridge k j xs v*) (*ps*, *a*)) ∪ {*k*, *j*}) =
(*nominals p* ∪ {*k*, *j*}) ∪ (*block-nominals* (*ps*, *a*) ∪ {*k*, *j*})›
by *blast*
moreover have
‹*nominals* (*bridge k j xs v* (*length ps*) *p*) ∪ {*k*, *j*} = *nominals p* ∪ {*k*, *j*}›
using *bridge-nominals* **by** *metis*
moreover note *Cons*
ultimately show *?case*
by *argo*
qed

lemma *bridge-branch-nominals*:
‹*branch-nominals* (*mapi-branch* (*bridge k j xs*) *branch*) ∪ {*k*, *j*} =
branch-nominals branch ∪ {*k*, *j*}›

```
proof (induct branch)
  case Nil
  then show ?case
    unfolding branch-nominals-def mapi-branch-def
    by simp
next
  case (Cons block branch)
  have ‹?case ⟷
    (block-nominals (mapi-block (bridge k j xs (length branch)) block)) ∪
    (branch-nominals (mapi-branch (bridge k j xs) branch) ∪ {k, j}) =
    (block-nominals block) ∪ (branch-nominals branch ∪ {k, j})›
    unfolding branch-nominals-def mapi-branch-def by simp
  also have ‹... ⟷
    (block-nominals (mapi-block (bridge k j xs (length branch)) block) ∪ {k, j}) ∪
    (branch-nominals (mapi-branch (bridge k j xs) branch) ∪ {k, j}) =
    (block-nominals block ∪ {k, j}) ∪ (branch-nominals branch ∪ {k, j})›
    by blast
  moreover have
    ‹block-nominals (mapi-block (bridge k j xs (length branch)) block) ∪ {k, j} =
     block-nominals block ∪ {k, j}›
    using bridge-block-nominals[where ps=‹fst block› and a=‹snd block›] by simp
  ultimately show ?case
    using Cons by argo
qed

lemma at-in-mapi-branch:
  assumes ‹p at a in branch› ‹p ≠ Nom a›
  shows ‹∃ v v'. f v v' p at a in mapi-branch f branch›
  using assms by (meson mapi-branch-mem rev-nth-mapi-block rev-nth-on)

lemma nom-at-in-bridge:
  fixes k j xs
  defines ‹f ≡ bridge k j xs›
  assumes ‹Nom i at a in branch›
  shows ‹Nom i at a in mapi-branch f branch›
proof -
  obtain qs where qs: ‹(qs, a) ∈. branch› ‹Nom i on (qs, a)›
    using assms(2) by blast
  then obtain l where ‹(mapi (f l) qs, a) ∈. mapi-branch f branch›
    using mapi-branch-mem by fast
  moreover have ‹Nom i on (mapi (f l) qs, a)›
    unfolding f-def using qs(2) by (induct qs) auto
  ultimately show ?thesis
    by blast
qed

lemma nominals-mapi-branch-bridge:
  assumes ‹Nom k at j in branch›
  shows ‹branch-nominals (mapi-branch (bridge k j xs) branch) = branch-nominals
```

branch›
proof −
  let *?f* = ‹*bridge k j xs*›
  have ‹*Nom k at j in mapi-branch ?f branch*›
    using *assms nom-at-in-bridge* **by** *fast*
  then have
    ‹*j* ∈ *branch-nominals* (*mapi-branch ?f branch*)›
    ‹*k* ∈ *branch-nominals* (*mapi-branch ?f branch*)›
    unfolding *branch-nominals-def* **by** *fastforce*+
  moreover have ‹*j* ∈ *branch-nominals branch*› ‹*k* ∈ *branch-nominals branch*›
    using *assms* **unfolding** *branch-nominals-def* **by** *fastforce*+
  moreover have
    ‹*branch-nominals* (*mapi-branch ?f branch*) ∪ {*k*, *j*} = *branch-nominals branch*
∪ {*k*, *j*}›
    using *bridge-branch-nominals* **by** *metis*
  ultimately show *?thesis*
    by *blast*
qed

lemma *bridge-proper-dia*:
  assumes ‹∄ *a*. *p* = *Nom a*›
  shows ‹*bridge k j xs v v′* (◇ *p*) = (◇ *p*)›
  using *assms* **by** (*induct p*) *simp-all*

lemma *bridge-compl-cases*:
  fixes *k j xs v v′ w w′ p*
  defines ‹*q* ≡ *bridge k j xs v v′ p*› **and** ‹*q′* ≡ *bridge k j xs w w′* (¬ *p*)›
  shows
    ‹(*q* = (◇ *Nom j*) ∧ *q′* = (¬ (◇ *Nom k*))) ∨
  (∃ *r*. *q* = (¬ (**@** *j r*)) ∧ *q′* = (¬ ¬ (**@** *k r*))) ∨
  (∃ *r*. *q* = (**@** *k r*) ∧ *q′* = (¬ (**@** *j r*))) ∨
    (*q* = *p* ∧ *q′* = (¬ *p*))›
proof (*cases p*)
  case (*Neg p*)
  then show *?thesis*
    by (*cases p*) (*simp-all add*: *q-def q′-def*)
next
  case (*Dia p*)
  then show *?thesis*
    by (*cases p*) (*simp-all add*: *q-def q′-def*)
qed (*simp-all add*: *q-def q′-def*)

10.2 Descendants

inductive *descendants* :: ‹*′b* ⇒ *′b* ⇒ (*′a*, *′b*) *branch* ⇒ (*nat* × *nat*) *set* ⇒ *bool*›
where
  Initial:
  ‹*branch* !. *v* = *Some* (*qs*, *i*) ⟹ *qs* !. *v′* = *Some* (◇ *Nom k*) ⟹
    descendants k i branch {(*v*, *v′*)}›

| *Derived*:
‹*branch* !. v = *Some* (*qs*, *a*) ⟹ *qs* !. v' = *Some* (¬ (**@** *k p*)) ⟹
descendants k i branch xs ⟹ (w, w') ∈ *xs* ⟹
branch !. w = *Some* (*rs*, *a*) ⟹ *rs* !. w' = *Some* (◇ *Nom k*) ⟹
descendants k i branch ({(v, v')} ∪ *xs*)›
| *Copied*:
‹*branch* !. v = *Some* (*qs*, *a*) ⟹ *qs* !. v' = *Some p* ⟹
descendants k i branch xs ⟹ (w, w') ∈ *xs* ⟹
branch !. w = *Some* (*rs*, *b*) ⟹ *rs* !. w' = *Some p* ⟹
Nom a at b in branch ⟹
descendants k i branch ({(v, v')} ∪ *xs*)›

lemma *descendants-initial*:
assumes ‹*descendants k i branch xs*›
shows ‹∃ (v, v') ∈ *xs*. ∃ *ps*.
branch !. v = *Some* (*ps*, *i*) ∧ *ps* !. v' = *Some* (◇ *Nom k*)›
using *assms* **by** (*induct k i branch xs rule*: *descendants.induct*) *simp-all*

lemma *descendants-bounds-fst*:
assumes ‹*descendants k i branch xs*› ‹(v, v') ∈ *xs*›
shows ‹v < *length branch*›
using *assms rev-nth-Some*
by (*induct k i branch xs rule*: *descendants.induct*) *fast+*

lemma *descendants-bounds-snd*:
assumes ‹*descendants k i branch xs*› ‹(v, v') ∈ *xs*› ‹*branch* !. v = *Some* (*ps*, *a*)›
shows ‹v' < *length ps*›
using *assms*
by (*induct k i branch xs rule*: *descendants.induct*) (*auto simp*: *rev-nth-Some*)

lemma *descendants-branch*:
‹*descendants k i branch xs* ⟹ *descendants k i* (*extra* @ *branch*) *xs*›
proof (*induct k i branch xs rule*: *descendants.induct*)
case (*Initial branch v qs i v' k*)
then show *?case*
using *rev-nth-append descendants.Initial* **by** *fast*
next
case (*Derived branch v qs a v' k p i xs w w' rs*)
then have
‹(*extra* @ *branch*) !. v = *Some* (*qs*, *a*)›
‹(*extra* @ *branch*) !. w = *Some* (*rs*, *a*)›
using *rev-nth-append* **by** *fast+*
then show *?case*
using *Derived*(*2*, *4*−*5*, *7*) *descendants.Derived* **by** *fast*
next
case (*Copied branch v qs a v' p k i xs w w' rs b*)
then have
‹(*extra* @ *branch*) !. v = *Some* (*qs*, *a*)›
‹(*extra* @ *branch*) !. w = *Some* (*rs*, *b*)›

```
    using rev-nth-append by fast+
  moreover have ‹Nom a at b in (extra @ branch)›
    using Copied(8) by auto
  ultimately show ?case
    using Copied(2−4, 5−7) descendants.Copied by fast
qed

lemma descendants-block:
  assumes ‹descendants k i ((ps, a) # branch) xs›
  shows ‹descendants k i ((ps' @ ps, a) # branch) xs›
  using assms
proof (induct k i ‹(ps, a) # branch› xs arbitrary: ps a branch rule: descen-
dants.induct)
  case (Initial v qs i v' k)
  have
    ‹((ps' @ ps, a) # branch) !. v = Some (qs, i) ∨
     ((ps' @ ps, a) # branch) !. v = Some (ps' @ qs, i)›
    using Initial(1) by auto
  moreover have
    ‹qs !. v' = Some (◇ Nom k)› ‹(ps' @ qs) !. v' = Some (◇ Nom k)›
    using Initial(2) rev-nth-append by simp-all
  ultimately show ?case
    using descendants.Initial by fast
next
  case (Derived v qs a' v' k p i xs w w' rs)
  have
    ‹((ps' @ ps, a) # branch) !. v = Some (qs, a') ∨
     ((ps' @ ps, a) # branch) !. v = Some (ps' @ qs, a')›
    using Derived(1) by auto
  moreover have
    ‹qs !. v' = Some (¬ (@ k p))› ‹(ps' @ qs) !. v' = Some (¬ (@ k p))›
    using Derived(2) rev-nth-append by simp-all
  moreover have
    ‹((ps' @ ps, a) # branch) !. w = Some (rs, a') ∨
     ((ps' @ ps, a) # branch) !. w = Some (ps' @ rs, a')›
    using ‹((ps, a) # branch) !. w = Some (rs, a')› by auto
  moreover have
    ‹rs !. w' = Some (◇ Nom k)› ‹(ps' @ rs) !. w' = Some (◇ Nom k)›
    using Derived(7) rev-nth-append by simp-all
  ultimately show ?case
    using Derived(4−5) descendants.Derived by fast
next
  case (Copied v qs a' v' p k i xs w w' rs b)
  have
    ‹((ps' @ ps, a) # branch) !. v = Some (qs, a') ∨
     ((ps' @ ps, a) # branch) !. v = Some (ps' @ qs, a')›
    using Copied(1) by auto
  moreover have ‹qs !. v' = Some p› ‹(ps' @ qs) !. v' = Some p›
    using Copied(2) rev-nth-append by simp-all
```

moreover have
‹((ps′ @ ps, a) # branch) !. w = Some (rs, b) ∨
((ps′ @ ps, a) # branch) !. w = Some (ps′ @ rs, b)›
using *Copied(6)* **by** *auto*
moreover have ‹rs !. w′ = Some p› ‹(ps′ @ rs) !. w′ = Some p›
using *Copied(7) rev-nth-append* **by** *simp-all*
moreover have
‹((ps′ @ ps, a) # branch) !. w = Some (rs, b) ∨
((ps′ @ ps, a) # branch) !. w = Some (ps′ @ rs, b)›
using *Copied(6)* **by** *auto*
moreover have ‹rs !. w′ = Some p› ‹(ps′ @ rs) !. w′ = Some p›
using *Copied(7) rev-nth-append* **by** *simp-all*
moreover have ‹Nom a′ at b in (ps′ @ ps, a) # branch›
using *Copied(8)* **by** *fastforce*
ultimately show *?case*
using *Copied(4−5) descendants.Copied*[**where** *branch*=‹(ps′ @ ps, a) # branch›]
by *blast*
qed

lemma *descendants-no-head*:
assumes ‹descendants k i ((ps, a) # branch) xs›
shows ‹descendants k i ((p # ps, a) # branch) xs›
using *assms descendants-block*[**where** *ps′*=‹[-]›] **by** *simp*

lemma *descendants-types*:
assumes
‹descendants k i branch xs› ‹(v, v′) ∈ xs›
‹branch !. v = Some (ps, a)› ‹ps !. v′ = Some p›
shows ‹p = (◊ Nom k) ∨ (∃ q. p = (¬ (@ k q)))›
using *assms* **by** (*induct k i branch xs arbitrary*: *v v′ ps a*) *fastforce+*

lemma *descendants-oob-head′*:
assumes ‹descendants k i ((ps, a) # branch) xs›
shows ‹(length branch, m + length ps) ∉ xs›
using *assms descendants-bounds-snd* **by** *fastforce*

lemma *descendants-oob-head*:
assumes ‹descendants k i ((ps, a) # branch) xs›
shows ‹(length branch, length ps) ∉ xs›
using *assms descendants-oob-head′*[**where** *m=0*] **by** *fastforce*

10.3 Induction

We induct over an arbitrary set of indices. That way, we can determine in each case whether the extension gets replaced or not by manipulating the set before applying the induction hypothesis.

lemma *STA-bridge′*:
fixes *a* :: ′*b*
assumes

```
    inf: ‹infinite (UNIV :: 'b set)› and fin: ‹finite A› and ‹j ∈ A›
    ‹A, n ⊢ (ps, a) # branch›
    ‹descendants k i ((ps, a) # branch) xs›
    ‹Nom k at j in branch›
  shows ‹A ⊢ mapi-branch (bridge k j xs) ((ps, a) # branch)›
  using assms(4−)
proof (induct n ‹(ps, a) # branch› arbitrary: ps a branch xs rule: STA.induct)
  case (Close p i' n)
  let ?f = ‹bridge k j xs›
  let ?branch = ‹mapi-branch ?f ((ps, a) # branch)›

  obtain qs where qs: ‹(qs, i') ∈. (ps, a) # branch› ‹p on (qs, i')›
    using Close(1) by blast
  obtain rs where rs: ‹(rs, i') ∈. (ps, a) # branch› ‹(¬ p) on (rs, i')›
    using Close(2) by blast

  obtain v where v: ‹(mapi (?f v) qs, i') ∈. ?branch›
    using qs mapi-branch-mem by fast
  obtain w where w: ‹(mapi (?f w) rs, i') ∈. ?branch›
    using rs mapi-branch-mem by fast

  have k: ‹Nom k at j in ?branch›
    using Close(4) nom-at-in-bridge unfolding mapi-branch-def by fastforce

  show ?case
  proof (cases ‹∃ a. p = Nom a›)
    case True
    then have ‹p on (mapi (?f v) qs, i')›
      using qs bridge-on-Nom by fast
    moreover have ‹(¬ p) on (mapi (?f w) rs, i')›
      using rs(2) True by (induct rs) auto
    ultimately show ?thesis
      using v w STA.Close by fast
  next
    case False
    then obtain v' where ‹qs !. v' = Some p›
      using qs rev-nth-on by fast
    then have qs': ‹(?f v v' p) on (mapi (?f v) qs, i')›
      using rev-nth-mapi-block by fast

    then obtain w' where ‹rs !. w' = Some (¬ p)›
      using rs rev-nth-on by fast
    then have rs': ‹(?f w w' (¬ p)) on (mapi (?f w) rs, i')›
      using rev-nth-mapi-block by fast

    obtain q q' where q: ‹?f v v' p = q› and q': ‹?f w w' (¬ p) = q'›
      by simp-all
    then consider
      (dia) ‹q = (◇ Nom j)› ‹q' = (¬ (◇ Nom k))› |
```

(*satn*)‹∃ *r*. *q* = (¬ (@ *j r*)) ∧ *q′* = (¬ ¬ (@ *k r*))› |
(*sat*) ‹∃ *r*. *q* = (@ *k r*) ∧ *q′* = (¬ (@ *j r*))› |
(*old*) ‹*q* = *p*› ‹*q′* = (¬ *p*)›
using *bridge-compl-cases* **by** *fast*
then show *?thesis*
proof *cases*
case *dia*
then have ∗:
‹(◊ *Nom j*) *on* (*mapi* (*?f v*) *qs*, *i′*)›
‹(¬ (◊ *Nom k*)) *on* (*mapi* (*?f w*) *rs*, *i′*)›
using *q qs′ q′ rs′* **by** *simp-all*

have ‹*i′* ∈ *branch-nominals ?branch*›
unfolding *branch-nominals-def* **using** *v* **by** *fastforce*
then have *?thesis* **if** ‹*A* ⊢ ([], *i′*) # *?branch*›
using *that GoTo* **by** *fast*
moreover have ‹(*mapi* (*?f v*) *qs*, *i′*) ∈. ([], *i′*) # *?branch*›
using *v* **by** *simp*
moreover have ‹(*mapi* (*?f w*) *rs*, *i′*) ∈. ([], *i′*) # *?branch*›
using *w* **by** *simp*
ultimately have *?thesis* **if** ‹*A* ⊢ ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
using *that* ∗ **by** (*meson DiaN′*)
moreover have ‹*j* ∈ *branch-nominals* (([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*)›
unfolding *branch-nominals-def* **by** *simp*
ultimately have *?thesis* **if** ‹*A* ⊢ ([], *j*) # ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
using *that GoTo* **by** *fast*
moreover have ‹(¬ (@ *j* (*Nom k*))) *at i′ in* ([], *j*) # ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
by *fastforce*
ultimately have *?thesis* **if** ‹*A* ⊢ ([¬ *Nom k*], *j*) # ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
using *that SatN′* **by** *fast*
moreover have ‹*Nom k at j in* ([¬ *Nom k*], *j*) # ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
using *k* **by** *fastforce*
moreover have ‹(¬ *Nom k*) *at j in* ([¬ *Nom k*], *j*) # ([¬ (@ *j* (*Nom k*))], *i′*) # *?branch*›
by *fastforce*
ultimately show *?thesis*
using *STA.Close* **by** *fast*
next
case *satn*
then obtain *r* **where** ∗:
‹(¬ (@ *j r*)) *on* (*mapi* (*?f v*) *qs*, *i′*)›
‹(¬ ¬ (@ *k r*)) *on* (*mapi* (*?f w*) *rs*, *i′*)›
using *q qs′ q′ rs′* **by** *auto*

have ‹*i′* ∈ *branch-nominals ?branch*›
unfolding *branch-nominals-def* **using** *v* **by** *fastforce*

```
    then have ?thesis if ‹A ⊢ ([], i') # ?branch›
      using that GoTo by fast
    moreover have ‹(mapi (?f w) rs, i') ∈. ([], i') # ?branch›
      using w by simp
    ultimately have ?thesis if ‹A ⊢ ([@ k r], i') # ?branch›
      using that ∗(2) by (meson Neg')
    moreover have ‹j ∈ branch-nominals (([@ k r], i') # ?branch)›
      unfolding branch-nominals-def using k by fastforce
    ultimately have ?thesis if ‹A ⊢ ([], j) # ([@ k r], i') # ?branch›
      using that GoTo by fast
    moreover have ‹(¬ (@ j r)) at i' in ([], j) # ([@ k r], i') # ?branch›
      using ∗(1) v by auto
    ultimately have ?thesis if ‹A ⊢ ([¬ r], j) # ([@ k r], i') # ?branch›
      using that SatN' by fast
    moreover have ‹k ∈ branch-nominals (([¬ r], j) # ([@ k r], i') # ?branch)›
      unfolding branch-nominals-def using k by fastforce
    ultimately have ?thesis if ‹A ⊢ ([], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      using that GoTo by fast
    moreover have ‹(@ k r) at i' in ([], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      by fastforce
    ultimately have ?thesis if ‹A ⊢ ([r], k) # ([¬ r], j) # ([@ k r], i') #
?branch›
      using that SatP' by fast
    moreover have
      ‹Nom k at j in ([r], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      ‹(¬ r) at j in ([r], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      using k by fastforce+
    ultimately have ?thesis if ‹A ⊢ ([¬ r, r], k) # ([¬ r], j) # ([@ k r], i') #
?branch›
      using that by (meson Nom' fm.distinct(21) fm.simps(18))
    moreover have
      ‹r at k in ([¬ r, r], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      ‹(¬ r) at k in ([¬ r, r], k) # ([¬ r], j) # ([@ k r], i') # ?branch›
      by fastforce+
    ultimately show ?thesis
      using STA.Close by fast
  next
    case sat
    then obtain r where ∗:
      ‹(@ k r) on (mapi (?f v) qs, i')›
      ‹(¬ (@ j r)) on (mapi (?f w) rs, i')›
      using q qs' q' rs' by auto

    have ‹j ∈ branch-nominals ?branch›
      unfolding branch-nominals-def using k by fastforce
    then have ?thesis if ‹A ⊢ ([], j) # ?branch›
      using that GoTo by fast
    moreover have ‹(¬ (@ j r)) at i' in ([], j) # ?branch›
      using ∗(2) w by auto
```

```
      ultimately have ?thesis if ‹A ⊢ ([¬ r], j) # ?branch›
        using that by (meson SatN')
      moreover have ‹k ∈ branch-nominals (([¬ r], j) # ?branch)›
        unfolding branch-nominals-def using k by fastforce
      ultimately have ?thesis if ‹A ⊢ ([], k) # ([¬ r], j) # ?branch›
        using that GoTo by fast
      moreover have ‹(@ k r) at i' in ([], k) # ([¬ r], j) # ?branch›
        using *(1) v by auto
      ultimately have ?thesis if ‹A ⊢ ([r], k) # ([¬ r], j) # ?branch›
        using that SatP' by fast
      moreover have
        ‹Nom k at j in ([r], k) # ([¬ r], j) # ?branch›
        ‹(¬ r) at j in ([r], k) # ([¬ r], j) # ?branch›
        using k by fastforce+
      ultimately have ?thesis if ‹A ⊢ ([¬ r, r], k) # ([¬ r], j) # ?branch›
        using that by (meson Nom' fm.distinct(21) fm.simps(18))
      moreover have
        ‹r at k in ([¬ r, r], k) # ([¬ r], j) # ?branch›
        ‹(¬ r) at k in ([¬ r, r], k) # ([¬ r], j) # ?branch›
        by fastforce+
      ultimately show ?thesis
        using STA.Close by fast
    next
      case old
      then have ‹p on (mapi (?f v) qs, i')› ‹(¬ p) on (mapi (?f w) rs, i')›
        using q qs' q' rs' by simp-all
      then show ?thesis
        using v w STA.Close[where p=p and i=i'] by fast
    qed
  qed
next
  case (Neg p a ps branch n)
  let ?f = ‹bridge k j xs›
  have p: ‹?f l l' (¬ ¬ p) = (¬ ¬ p)› for l l'
    by simp

  have ‹descendants k i ((p # ps, a) # branch) xs›
    using Neg(5) descendants-no-head by fast
  then have ‹A ⊢ mapi-branch ?f ((p # ps, a) # branch)›
    using Neg(4−) by blast
  moreover have ‹(length branch, length ps) ∉ xs›
    using Neg(5) descendants-oob-head by fast
  ultimately have ‹A ⊢ (p # mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    unfolding mapi-branch-def by simp
  moreover have ‹∃ l l'. ?f l l' (¬ ¬ p) at a in mapi-branch ?f ((ps, a) # branch)›
    using Neg(1) at-in-mapi-branch by fast
  then have ‹(¬ ¬ p) at a in (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
```

unfolding *mapi-branch-def* **using** *p* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
using *Neg′* **by** *fast*
then show *?case*
unfolding *mapi-branch-def* **by** *auto*
next
case (*DisP p q a ps branch n*)
let *?f* = ‹*bridge k j xs*›
have *p*: ‹*?f l l′* (*p* ∨ *q*) = (*p* ∨ *q*)› **for** *l l′*
by *simp*

have
‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *xs*›
‹*descendants k i* ((*q* # *ps*, *a*) # *branch*) *xs*›
using *DisP*(*8*) *descendants-no-head* **by** *fast+*
then have
‹*A* ⊢ *mapi-branch ?f* ((*p* # *ps*, *a*) # *branch*)›
‹*A* ⊢ *mapi-branch ?f* ((*q* # *ps*, *a*) # *branch*)›
using *DisP*(*5*−) **by** *blast+*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *DisP*(*8*) *descendants-oob-head* **by** *fast*
ultimately have
‹*A* ⊢ (*p* # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
‹*A* ⊢ (*q* # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **by** *simp-all*
moreover have ‹∃ *l l′*. *?f l l′* (*p* ∨ *q*) *at a in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *DisP*(*1*) *at-in-mapi-branch* **by** *fast*
then have ‹(*p* ∨ *q*) *at a in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **using** *p* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
using *DisP″* **by** *fast*
then show *?case*
unfolding *mapi-branch-def* **by** *auto*
next
case (*DisN p q a ps branch n*)
let *?f* = ‹*bridge k j xs*›
have *p*: ‹*?f l l′* (¬ (*p* ∨ *q*)) = (¬ (*p* ∨ *q*))› **for** *l l′*
by *simp*

have ‹*descendants k i* (((¬ *p*) # *ps*, *a*) # *branch*) *xs*›
using *DisN*(*5*) *descendants-no-head* **by** *fast*
then have ‹*descendants k i* (((¬ *q*) # (¬ *p*) # *ps*, *a*) # *branch*) *xs*›
using *descendants-no-head* **by** *fast*
then have ‹*A* ⊢ *mapi-branch ?f* (((¬ *q*) # (¬ *p*) # *ps*, *a*) # *branch*)›
using *DisN*(*4*−) **by** *blast*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›

using *DisN(5) descendants-oob-head* **by** *fast*
moreover have ‹(*length branch*, *1* + *length ps*) ∉ *xs*›
using *DisN(5) descendants-oob-head′* **by** *fast*
ultimately have ‹*A* ⊢ ((¬ *q*) # (¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **by** *simp*
moreover have ‹∃ *l l′*. *?f l l′* (¬ (*p* ∨ *q*)) *at a in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *DisN(1) at-in-mapi-branch* **by** *fast*
then have ‹(¬ (*p* ∨ *q*)) *at a in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **using** *p* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
using *DisN′* **by** *fast*
then show *?case*
unfolding *mapi-branch-def* **by** *auto*
next
case (*DiaP p a ps branch i′ n*)
let *?f* = ‹*bridge k j xs*›
have *p*: ‹*?f l l′* (◊ *p*) = (◊ *p*)› **for** *l l′*
using *DiaP(3) bridge-proper-dia* **by** *fast*

have ‹*branch-nominals* (*mapi-branch ?f* ((*ps*, *a*) # *branch*)) = *branch-nominals* ((*ps*, *a*) # *branch*)›
using *DiaP(8−) nominals-mapi-branch-bridge*[**where** *j*=*j* **and** *k*=*k* **and** *branch*=‹(*ps*, *a*) # *branch*›]
by *auto*
then have *i′*:
‹*i′* ∉ *A* ∪ *branch-nominals* ((*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*)›
unfolding *mapi-branch-def* **using** *DiaP(2)* **by** *simp*

have *1*: ‹*?f* (*length branch*) (*1* + *length ps*) (**@** *i′ p*) = (**@** *i′ p*)›
by *simp*
have ‹*i′* ≠ *k*›
using *DiaP(2, 8)* **unfolding** *branch-nominals-def* **by** *fastforce*
then have *2*: ‹*?f* (*length branch*) (*length ps*) (◊ *Nom i′*) = (◊ *Nom i′*)›
by *simp*

have ‹*i′* ≠ *j*›
using *DiaP(2, 8)* **unfolding** *branch-nominals-def* **by** *fastforce*
moreover have ‹*descendants k i* (((**@** *i′ p*) # (◊ *Nom i′*) # *ps*, *a*) # *branch*) *xs*›
using *DiaP(7) descendants-block*[**where** *ps′*=‹[-, -]›] **by** *fastforce*
ultimately have ‹*A* ⊢ *mapi-branch ?f* (((**@** *i′ p*) # (◊ *Nom i′*) # *ps*, *a*) # *branch*)›
using *DiaP(4−)* **by** *blast*
then have ‹*A* ⊢ ((**@** *i′ p*) # (◊ *Nom i′*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) #

```
    mapi-branch ?f branch›
    unfolding mapi-branch-def using 1 by (simp add: 2)
  moreover have ‹∃ l l'. ?f l l' (◇ p) at a in mapi-branch ?f ((ps, a) # branch)›
    using DiaP(1) at-in-mapi-branch by fast
  then have ‹(◇ p) at a in (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    unfolding mapi-branch-def using p by simp
  ultimately have ‹A ⊢ (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    using i' DiaP(3) fin DiaP'' by fast
  then show ?case
    unfolding mapi-branch-def by simp
next
  case (DiaN p a ps branch i' n)
  have p: ‹bridge k j xs l l' (¬ (◇ p)) = (¬ (◇ p))› for xs l l'
    by simp

  obtain rs where rs: ‹(rs, a) ∈. (ps, a) # branch› ‹(◇ Nom i') on (rs, a)›
    using DiaN(2) by fast
  obtain v where v: ‹((ps, a) # branch) !. v = Some (rs, a)›
    using rs(1) rev-nth-mem by fast
  obtain v' where v': ‹rs !. v' = Some (◇ Nom i')›
    using rs(2) rev-nth-on by fast

  show ?case
  proof (cases ‹(v, v') ∈ xs›)
    case True
    then have ‹i' = k›
      using DiaN(6) v v' descendants-types by fast

    let ?xs = ‹{(length branch, length ps)} ∪ xs›
    let ?f = ‹bridge k j ?xs›
    let ?branch = ‹((¬ (@ i' p)) # ps, a) # branch›

    obtain rs' where
      ‹(((¬ (@ k p)) # ps, a) # branch) !. v = Some (rs', a)›
      ‹rs' !. v' = Some (◇ Nom i')›
      using v v' index-Cons by fast
    moreover have ‹descendants k i (((¬ (@ k p)) # ps, a) # branch) xs›
      using DiaN(6) descendants-block[where ps'=‹[-]›] by fastforce
    moreover have ‹?branch !. length branch = Some ((¬ (@ k p)) # ps, a)›
      using ‹i' = k› by simp
    moreover have ‹((¬ (@ k p)) # ps) !. length ps = Some (¬ (@ k p))›
      by simp
    ultimately have ‹descendants k i (((¬ (@ k p)) # ps, a) # branch) ?xs›
      using True ‹i' = k› Derived[where branch=‹- # branch›] by simp

    then have ‹A ⊢ mapi-branch ?f (((¬ (@ k p)) # ps, a) # branch)›
      using ‹i' = k› DiaN(5-) by blast
```

then have ‹$A \vdash ((\neg (@ j\ p)) \# mapi\ (?f\ (length\ branch))\ ps,\ a) \#$
mapi-branch (*bridge k j ?xs*) *branch*›
unfolding *mapi-branch-def* **using** ‹$i' = k$› **by** *simp*
moreover have ‹$\exists\ l\ l'.\ ?f\ l\ l'\ (\neg (\Diamond\ p))$ *at a in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *DiaN*(*1*) *at-in-mapi-branch* **by** *fast*
then have ‹$(\neg (\Diamond\ p))$ *at a in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **using** *p*[**where** *xs*=‹*?xs*›] **by** *simp*
moreover have ‹(*mapi* (*?f v*) *rs*, *a*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *v rev-nth-mapi-branch* **by** *fast*
then have ‹(*mapi* (*?f v*) *rs*, *a*) ∈
set ((*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*)›
unfolding *mapi-branch-def* **by** *simp*
moreover have ‹$?f\ v\ v'\ (\Diamond\ Nom\ i')$ *on* (*mapi* (*?f v*) *rs*, *a*)›
using v' *rev-nth-mapi-block* **by** *fast*
then have ‹$(\Diamond\ Nom\ j)$ *on* (*mapi* (*?f v*) *rs*, *a*)›
using *True* ‹$i' = k$› **by** *simp*
ultimately have ‹$A \vdash$ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
by (*meson DiaN'*)
then have ‹$A \vdash$ (*mapi* (*bridge k j xs* (*length branch*)) *ps*, *a*) #
mapi-branch (*bridge k j xs*) *branch*›
using *mapi-branch-head-add-oob*[**where** *branch*=*branch* **and** *ps*=*ps*] **unfolding** *mapi-branch-def*
by *simp*
then show *?thesis*
unfolding *mapi-branch-def* **by** *simp*
next
case *False*
let *?f* = ‹*bridge k j xs*›

have ‹*descendants k i* $(((\neg (@\ i'\ p)) \#\ ps,\ a) \#\ branch)$ *xs*›
using *DiaN*(*6*) *descendants-no-head* **by** *fast*
then have ‹$A \vdash$ *mapi-branch ?f* $(((\neg (@\ i'\ p)) \#\ ps,\ a) \#\ branch)$›
using *DiaN*(*5−*) **by** *blast*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *DiaN*(*6*) *descendants-oob-head* **by** *fast*
ultimately have ‹$A \vdash ((\neg (@\ i'\ p)) \#\ mapi\ (?f\ (length\ branch))\ ps,\ a) \#$
mapi-branch ?f branch›
unfolding *mapi-branch-def* **by** *simp*
moreover have ‹$\exists\ l\ l'.\ ?f\ l\ l'\ (\neg (\Diamond\ p))$ *at a in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *DiaN*(*1*) *at-in-mapi-branch* **by** *fast*
then have ‹$(\neg (\Diamond\ p))$ *at a in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **using** *p*[**where** *xs*=‹*xs*›] **by** *simp*
moreover have ‹(*mapi* (*?f v*) *rs*, *a*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *v rev-nth-mapi-branch* **by** *fast*

```
    then have ‹(mapi (?f v) rs, a) ∈
        set ((mapi (?f (length branch)) ps, a) # mapi-branch ?f branch)›
      unfolding mapi-branch-def by simp
    moreover have ‹?f v v' (◇ Nom i') on (mapi (?f v) rs, a)›
      using v' rev-nth-mapi-block by fast
    then have ‹(◇ Nom i') on (mapi (?f v) rs, a)›
      using False by simp
    ultimately have ‹A ⊢ (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
      by (meson DiaN')
    then show ?thesis
      unfolding mapi-branch-def by simp
  qed
next
  case (SatP a p b ps branch n)
  let ?f = ‹bridge k j xs›
  have p: ‹?f l l' (@ a p) = (@ a p)› for l l'
    by simp

  have ‹descendants k i ((p # ps, a) # branch) xs›
    using SatP(5) descendants-no-head by fast
  then have ‹A ⊢ mapi-branch ?f ((p # ps, a) # branch)›
    using SatP(4−) by blast
  moreover have ‹(length branch, length ps) ∉ xs›
    using SatP(5) descendants-oob-head by fast
  ultimately have ‹A ⊢ (p # mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    unfolding mapi-branch-def by simp
  moreover have ‹∃ l l'. ?f l l' (@ a p) at b in mapi-branch ?f ((ps, a) # branch)›
    using SatP(1) at-in-mapi-branch by fast
  then have ‹(@ a p) at b in (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    unfolding mapi-branch-def using p by simp
  ultimately have ‹A ⊢ (mapi (?f (length branch)) ps, a) # mapi-branch ?f
branch›
    using SatP' by fast
  then show ?case
    unfolding mapi-branch-def by simp
next
  case (SatN a p b ps branch n)
  obtain qs where qs: ‹(qs, b) ∈. (ps, a) # branch› ‹(¬ (@ a p)) on (qs, b)›
    using SatN(1) by fast
  obtain v where v: ‹((ps, a) # branch) !. v = Some (qs, b)›
    using qs(1) rev-nth-mem by fast
  obtain v' where v': ‹qs !. v' = Some (¬ (@ a p))›
    using qs(2) rev-nth-on by fast

  show ?case
  proof (cases ‹(v, v') ∈ xs›)
```

case *True*
then have ‹$a = k$›
using *SatN*(*5*) *v v'* *descendants-types* **by** *fast*

let *?f* = ‹*bridge k j xs*›
let *?branch* = ‹((¬ *p*) # *ps*, *a*) # *branch*›
have *p*: ‹*?f v v'* (¬ (**@** *k p*)) = (¬ (**@** *j p*))›
using *True* **by** *simp*

obtain *rs'* **where**
‹*?branch* !. *v* = *Some* (*rs'*, *b*)›
‹*rs'* !. *v'* = *Some* (¬ (**@** *k p*))›
using *v v'* ‹$a = k$› *index-Cons* **by** *fast*
have ‹*descendants k i ?branch xs*›
using *SatN*(*5*) *descendants-no-head* **by** *fast*
then have ‹*A* ⊢ *mapi-branch ?f ?branch*›
using ‹$a = k$› *SatN*(*4*−) **by** *blast*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
using *SatN*(*5*) *descendants-oob-head* **by** *fast*
ultimately have ‹*A* ⊢ ((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **using** ‹$a = k$› **by** *simp*
moreover have ‹*set* (((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*) ⊆
set (((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*)›
by *auto*
ultimately have ∗:
‹*A* ⊢ ((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*›
using *inf fin STA-struct* **by** *fastforce*

have *k*: ‹*Nom k at j in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *SatN*(*6*) *nom-at-in-bridge* **unfolding** *mapi-branch-def* **by** *fastforce*

have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *v rev-nth-mapi-branch* **by** *fast*
moreover have ‹*?f v v'* (¬ (**@** *k p*)) *on* (*mapi* (*?f v*) *qs*, *b*)›
using *v'* ‹$a = k$› *rev-nth-mapi-block* **by** *fast*
then have ‹(¬ (**@** *j p*)) *on* (*mapi* (*?f v*) *qs*, *b*)›
using *p* **by** *simp*
ultimately have *satn*: ‹(¬ (**@** *j p*)) *at b in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
by *blast*

have ‹*j* ∈ *branch-nominals* (*mapi-branch ?f* ((*ps*, *a*) # *branch*))›
unfolding *branch-nominals-def* **using** *k* **by** *fastforce*
then have *?thesis* **if** ‹*A* ⊢ ([], *j*) # *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *that GoTo* **by** *fast*
moreover have ‹(¬ (**@** *j p*)) *at b in* ([], *j*) # *mapi-branch ?f* ((*ps*, *a*) #

branch)›
    using *satn* **by** *auto*
    ultimately have *?thesis* **if** ‹*A* ⊢ ([¬ *p*], *j*) # *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
    using *that SatN′* **by** *fast*
  then have *?thesis* **if** ‹*A* ⊢ ([¬ *p*], *j*) # *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
    using *that SatN′* **by** *fast*
  then have *?thesis* **if**
   ‹*A* ⊢ ([¬ *p*], *j*) # (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
    using *that* **unfolding** *mapi-branch-def* **by** *simp*
    moreover have ‹*set* ((*mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*) ⊆
     set (([¬ *p*], *j*) # (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*)›
    by *auto*
  ultimately have *?thesis* **if**
   ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*›
    using *that inf fin STA-struct* **by** *blast*
  moreover have
   ‹*Nom k at j in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*›
    using *k* **unfolding** *mapi-branch-def* **by** *auto*
  moreover have
   ‹(¬ *p*) *at j in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*›
    by *fastforce*
  ultimately have *?thesis* **if**
   ‹*A* ⊢ ((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # ([¬ *p*], *j*) # *mapi-branch ?f branch*›
    using *that* ‹*a* = *k*› **by** (*meson Nom′ fm.distinct*(*21*) *fm.simps*(*18*))
  then show *?thesis*
    using ∗ **by** *blast*
 next
  case *False*
  let *?f* = ‹*bridge k j xs*›

  have ‹*descendants k i* (((¬ *p*) # *ps*, *a*) # *branch*) *xs*›
    using *SatN*(*5*) *descendants-no-head* **by** *fast*
  then have ‹*A* ⊢ *mapi-branch* (*bridge k j xs*) (((¬ *p*) # *ps*, *a*) # *branch*)›
    using *SatN*(*4*−) **by** *blast*
  moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
    using *SatN*(*5*) *descendants-oob-head* **by** *fast*
 ultimately have ‹*A* ⊢ ((¬ *p*) # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
    unfolding *mapi-branch-def* **by** *simp*
  moreover have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
    using *v rev-nth-mapi-branch* **by** *fast*
  then have ‹(*mapi* (*?f v*) *qs*, *b*) ∈
     set ((*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*)›
    unfolding *mapi-branch-def* **by** *simp*

moreover have ‹*?f v v′* (¬ (@ *a p*)) *on* (*mapi* (*?f v*) *qs*, *b*)›
using *v′ rev-nth-mapi-block* **by** *fast*
then have ‹(¬ (@ *a p*)) *on* (*mapi* (*?f v*) *qs*, *b*)›
using *False* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
by (*meson SatN′*)
then show *?thesis*
unfolding *mapi-branch-def* **by** *simp*
qed
next
case (*GoTo i′ n ps a branch*)
let *?f* = ‹*bridge k j xs*›

have ‹*descendants k i* (([], *i′*) # (*ps*, *a*) # *branch*) *xs*›
using *GoTo*(*4*) *descendants-branch*[**where** *extra*=‹[-]›] **by** *simp*
then have ‹*A* ⊢ *mapi-branch ?f* (([], *i′*) # (*ps*, *a*) # *branch*)›
using *GoTo*(*3*, *5*−) **by** *auto*
then have ‹*A* ⊢ ([], *i′*) # *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
unfolding *mapi-branch-def* **by** *simp*
moreover have
‹*branch-nominals* (*mapi-branch ?f* ((*ps*, *a*) # *branch*)) = *branch-nominals* ((*ps*, *a*) # *branch*)›
using *GoTo*(*5*−) *nominals-mapi-branch-bridge*[**where** *j*=*j* **and** *k*=*k* **and** *branch*=‹(*ps*, *a*) # *branch*›]
by *auto*
then have ‹*i′* ∈ *branch-nominals* (*mapi-branch* (*bridge k j xs*) ((*ps*, *a*) # *branch*))›
using *GoTo*(*1*) **by** *blast*
ultimately show *?case*
using *STA.GoTo* **by** *fast*
next
case (*Nom p b ps a branch n*)
show *?case*
proof (*cases* ‹∃ *j*. *p* = *Nom j*›)
case *True*
let *?f* = ‹*bridge k j xs*›

have ‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *xs*›
using *Nom*(*7*) *descendants-block*[**where** *ps′*=‹[*p*]›] **by** *simp*
then have ‹*A* ⊢ *mapi-branch ?f* ((*p* # *ps*, *a*) # *branch*)›
using *Nom*(*6*−) **by** *blast*
moreover have ‹*?f* (*length branch*) (*length ps*) *p* = *p*›
using *True* **by** *auto*
ultimately have ‹*A* ⊢ (*p* # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **by** *simp*
moreover have ‹*p at b in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *Nom*(*1*) *True nom-at-in-bridge* **by** *fast*
then have ‹*p at b in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f*

branch›
      unfolding *mapi-branch-def* **by** *simp*
    moreover have ‹*Nom a at b in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
      using *Nom*(*2*) *True nom-at-in-bridge* **by** *fast*
    then have ‹*Nom a at b in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
      unfolding *mapi-branch-def* **by** *simp*
    ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
      by (*meson Nom′ Nom.hyps*(*3*))
    then show *?thesis*
      unfolding *mapi-branch-def* **by** *simp*
  next
    case *False*
    obtain *qs* **where** *qs*: ‹(*qs*, *b*) ∈. (*ps*, *a*) # *branch*› ‹*p on* (*qs*, *b*)›
      using *Nom*(*1*) **by** *blast*
    obtain *v* **where** *v*: ‹((*ps*, *a*) # *branch*) !. *v* = *Some* (*qs*, *b*)›
      using *qs*(*1*) *rev-nth-mem* **by** *fast*
    obtain *v′* **where** *v′*: ‹*qs* !. *v′* = *Some p*›
      using *qs*(*2*) *False rev-nth-on* **by** *fast*

    show *?thesis*
    proof (*cases* ‹(*v*, *v′*) ∈ *xs*›)
      case *True*
      let *?xs* = ‹{(*length branch*, *length ps*)} ∪ *xs*›
      let *?f* = ‹*bridge k j ?xs*›

      let *?p* = ‹*bridge′ k j p*›
      have *p*: ‹*?f v v′ p* = *?p*›
        using *True* **by** *simp*

      consider (*dia*) ‹*p* = (◇ *Nom k*)› | (*satn*) *q* **where** ‹*p* = (¬ (@ *k q*))› | (*old*) ‹*?p* = *p*›
        by (*meson Nom.prems*(*1*) *True descendants-types v v′*)
      then have *A*: ‹∀ *i*. *?p* = *Nom i* ∨ *?p* = (◇ *Nom i*) ⟶ *i* ∈ *A*›
        using *Nom*(*3*) ‹*j* ∈ *A*› **by** *cases simp-all*

      obtain *qs′* **where**
        ‹((*p* # *ps*, *a*) # *branch*) !. *v* = *Some* (*qs′*, *b*)›
        ‹*qs′* !. *v′* = *Some p*›
        using *v v′ index-Cons* **by** *fast*
      moreover have ‹*Nom a at b in* (*p* # *ps*, *a*) # *branch*›
        using *Nom*(*2*) **by** *fastforce*
      moreover have ‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *xs*›
        using *Nom*(*7*) *descendants-block*[**where** *ps′*=‹[*p*]›] **by** *simp*
      moreover have
        ‹((*p* # *ps*, *a*) # *branch*) !. *length branch* = *Some* (*p* # *ps*, *a*)›
        ‹(*p* # *ps*) !. *length ps* = *Some p*›
        by *simp-all*

ultimately have ‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *?xs*›
  using *True Copied* **by** *fast*
then have ‹*A* ⊢ *mapi-branch ?f* ((*p* # *ps*, *a*) # *branch*)›
  using *Nom*(*6*−) **by** *blast*
then have ‹*A* ⊢ (*?p* # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
  unfolding *mapi-branch-def* **by** *simp*

moreover have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
  using *v rev-nth-mapi-branch* **by** *fast*
then have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
  unfolding *mapi-branch-def* **by** *simp*
moreover have ‹*?f v v′ p on* (*mapi* (*?f v*) *qs*, *b*)›
  using *v v′ rev-nth-mapi-block* **by** *fast*
then have ‹*?p on* (*mapi* (*?f v*) *qs*, *b*)›
  using *p* **by** *simp*

moreover have ‹*Nom a at b in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
  using *Nom*(*2*) *nom-at-in-bridge* **by** *fast*
then have ‹*Nom a at b in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
  unfolding *mapi-branch-def* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
  using *A* **by** (*meson Nom′ Nom*(*3*))
then have ‹*A* ⊢ (*mapi* (*bridge k j xs* (*length branch*)) *ps*, *a*) # *mapi-branch* (*bridge k j xs*) *branch*›
  using *mapi-branch-head-add-oob*[**where** *branch*=*branch* **and** *ps*=*ps*]
  unfolding *mapi-branch-def* **by** *simp*
then show *?thesis*
  unfolding *mapi-branch-def* **by** *simp*
next
case *False*
let *?f* = ‹*bridge k j xs*›

have ‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *xs*›
  using *Nom*(*7*) *descendants-no-head* **by** *fast*
then have ‹*A* ⊢ *mapi-branch ?f* ((*p* # *ps*, *a*) # *branch*)›
  using *Nom*(*6*−) **by** *blast*
moreover have ‹(*length branch*, *length ps*) ∉ *xs*›
  using *Nom*(*7*) *descendants-oob-head* **by** *fast*
ultimately have ‹*A* ⊢ (*p* # *mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
  unfolding *mapi-branch-def* **by** *simp*

moreover have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. *mapi-branch ?f* ((*ps*, *a*) # *branch*)›
  using *v rev-nth-mapi-branch* **by** *fast*
then have ‹(*mapi* (*?f v*) *qs*, *b*) ∈. (*mapi* (*?f* (*length branch*)) *ps*, *a*) #

mapi-branch ?f branch›
unfolding *mapi-branch-def* **by** *simp*
moreover have ‹*?f v v′ p on* (*mapi* (*?f v*) *qs*, *b*)›
using *v v′ rev-nth-mapi-block* **by** *fast*
then have ‹*p on* (*mapi* (*?f v*) *qs*, *b*)›
using *False* **by** *simp*
moreover have ‹*Nom a at b in mapi-branch ?f* ((*ps*, *a*) # *branch*)›
using *Nom*(*2*) *nom-at-in-bridge* **by** *fast*
then have ‹*Nom a at b in* (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
unfolding *mapi-branch-def* **by** *simp*
ultimately have ‹*A* ⊢ (*mapi* (*?f* (*length branch*)) *ps*, *a*) # *mapi-branch ?f branch*›
by (*meson Nom′ Nom*(*3*))
then show *?thesis*
unfolding *mapi-branch-def* **by** *simp*
qed
qed
qed

lemma *STA-bridge*:
fixes *i* :: ′*b*
assumes
inf: ‹*infinite* (*UNIV* :: ′*b set*)› **and**
‹*A* ⊢ *branch*› ‹*descendants k i branch xs*›
‹*Nom k at j in branch*›
‹*finite A*› ‹*j* ∈ *A*›
shows ‹*A* ⊢ *mapi-branch* (*bridge k j xs*) *branch*›
proof −
have ‹*A* ⊢ ([], *j*) # *branch*›
using *assms*(*2*, *5*−*6*) *inf STA-struct*[**where** *branch′*=‹([], *j*) # *branch*›] **by** *auto*
moreover have ‹*descendants k i* (([], *j*) # *branch*) *xs*›
using *assms*(*3*) *descendants-branch*[**where** *extra*=‹[-]›] **by** *fastforce*
ultimately have ‹*A* ⊢ *mapi-branch* (*bridge k j xs*) (([], *j*) # *branch*)›
using *STA-bridge′ inf assms*(*3*−) **by** *fast*
then have ∗: ‹*A* ⊢ ([], *j*) # *mapi-branch* (*bridge k j xs*) *branch*›
unfolding *mapi-branch-def* **by** *simp*
have ‹*branch-nominals* (*mapi-branch* (*bridge k j xs*) *branch*) = *branch-nominals branch*›
using *nominals-mapi-branch-bridge assms*(*4*−) **by** *fast*
moreover have ‹*j* ∈ *branch-nominals branch*›
using *assms*(*4*) **unfolding** *branch-nominals-def* **by** *fastforce*
ultimately have ‹*j* ∈ *branch-nominals* (*mapi-branch* (*bridge k j xs*) *branch*)›
by *simp*
then show *?thesis*
using ∗ *GoTo* **by** *fast*
qed

10.4 Derivation

theorem *Bridge*:
  fixes $i :: 'b$
  assumes *inf*: ‹*infinite* (*UNIV* :: *'b set*)› **and** *fin*: ‹*finite A*› **and** ‹$j \in A$›
    ‹*Nom k at j in* (*ps*, *i*) # *branch*› ‹(◊ *Nom j*) *at i in* (*ps*, *i*) # *branch*›
    ‹$A \vdash$ ((◊ *Nom k*) # *ps*, *i*) # *branch*›
  shows ‹$A \vdash$ (*ps*, *i*) # *branch*›
proof −
  let *?xs* = ‹{(*length branch*, *length ps*)}›

  have ‹*descendants k i* (((◊ *Nom k*) # *ps*, *i*) # *branch*) *?xs*›
    using *Initial* **by** *force*
  moreover have ‹*Nom k at j in* ((◊ *Nom k*) # *ps*, *i*) # *branch*›
    using *assms*(*4*) **by** *fastforce*
  ultimately have ‹$A \vdash$ *mapi-branch* (*bridge k j ?xs*) (((◊ *Nom k*) # *ps*, *i*) # *branch*)›
    using *STA-bridge inf fin assms*(*3*, *6*) **by** *fast*
  then have ‹$A \vdash$ ((◊ *Nom j*) # *mapi* (*bridge k j ?xs* (*length branch*)) *ps*, *i*) # *mapi-branch* (*bridge k j ?xs*) *branch*›
    unfolding *mapi-branch-def* **by** *simp*
  moreover have ‹*mapi-branch* (*bridge k j* {(*length branch*, *length ps*)}) *branch* = *mapi-branch* (*bridge k j* {}) *branch*›
    using *mapi-branch-add-oob*[**where** *xs*=‹{}›] **by** *fastforce*
  moreover have ‹*mapi* (*bridge k j ?xs* (*length branch*)) *ps* = *mapi* (*bridge k j* {} (*length branch*)) *ps*›
    using *mapi-block-add-oob*[**where** *xs*=‹{}› **and** *ps*=*ps*] **by** *simp*
  ultimately have ‹$A \vdash$ ((◊ *Nom j*) # *ps*, *i*) # *branch*›
    using *mapi-block-id*[**where** *ps*=*ps*] *mapi-branch-id*[**where** *branch*=*branch*] **by** *simp*
  then show *?thesis*
    using *Dup assms*(*5*) **by** (*metis new-def*)
qed

11 Completeness

11.1 Hintikka

abbreviation *at-in-set* :: ‹(*'a*, *'b*) *fm* ⇒ *'b* ⇒ (*'a*, *'b*) *block set* ⇒ *bool*› **where**
  ‹*at-in-set p a S* ≡ ∃ *ps*. (*ps*, *a*) ∈ *S* ∧ *p on* (*ps*, *a*)›

notation *at-in-set* (‹- *at* - *in''* -› [*51*, *51*, *51*] *50*)

A set of blocks is Hintikka if it satisfies the following requirements. Intuitively, if it corresponds to an exhausted open branch with respect to the fixed set of allowed nominals A. For example, we only require symmetry, "if j occurs at i then i occurs at j" if $i \in A$.

locale *Hintikka* =
  fixes *A* :: ‹*'b set*› **and** *H* :: ‹(*'a*, *'b*) *block set*› **assumes**

ProP: ‹*Nom j at i in' H* ⟹ *Pro x at j in' H* ⟹ ¬ (¬ *Pro x*) *at i in' H*› **and**
NomP: ‹*Nom a at i in' H* ⟹ ¬ (¬ *Nom a*) *at i in' H*› **and**
NegN: ‹(¬ ¬ *p*) *at i in' H* ⟹ *p at i in' H*› **and**
DisP: ‹(*p* ∨ *q*) *at i in' H* ⟹ *p at i in' H* ∨ *q at i in' H*› **and**
DisN: ‹(¬ (*p* ∨ *q*)) *at i in' H* ⟹ (¬ *p*) *at i in' H* ∧ (¬ *q*) *at i in' H*› **and**
DiaP: ‹∄ *a*. *p* = *Nom a* ⟹ (◇ *p*) *at i in' H* ⟹
∃ *j*. (◇ *Nom j*) *at i in' H* ∧ (@ *j p*) *at i in' H*› **and**
DiaN: ‹(¬ (◇ *p*)) *at i in' H* ⟹ (◇ *Nom j*) *at i in' H* ⟹ (¬ (@ *j p*)) *at i in' H*› **and**
SatP: ‹(@ *i p*) *at a in' H* ⟹ *p at i in' H*› **and**
SatN: ‹(¬ (@ *i p*)) *at a in' H* ⟹ (¬ *p*) *at i in' H*› **and**
GoTo: ‹*i* ∈ *nominals p* ⟹ ∃ *a*. *p at a in' H* ⟹ ∃ *ps*. (*ps*, *i*) ∈ *H*› **and**
Nom: ‹∀ *a*. *p* = *Nom a* ∨ *p* = (◇ *Nom a*) ⟶ *a* ∈ *A* ⟹
p at i in' H ⟹ *Nom j at i in' H* ⟹ *p at j in' H*›

Two nominals *i* and *j* are equivalent in respect to a Hintikka set *H* if *H* contains an *i*-block with *j* on it. This is an equivalence relation on the names in *H* intersected with the allowed nominals *A*.

definition *hequiv* :: ‹(*'a*, *'b*) *block set* ⇒ *'b* ⇒ *'b* ⇒ *bool*› **where**
‹*hequiv H i j* ≡ *Nom j at i in' H*›

abbreviation *hequiv-rel* :: ‹*'b set* ⇒ (*'a*, *'b*) *block set* ⇒ (*'b* × *'b*) *set*› **where**
‹*hequiv-rel A H* ≡ {(*i*, *j*) |*i j*. *hequiv H i j* ∧ *i* ∈ *A* ∧ *j* ∈ *A*}›

definition *names* :: ‹(*'a*, *'b*) *block set* ⇒ *'b set*› **where**
‹*names H* ≡ {*i* |*ps i*. (*ps*, *i*) ∈ *H*}›

lemma *hequiv-refl*: ‹*i* ∈ *names H* ⟹ *hequiv H i i*›
unfolding *hequiv-def names-def* **by** *simp*

lemma *hequiv-refl'*: ‹(*ps*, *i*) ∈ *H* ⟹ *hequiv H i i*›
using *hequiv-refl* **unfolding** *names-def* **by** *fastforce*

lemma *hequiv-sym'*:
assumes ‹*Hintikka A H*› ‹*i* ∈ *A*› ‹*hequiv H i j*›
shows ‹*hequiv H j i*›
proof −
have ‹*i* ∈ *A* ⟶ *Nom i at i in' H* ⟶ *Nom j at i in' H* ⟶ *Nom i at j in' H*› **for** *i j*
using *assms*(*1*) *Hintikka.Nom* **by** *fast*
then show *?thesis*
using *assms*(*2*−) **unfolding** *hequiv-def* **by** *auto*
qed

lemma *hequiv-sym*: ‹*Hintikka A H* ⟹ *i* ∈ *A* ⟹ *j* ∈ *A* ⟹ *hequiv H i j* ⟷ *hequiv H j i*›
by (*meson hequiv-sym'*)

lemma *hequiv-trans*:

assumes ‹*Hintikka A H*› ‹*i* ∈ *A*› ‹*k* ∈ *A*› ‹*hequiv H i j*› ‹*hequiv H j k*›
shows ‹*hequiv H i k*›
proof −
have ‹*hequiv H j i*›
by (*meson assms*(*1*−*2*, *4*) *hequiv-sym*′)
moreover have ‹*k* ∈ *A* ⟶ *Nom k at j in*′ *H* ⟶ *Nom i at j in*′ *H* ⟶ *Nom k at i in*′ *H*› **for** *i k j*
using *assms*(*1*) *Hintikka.Nom* **by** *fast*
ultimately show *?thesis*
using *assms*(*3*−) **unfolding** *hequiv-def* **by** *blast*
qed

lemma *hequiv-names*: ‹*hequiv H i j* ⟹ *i* ∈ *names H*›
unfolding *hequiv-def names-def* **by** *blast*

lemma *hequiv-names-rel*:
assumes ‹*Hintikka A H*›
shows ‹*hequiv-rel A H* ⊆ *names H* × *names H*›
using *assms hequiv-names hequiv-sym* **by** *fast*

lemma *hequiv-refl-rel*:
assumes ‹*Hintikka A H*›
shows ‹*refl-on* (*names H* ∩ *A*) (*hequiv-rel A H*)›
unfolding *refl-on-def* **using** *assms hequiv-refl hequiv-names-rel* **by** *fast*

lemma *hequiv-sym-rel*: ‹*Hintikka A H* ⟹ *sym* (*hequiv-rel A H*)›
unfolding *sym-def* **using** *hequiv-sym* **by** *fast*

lemma *hequiv-trans-rel*: ‹*Hintikka B A* ⟹ *trans* (*hequiv-rel B A*)›
unfolding *trans-def* **using** *hequiv-trans* **by** *fast*

lemma *hequiv-rel*: ‹*Hintikka A H* ⟹ *equiv* (*names H* ∩ *A*) (*hequiv-rel A H*)›
using *hequiv-refl-rel hequiv-sym-rel hequiv-trans-rel* **by** (*rule equivI*)

lemma *nominal-in-names*:
assumes ‹*Hintikka A H*› ‹∃ *block* ∈ *H*. *i* ∈ *block-nominals block*›
shows ‹*i* ∈ *names H*›
using *assms Hintikka.GoTo* **unfolding** *names-def* **by** *fastforce*

11.1.1 Named model

Given a Hintikka set H, a formula p on a block in H and a set of allowed nominals A which contains all "root-like" nominals in p we construct a model that satisfies p.

The worlds of our model are sets of equivalent nominals and nominals are assigned to the equivalence class of an equivalent allowed nominal. This definition resembles the "ur-father" notion.

From a world *is*, we can reach a world *js* iff there is an $i \in$ *is* and a $j \in$ *js* s.t. there is an i-block in H with ◊ *Nom j* on it.

A propositional symbol p is true in a world *is* if there exists an $i \in$ *is* s.t. p occurs on an i-block in H.

definition *assign* :: ‹*′b set ⇒ (′a, ′b) block set ⇒ ′b ⇒ ′b set*› **where**
  ‹*assign A H i ≡ if ∃ a. a ∈ A ∧ Nom a at i in′ H*
    then proj (hequiv-rel A H) (SOME a. a ∈ A ∧ Nom a at i in′ H)
    else {i}›

definition *reach* :: ‹*′b set ⇒ (′a, ′b) block set ⇒ ′b set ⇒ ′b set set*› **where**
  ‹*reach A H is ≡ {assign A H j |i j. i ∈ is ∧ (◊ Nom j) at i in′ H}*›

definition *val* :: ‹*(′a, ′b) block set ⇒ ′b set ⇒ ′a ⇒ bool*› **where**
  ‹*val H is x ≡ ∃ i ∈ is. Pro x at i in′ H*›

lemma *ex-assignment*:
  assumes ‹*Hintikka A H*›
  shows ‹*assign A H i ≠ {}*›
proof (*cases* ‹*∃ b. b ∈ A ∧ Nom b at i in′ H*›)
  case *True*
  let *?b* = ‹*SOME b. b ∈ A ∧ Nom b at i in′ H*›
  have ∗: ‹*?b ∈ A ∧ Nom ?b at i in′ H*›
    using *someI-ex True* **.**
  moreover from *this* **have** ‹*hequiv H ?b ?b*›
    using *assms block-nominals nominal-in-names hequiv-refl*
    by (*metis* (*no-types*, *lifting*) *nominals.simps*(*2*) *singletonI*)
  ultimately show *?thesis*
    unfolding *assign-def proj-def* **by** *auto*
next
  case *False*
  then show *?thesis*
    unfolding *assign-def* **by** *auto*
qed

lemma *ur-closure*:
  assumes ‹*Hintikka A H*› ‹*p at i in′ H*› ‹*∀ a. p = Nom a ∨ p = (◊ Nom a) ⟶ a ∈ A*›
  shows ‹*∀ a ∈ assign A H i. p at a in′ H*›
proof (*cases* ‹*∃ b. b ∈ A ∧ Nom b at i in′ H*›)
  case *True*
  let *?b* = ‹*SOME b. b ∈ A ∧ Nom b at i in′ H*›
  have ∗: ‹*?b ∈ A ∧ Nom ?b at i in′ H*›
    using *someI-ex True* **.**
  then have ‹*p at ?b in′ H*›
    using *assms* **by** (*meson Hintikka.Nom*)
  then have ‹*p at a in′ H*› **if** ‹*hequiv H ?b a*› **for** *a*
    using *that assms*(*1*, *3*) **unfolding** *hequiv-def* **by** (*meson Hintikka.Nom*)
  moreover have ‹*assign A H i = proj (hequiv-rel A H) ?b*›
    unfolding *assign-def* **using** *True* **by** *simp*

```
  ultimately show ?thesis
    unfolding proj-def by blast
next
  case False
  then show ?thesis
    unfolding assign-def using assms by auto
qed

lemma ur-closure′:
  assumes ‹Hintikka A H› ‹p at i in′ H› ‹∀ a. p = Nom a ∨ p = (◇ Nom a) ⟶
a ∈ A›
  shows ‹∃ a ∈ assign A H i. p at a in′ H›
proof −
  obtain a where ‹a ∈ assign A H i›
    using assms(1) ex-assignment by fast
  then show ?thesis
    using assms ur-closure[where i=i] by blast
qed

lemma mem-hequiv-rel: ‹a ∈ proj (hequiv-rel A H) b ⟹ a ∈ A›
  unfolding proj-def by blast

lemma hequiv-proj:
  assumes ‹Hintikka A H›
    ‹Nom a at i in′ H› ‹a ∈ A› ‹Nom b at i in′ H› ‹b ∈ A›
  shows ‹proj (hequiv-rel A H) a = proj (hequiv-rel A H) b›
proof −
  have ‹equiv (names H ∩ A) (hequiv-rel A H)›
    using assms(1) hequiv-rel by fast
  moreover have ‹{a, b} ⊆ names H ∩ A›
    using assms(1−5) nominal-in-names by fastforce
  moreover have ‹Nom b at a in′ H›
    using assms(1−2, 4−5) Hintikka.Nom by fast
  then have ‹hequiv H a b›
    unfolding hequiv-def by simp
  ultimately show ?thesis
    by (simp add: proj-iff)
qed

lemma hequiv-proj-opening:
  assumes ‹Hintikka A H› ‹Nom a at i in′ H› ‹a ∈ A› ‹i ∈ A›
  shows ‹proj (hequiv-rel A H) a = proj (hequiv-rel A H) i›
  using hequiv-proj assms by fastforce

lemma assign-proj-refl:
  assumes ‹Hintikka A H› ‹Nom i at i in′ H› ‹i ∈ A›
  shows ‹assign A H i = proj (hequiv-rel A H) i›
proof −
  let ?a = ‹SOME a. a ∈ A ∧ Nom a at i in′ H›
```

```
  have ‹∃ a. a ∈ A ∧ Nom a at i in' H›
    using assms(2−3) by fast
  with someI-ex have ∗: ‹?a ∈ A ∧ Nom ?a at i in' H› .
  then have ‹assign A H i = proj (hequiv-rel A H) ?a›
    unfolding assign-def by auto
  then show ?thesis
    unfolding assign-def
    using hequiv-proj ∗ assms by fast
qed

lemma assign-named:
  assumes ‹Hintikka A H› ‹i ∈ proj (hequiv-rel A H) a›
  shows ‹i ∈ names H›
  using assms unfolding proj-def by simp (meson hequiv-names hequiv-sym')

lemma assign-unique:
  assumes ‹Hintikka A H› ‹a ∈ assign A H i›
  shows ‹assign A H a = assign A H i›
proof (cases ‹∃ b. b ∈ A ∧ Nom b at i in' H›)
  case True
  let ?b = ‹SOME b. b ∈ A ∧ Nom b at i in' H›
  have ∗: ‹?b ∈ A ∧ Nom ?b at i in' H›
    using someI-ex True .

  have ∗∗: ‹assign A H i = proj (hequiv-rel A H) ?b›
    unfolding assign-def using True by simp
  moreover from this have ‹Nom a at a in' H›
    using assms assign-named unfolding names-def by fastforce
  ultimately have ‹assign A H a = proj (hequiv-rel A H) a›
    using assms assign-proj-refl mem-hequiv-rel by fast
  with ∗∗ show ?thesis
    unfolding proj-def using assms
    by simp (meson hequiv-sym' hequiv-trans)
next
  case False
  then have ‹assign A H i = {i}›
    unfolding assign-def by auto
  then have ‹a = i›
    using assms(2) by simp
  then show ?thesis
    by simp
qed

lemma assign-val:
  assumes
    ‹Hintikka A H› ‹Pro x at a in' H› ‹(¬ Pro x) at i in' H›
    ‹a ∈ assign A H i› ‹i ∈ names H›
  shows False
  using assms Hintikka.ProP ur-closure by fastforce
```

```
lemma Hintikka-model:
  assumes ‹Hintikka A H›
  shows
    ‹p at i in' H ⟹ nominals p ⊆ A ⟹
        Model (reach A H) (val H), assign A H, assign A H i ⊨ p›
    ‹(¬ p) at i in' H ⟹ nominals p ⊆ A ⟹
      ¬ Model (reach A H) (val H), assign A H, assign A H i ⊨ p›
proof (induct p arbitrary: i)
  fix i
  case (Pro x)
  assume ‹Pro x at i in' H›
  then show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ Pro x›
    using assms(1) ur-closure' unfolding val-def by fastforce
next
  fix i
  case (Pro x)
  assume ‹(¬ Pro x) at i in' H›
  then have ‹∄ a. a ∈ assign A H i ∧ Pro x at a in' H›
    using assms(1) assign-val unfolding names-def by fast
  then have ‹¬ val H (assign A H i) x›
    unfolding proj-def val-def hequiv-def by simp
  then show ‹¬ Model (reach A H) (val H), assign A H, assign A H i ⊨ Pro x›
    by simp
next
  fix i
  case (Nom a)
  assume *: ‹Nom a at i in' H› ‹nominals (Nom a) ⊆ A›

  let ?b = ‹SOME b. b ∈ A ∧ Nom b at i in' H›
  let ?c = ‹SOME b. b ∈ A ∧ Nom b at a in' H›

  have ‹a ∈ A›
    using *(2) by simp
  then have ‹∃ b. b ∈ A ∧ Nom b at i in' H›
    using * by fast
  with someI-ex have b: ‹?b ∈ A ∧ Nom ?b at i in' H› .
  then have ‹assign A H i = proj (hequiv-rel A H) ?b›
    unfolding assign-def by auto
  also have ‹proj (hequiv-rel A H) ?b = proj (hequiv-rel A H) a›
    using hequiv-proj assms(1) b * ‹a ∈ A› by fast

  also have ‹Nom a at a in' H›
    using * ‹a ∈ A› assms(1) Hintikka.Nom by fast
  then have ‹∃ c. c ∈ A ∧ Nom c at a in' H›
    using ‹a ∈ A› by blast
  with someI-ex have c: ‹?c ∈ A ∧ Nom ?c at a in' H› .
  then have ‹assign A H a = proj (hequiv-rel A H) ?c›
    unfolding assign-def by auto
```

```
  then have ‹proj (hequiv-rel A H) a = assign A H a›
    using hequiv-proj-opening assms(1) ‹a ∈ A› c by fast

  finally have ‹assign A H i = assign A H a› .
  then show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ Nom a›
    by simp
next
  fix i
  case (Nom a)
  assume *: ‹(¬ Nom a) at i in' H› ‹nominals (Nom a) ⊆ A›
  then have ‹a ∈ A›
    by simp

  have ‹hequiv H a a›
    using hequiv-refl * nominal-in-names assms(1) by fastforce
  obtain j where j: ‹j ∈ assign A H i› ‹(¬ Nom a) at j in' H›
    using ur-closure' assms(1) * by fastforce
  then have ‹¬ Nom a at j in' H›
    using assms(1) Hintikka.NomP by fast

  moreover have ‹∀ b ∈ assign A H a. Nom a at b in' H›
    using assms ‹a ∈ A› ‹hequiv H a a› ur-closure unfolding hequiv-def by fast
  ultimately have ‹assign A H a ≠ assign A H i›
    using j by blast
  then show ‹¬ Model (reach A H) (val H), assign A H, assign A H i ⊨ Nom a›
    by simp
next
  fix i
  case (Neg p)
  moreover assume ‹(¬ p) at i in' H› ‹nominals (¬ p) ⊆ A›
  ultimately show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ ¬
p›
    by simp
next
  fix i
  case (Neg p)
  moreover assume *: ‹(¬ ¬ p) at i in' H›
  then have ‹p at i in' H›
    using assms(1) Hintikka.NegN by fast
  moreover assume ‹nominals (¬ p) ⊆ A›
  moreover from this * have ‹∀ a. p = (◇ Nom a) ⟶ a ∈ A›
    by auto
  ultimately show ‹¬ Model (reach A H) (val H), assign A H, assign A H i ⊨
¬ p›
    using assms(1) by auto
next
  fix i
  case (Dis p q)
  moreover assume *: ‹(p ∨ q) at i in' H›
```

```
  then have ‹p at i in' H ∨ q at i in' H›
    using assms(1) Hintikka.DisP by fast
  moreover assume ‹nominals (p ∨ q) ⊆ A›
  moreover from this * have ‹∀ a. p = (◊ Nom a) ⟶ a ∈ A› ‹∀ a. q = (◊ Nom
a) ⟶ a ∈ A›
    by auto
  ultimately show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ (p
∨ q)›
    by simp metis
next
  fix i
  case (Dis p q)
  moreover assume *: ‹(¬ (p ∨ q)) at i in' H›
  then have ‹(¬ p) at i in' H› ‹(¬ q) at i in' H›
    using assms(1) Hintikka.DisN by fast+
  moreover assume ‹nominals (p ∨ q) ⊆ A›
  moreover from this * have ‹∀ a. p = (◊ Nom a) ⟶ a ∈ A› ‹∀ a. q = (◊ Nom
a) ⟶ a ∈ A›
    by auto
  ultimately show ‹¬ Model (reach A H) (val H), assign A H, assign A H i ⊨
(p ∨ q)›
    by auto
next
  fix i
  case (Dia p)
  assume *: ‹(◊ p) at i in' H› ‹nominals (◊ p) ⊆ A›
  with * have p: ‹∀ a. p = (◊ Nom a) ⟶ a ∈ A›
    by auto

  show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ ◊ p›
  proof (cases ‹∃ j. p = Nom j›)
    case True
    then obtain j where j: ‹p = Nom j› ‹j ∈ A›
      using *(2) by auto
    then obtain a where a: ‹a ∈ assign A H i› ‹(◊ Nom j) at a in' H›
      using ur-closure' assms(1) ‹(◊ p) at i in' H› by fast

    from j have ‹(◊ Nom j) at i in' H›
      using *(1) by simp
    then have ‹(◊ Nom j) at a in' H›
      using ur-closure assms(1) a(2) by fast
    then have ‹assign A H j ∈ reach A H (assign A H i)›
      unfolding reach-def using a(1) by fast
    then show ?thesis
      using j(1) by simp
  next
    case False
    then obtain a where a: ‹a ∈ assign A H i› ‹(◊ p) at a in' H›
      using ur-closure' assms(1) ‹(◊ p) at i in' H› by fast
```

```
    then have ‹∃j. (◇ Nom j) at a in' H ∧ (@ j p) at a in' H›
      using False assms ‹(◇ p) at i in' H› by (meson Hintikka.DiaP)
    then obtain j where j: ‹(◇ Nom j) at a in' H› ‹(@ j p) at a in' H›
      by blast

    from j(2) have ‹p at j in' H›
      using assms(1) Hintikka.SatP by fast
    then have ‹Model (reach A H) (val H), assign A H, assign A H j ⊨ p›
      using Dia p *(2) by simp
    moreover have ‹assign A H j ∈ reach A H (assign A H i)›
      unfolding reach-def using a(1) j(1) by blast
    ultimately show ?thesis
      by auto
  qed
next
  fix i
  case (Dia p)
  assume *: ‹(¬ (◇ p)) at i in' H› ‹nominals (◇ p) ⊆ A›
  then obtain a where a: ‹a ∈ assign A H i› ‹(¬ (◇ p)) at a in' H›
    using ur-closure' assms(1) by fast
  {
    fix j b
    assume ‹(◇ Nom j) at b in' H› ‹b ∈ assign A H a›
    moreover have ‹(¬ (◇ p)) at b in' H›
      using a(2) assms(1) calculation(2) ur-closure by fast
    ultimately have ‹(¬ (@ j p)) at b in' H›
      using assms(1) Hintikka.DiaN by fast
    then have ‹(¬ p) at j in' H›
      using assms(1) Hintikka.SatN by fast
    then have ‹¬ Model (reach A H) (val H), assign A H, assign A H j ⊨ p›
      using Dia *(2) by simp
  }
  then have ‹¬ Model (reach A H) (val H), assign A H, assign A H a ⊨ ◇ p›
    unfolding reach-def by auto
  moreover have ‹assign A H a = assign A H i›
    using assms(1) a assign-unique by fast
  ultimately show ‹¬ Model (reach A H) (val H), assign A H, assign A H i ⊨
◇ p›
    by simp
next
  fix i
  case (Sat j p)
  assume ‹(@ j p) at i in' H› ‹nominals (@ j p) ⊆ A›
  moreover from this have ‹∀ a. p = (◇ Nom a) ⟶ a ∈ A›
    by auto
  moreover have ‹p at j in' H› if ‹∃ a. (@ j p) at a in' H›
    using that assms(1) Hintikka.SatP by fast
  ultimately show ‹Model (reach A H) (val H), assign A H, assign A H i ⊨ @
j p›
```

using *Sat* **by** *auto*
next
fix *i*
case (*Sat j p*)
assume ‹(¬ (@ *j p*)) *at i in′ H*› ‹*nominals* (@ *j p*) ⊆ *A*›
moreover from *this* **have** ‹∀ *a*. *p* = (◇ *Nom a*) ⟶ *a* ∈ *A*›
by *auto*
moreover have ‹(¬ *p*) *at j in′ H*› **if** ‹∃ *a*. (¬ (@ *j p*)) *at a in′ H*›
using *that assms*(*1*) *Hintikka.SatN* **by** *fast*
ultimately show ‹¬ *Model* (*reach A H*) (*val H*), *assign A H*, *assign A H i* ⊨ @ *j p*›
using *Sat* **by** *fastforce*
qed

11.2 Lindenbaum-Henkin

A set of blocks is consistent if no finite subset can be derived. Given a consistent set of blocks we are going to extend it to be saturated and maximally consistent and show that is then Hintikka. All definitions are with respect to the set of allowed nominals.

definition *consistent* :: ‹′*b set* ⇒ (′*a*, ′*b*) *block set* ⇒ *bool*› **where**
‹*consistent A S* ≡ ∄ *S*′. *set S*′ ⊆ *S* ∧ *A* ⊢ *S*′›

instance *fm* :: (*countable*, *countable*) *countable*
by *countable-datatype*

definition *proper-dia* :: ‹(′*a*, ′*b*) *fm* ⇒ (′*a*, ′*b*) *fm option*› **where**
‹*proper-dia p* ≡ *case p of* (◇ *p*) ⇒ (*if* ∄ *a*. *p* = *Nom a then Some p else None*) | - ⇒ *None*›

lemma *proper-dia*: ‹*proper-dia p* = *Some q* ⟹ *p* = (◇ *q*) ∧ (∄ *a*. *q* = *Nom a*)›
unfolding *proper-dia-def* **by** (*cases p*) (*simp-all*, *metis option.discI option.inject*)

The following function witnesses each ◇ *p* in a fresh world.

primrec *witness-list* :: ‹(′*a*, ′*b*) *fm list* ⇒ ′*b set* ⇒ (′*a*, ′*b*) *fm list*› **where**
‹*witness-list* [] - = []›
| ‹*witness-list* (*p* # *ps*) *used* =
(*case proper-dia p of*
None ⇒ *witness-list ps used*
| *Some q* ⇒
let i = *SOME i*. *i* ∉ *used*
in (@ *i q*) # (◇ *Nom i*) # *witness-list ps* ({*i*} ∪ *used*))›

primrec *witness* :: ‹(′*a*, ′*b*) *block* ⇒ ′*b set* ⇒ (′*a*, ′*b*) *block*› **where**
‹*witness* (*ps*, *a*) *used* = (*witness-list ps used*, *a*)›

lemma *witness-list*:
‹*proper-dia p* = *Some q* ⟹ *witness-list* (*p* # *ps*) *used* =

(*let i* = *SOME i*. *i* ∉ *used*
in (**@** *i q*) # (◇ *Nom i*) # *witness-list ps* ({*i*} ∪ *used*))›
by *simp*

primrec *extend* ::
‹′*b set* ⇒ (′*a*, ′*b*) *block set* ⇒ (*nat* ⇒ (′*a*, ′*b*) *block*) ⇒ *nat* ⇒ (′*a*, ′*b*) *block set*›
where
‹*extend A S f 0* = *S*›
| ‹*extend A S f* (*Suc n*) =
(*if* ¬ *consistent A* ({*f n*} ∪ *extend A S f n*)
then extend A S f n
else
let used = *A* ∪ (⋃ *block* ∈ {*f n*} ∪ *extend A S f n*. *block-nominals block*)
in {*f n*, *witness* (*f n*) *used*} ∪ *extend A S f n*)›

definition *Extend* ::
‹′*b set* ⇒ (′*a*, ′*b*) *block set* ⇒ (*nat* ⇒ (′*a*, ′*b*) *block*) ⇒ (′*a*, ′*b*) *block set*› **where**
‹*Extend A S f* ≡ (⋃ *n*. *extend A S f n*)›

lemma *extend-chain*: ‹*extend A S f n* ⊆ *extend A S f* (*Suc n*)›
by *auto*

lemma *extend-mem*: ‹*S* ⊆ *extend A S f n*›
by (*induct n*) *auto*

lemma *Extend-mem*: ‹*S* ⊆ *Extend A S f*›
unfolding *Extend-def* **using** *extend-mem* **by** *fast*

11.2.1 Consistency

lemma *split-list*:
‹*set A* ⊆ {*x*} ∪ *X* ⟹ *x* ∈. *A* ⟹ ∃ *B*. *set* (*x* # *B*) = *set A* ∧ *x* ∉ *set B*›
by *simp* (*metis Diff-insert-absorb mk-disjoint-insert set-removeAll*)

lemma *consistent-drop-single*:
fixes *a* :: ′*b*
assumes
inf: ‹*infinite* (*UNIV* :: ′*b set*)› **and**
fin: ‹*finite A*› **and**
cons: ‹*consistent A* ({(*p* # *ps*, *a*)} ∪ *S*)›
shows ‹*consistent A* ({(*ps*, *a*)} ∪ *S*)›
unfolding *consistent-def*
proof
assume ‹∃ *S*′. *set S*′ ⊆ {(*ps*, *a*)} ∪ *S* ∧ *A* ⊢ *S*′›
then obtain *S*′ *n* **where** ‹*set S*′ ⊆ {(*ps*, *a*)} ∪ *S*› ‹(*ps*, *a*) ∈. *S*′› ‹*A*, *n* ⊢ *S*′›
using *assms* **unfolding** *consistent-def* **by** *blast*
then obtain *S*′′ **where** ‹*set* ((*ps*, *a*) # *S*′′) = *set S*′› ‹(*ps*, *a*) ∉ *set S*′′›
using *split-list* **by** *metis*
then have ‹*A* ⊢ (*ps*, *a*) # *S*′′›

using *inf fin STA-struct* ‹A, $n \vdash S'$› **by** *blast*
then have ‹$A \vdash (p \# ps,\ a) \# S''$›
using *inf fin STA-struct-block*[**where** ps'=‹$p \# ps$›] **by** *fastforce*
moreover have ‹*set* $((p \# ps,\ a) \# S'') \subseteq \{(p \# ps,\ a)\} \cup S$›
using ‹$(ps,\ a) \notin$ *set* S''› ‹*set* $((ps,\ a) \# S'')$ = *set* S'› ‹*set* $S' \subseteq \{(ps,\ a)\} \cup S$›
by *auto*
ultimately show *False*
using *cons* **unfolding** *consistent-def* **by** *blast*
qed

lemma *consistent-drop-block*: ‹*consistent* A ({*block*} $\cup S$) $\Longrightarrow$ *consistent* A S›
unfolding *consistent-def* **by** *blast*

lemma *inconsistent-weaken*: ‹$\neg$ *consistent* A $S \Longrightarrow S \subseteq S' \Longrightarrow \neg$ *consistent* A S'›
unfolding *consistent-def* **by** *blast*

lemma *finite-nominals-set*: ‹*finite* $S \Longrightarrow$ *finite* ($\bigcup$ *block* $\in S$. *block-nominals block*)›
by (*induct S rule*: *finite-induct*) (*simp-all add*: *finite-block-nominals*)

lemma *witness-list-used*:
fixes i :: $'b$
assumes *inf*: ‹*infinite* (*UNIV* :: $'b$ *set*)› **and** ‹*finite used*› ‹$i \notin$ *list-nominals ps*›
shows ‹$i \notin$ *list-nominals* (*witness-list ps* ({i} $\cup$ *used*))›
using *assms*(*2*−)
proof (*induct ps arbitrary*: *used*)
case (*Cons p ps*)
then show *?case*
proof (*cases* ‹*proper-dia p*›)
case (*Some q*)
let *?j* = ‹*SOME j*. $j \notin$ {i} $\cup$ *used*›
have ‹*finite* ({i} $\cup$ *used*)›
using ‹*finite used*› **by** *simp*
then have ‹$\exists j$. $j \notin$ {i} $\cup$ *used*›
using *inf ex-new-if-finite* **by** *metis*
then have *j*: ‹*?j* $\notin$ {i} $\cup$ *used*›
using *someI-ex* **by** *metis*

have ‹*witness-list* ($p \# ps$) ({i} $\cup$ *used*) =
(**@** *?j q*) # ($\Diamond$ *Nom ?j*) # *witness-list ps* ({*?j*} $\cup$ ({i} $\cup$ *used*))›
using *Some witness-list* **by** *metis*
then have $*$: ‹*list-nominals* (*witness-list* ($p \# ps$) ({i} $\cup$ *used*)) =
{*?j*} $\cup$ *nominals q* $\cup$ *list-nominals* (*witness-list ps* ({*?j*} $\cup$ ({i} $\cup$ *used*)))›
by *simp*

have ‹*finite* ({*?j*} $\cup$ *used*)›
using ‹*finite used*› **by** *simp*
moreover have ‹$i \notin$ *list-nominals ps*›
using ‹$i \notin$ *list-nominals* ($p \# ps$)› **by** *simp*
ultimately have ‹$i \notin$ *list-nominals* (*witness-list ps* ({i} $\cup$ ({*?j*} $\cup$ *used*)))›

```
      using Cons by metis
    moreover have ‹{i} ∪ ({?j} ∪ used) = {?j} ∪ ({i} ∪ used)›
      by blast
    moreover have ‹i ≠ ?j›
      using j by auto
    ultimately have ‹i ∈ list-nominals (witness-list (p # ps) ({i} ∪ used)) ⟷ i
∈ nominals q›
      using ∗ by simp
    moreover have ‹i ∉ nominals q›
      using Cons(3) Some proper-dia by fastforce
    ultimately show ?thesis
      by blast
  qed simp
qed simp

lemma witness-used:
  fixes i :: 'b
  assumes inf: ‹infinite (UNIV :: 'b set)› and
    ‹finite used› ‹i ∉ block-nominals block›
  shows ‹i ∉ block-nominals (witness block ({i} ∪ used))›
  using assms witness-list-used by (induct block) fastforce

lemma consistent-witness-list:
  fixes a :: 'b
  assumes inf: ‹infinite (UNIV :: 'b set)› and ‹consistent A S›
    ‹(ps, a) ∈ S› ‹finite used› ‹A ∪ ⋃ (block-nominals ‘ S) ⊆ used›
  shows ‹consistent A ({(witness-list ps used, a)} ∪ S)›
  using assms(2−)
proof (induct ps arbitrary: used S)
  case Nil
  then have ‹{(witness-list [] used, a)} ∪ S = S›
    by auto
  moreover have ‹finite {}› ‹{} ∩ used = {}›
    by simp-all
  ultimately show ?case
    using ‹consistent A S› by simp
next
  case (Cons p ps)
  have fin: ‹finite A›
    using assms(4−5) finite-subset by fast
  have ‹{(p # ps, a)} ∪ S = S›
    using ‹(p # ps, a) ∈ S› by blast
  then have ‹consistent A ({(p # ps, a)} ∪ S)›
    using ‹consistent A S› by simp
  then have ‹consistent A ({(ps, a)} ∪ S)›
    using inf fin consistent-drop-single by fast
  moreover have ‹(ps, a) ∈ {(ps, a)} ∪ S›
    by simp
  moreover have ‹A ∪ ⋃ (block-nominals ‘ ({(ps, a)} ∪ S)) ⊆ extra ∪ used› for
```

extra
  using ‹(p # ps, a) ∈ S› ‹A ∪ ⋃ (block-nominals ‘ S) ⊆ used› **by** *fastforce*
 moreover have ‹finite (extra ∪ used)› **if** ‹finite extra› **for** *extra*
  using *that* ‹finite used› **by** *blast*
 ultimately have *cons*:
  ‹consistent A ({(witness-list ps (extra ∪ used), a)} ∪ ({(ps, a)} ∪ S))›
  if ‹finite extra› **for** *extra*
  using *that Cons* **by** *metis*

 show *?case*
 proof (*cases* ‹proper-dia p›)
  case *None*
  then have ‹witness-list (p # ps) used = witness-list ps used›
   by *auto*
  moreover have ‹consistent A ({(witness-list ps used, a)} ∪ ({(ps, a)} ∪ S))›
   using *cons*[**where** *extra*=‹{}›] **by** *simp*
  then have ‹consistent A ({(witness-list ps used, a)} ∪ S)›
   using *consistent-drop-block*[**where** *block*=‹(ps, a)›] **by** *auto*
  ultimately show *?thesis*
   by *simp*
 next
  case (*Some q*)
  let *?i* = ‹SOME i. i ∉ used›
  have ‹∃ i. i ∉ used›
   using *ex-new-if-finite inf* ‹finite used› **.**
  with *someI-ex* **have** ‹?i ∉ used› **.**
  then have *i*: ‹?i ∉ ⋃ (block-nominals ‘ S)›
   using *Cons* **by** *auto*
  then have ‹?i ∉ block-nominals (p # ps, a)›
   using *Cons* **by** *blast*

  let *?tail* = ‹witness-list ps ({?i} ∪ used)›

  have ‹consistent A ({(?tail, a)} ∪ ({(ps, a)} ∪ S))›
   using *cons*[**where** *extra*=‹{?i}›] **by** *blast*
  then have ∗: ‹consistent A ({(?tail, a)} ∪ S)›
   using *consistent-drop-block*[**where** *block*=‹(ps, a)›] **by** *simp*

  have ‹witness-list (p # ps) used = (@ ?i q) # (◇ Nom ?i) # ?tail›
   using *Some witness-list* **by** *metis*
  moreover have ‹consistent A ({((@ ?i q) # (◇ Nom ?i) # ?tail, a)} ∪ S)›
   unfolding *consistent-def*
  proof
   assume ‹∃ S′. set S′ ⊆ {((@ ?i q) # (◇ Nom ?i) # ?tail, a)} ∪ S ∧ A ⊢ S′›
   then obtain *S′ n* **where**
    ‹A, n ⊢ S′› **and** *S′*:
    ‹set S′ ⊆ {((@ ?i q) # (◇ Nom ?i) # ?tail, a)} ∪ S›
    ‹((@ ?i q) # (◇ Nom ?i) # ?tail, a) ∈. S′›
    using ∗ **unfolding** *consistent-def* **by** *blast*

```
      then obtain S'' where S'':
        ‹set (((@ ?i q) # (◇ Nom ?i) # ?tail, a) # S'') = set S'›
        ‹((@ ?i q) # (◇ Nom ?i) # ?tail, a) ∉ set S''›
        using split-list[where x=‹((@ ?i q) # (◇ Nom ?i) # ?tail, a)›] by blast
      then have ‹A ⊢ ((@ ?i q) # (◇ Nom ?i) # ?tail, a) # S''›
        using inf ‹finite A› STA-struct ‹A, n ⊢ S'› by blast
      moreover have ‹set (((@ ?i q) # (◇ Nom ?i) # ?tail, a) # S'') ⊆
        set (((@ ?i q) # (◇ Nom ?i) # ?tail, a) # (p # ps, a) # S'')›
        by auto
      ultimately have **: ‹A ⊢ ((@ ?i q) # (◇ Nom ?i) # ?tail, a) # (p # ps, a) # S''›
        using inf ‹finite A› STA-struct by blast

      have ‹?i ∉ block-nominals (?tail, a)›
        using inf ‹finite used› ‹?i ∉ block-nominals (p # ps, a)› witness-used by fastforce
      moreover have ‹?i ∉ branch-nominals S''›
        unfolding branch-nominals-def using i S' S'' by auto
      ultimately have ‹?i ∉ branch-nominals ((?tail, a) # (p # ps, a) # S'')›
        using ‹?i ∉ block-nominals (p # ps, a)› unfolding branch-nominals-def
        by simp
      then have ‹?i ∉ A ∪ branch-nominals ((?tail, a) # (p # ps, a) # S'')›
        using ‹?i ∉ used› Cons.prems(4) by blast

      moreover have ‹∄ a. q = Nom a›
        using Some proper-dia by blast
      moreover have ‹(p # ps, a) ∈. (?tail, a) # (p # ps, a) # S''›
        by simp
      moreover have ‹p = (◇ q)›
        using Some proper-dia by blast
      then have ‹(◇ q) on (p # ps, a)›
        by simp
      ultimately have ‹A ⊢ (?tail, a) # (p # ps, a) # S''›
        using ** ‹finite A› DiaP'' by fast
      moreover have ‹set ((p # ps, a) # S'') ⊆ S›
        using Cons(3) S' S'' by auto
      ultimately show False
        using * unfolding consistent-def by (simp add: subset-Un-eq)
    qed
    ultimately show ?thesis
      by simp
  qed
qed

lemma consistent-witness:
  fixes block :: ‹('a, 'b) block›
  assumes ‹infinite (UNIV :: 'b set)›
    ‹consistent A S› ‹finite (⋃ (block-nominals ‘ S))› ‹block ∈ S› ‹finite A›
  shows ‹consistent A ({witness block (A ∪ ⋃ (block-nominals ‘ S))} ∪ S)›
```

using *assms consistent-witness-list* **by** (*cases block*) *fastforce*

lemma *consistent-extend*:
fixes *S* :: ‹(′*a*, ′*b*) *block set*›
assumes *inf*: ‹*infinite* (*UNIV* :: ′*b set*)› **and** *fin*: ‹*finite A*› **and**
‹*consistent A* (*extend A S f n*)› ‹*finite* (⋃ (*block-nominals* ‘ *extend A S f n*))›
shows ‹*consistent A* (*extend A S f* (*Suc n*))›
proof (*cases* ‹*consistent A* ({*f n*} ∪ *extend A S f n*)›)
case *True*
let *?used* = ‹*A* ∪ (⋃*block* ∈ {*f n*} ∪ *extend A S f n*. *block-nominals block*)›
have ∗: ‹*extend A S f* (*n* + *1*) = {*f n*, *witness* (*f n*) *?used*} ∪ *extend A S f n*›
using *True* **by** *simp*

have ‹*consistent A* ({*f n*} ∪ *extend A S f n*)›
using *True* **by** *simp*
moreover have ‹*finite* ((⋃ (*block-nominals* ‘ ({*f n*} ∪ *extend A S f n*))))›
using ‹*finite* (⋃ (*block-nominals* ‘ *extend A S f n*))› *finite-nominals-set* **by** *force*
moreover have ‹*f n* ∈ {*f n*} ∪ *extend A S f n*›
by *simp*
ultimately have ‹*consistent A* ({*witness* (*f n*) *?used*} ∪ ({*f n*} ∪ *extend A S f n*))›
using *inf fin consistent-witness* **by** *blast*
then show *?thesis*
using ∗ **by** *simp*
next
case *False*
then show *?thesis*
using *assms*(*3*) **by** *simp*
qed

lemma *finite-nominals-extend*:
assumes ‹*finite* (⋃ (*block-nominals* ‘ *S*))›
shows ‹*finite* (⋃ (*block-nominals* ‘ *extend A S f n*))›
using *assms* **by** (*induct n*) (*auto simp add*: *finite-block-nominals*)

lemma *consistent-extend′*:
fixes *S* :: ‹(′*a*, ′*b*) *block set*›
assumes ‹*infinite* (*UNIV* :: ′*b set*)› ‹*finite A*› ‹*consistent A S*› ‹*finite* (⋃ (*block-nominals* ‘ *S*))›
shows ‹*consistent A* (*extend A S f n*)›
using *assms*
proof (*induct n*)
case (*Suc n*)
then show *?case*
by (*metis consistent-extend finite-nominals-extend*)
qed *simp*

lemma *UN-finite-bound*:

assumes ‹*finite A*› ‹$A \subseteq (\bigcup n.\ f\ n)$›
shows ‹$\exists\, m :: nat.\ A \subseteq (\bigcup n \le m.\ f\ n)$›
using *assms*
proof (*induct A rule*: *finite-induct*)
case (*insert x A*)
then obtain *m* **where** ‹$A \subseteq (\bigcup n \le m.\ f\ n)$›
by *fast*
then have ‹$A \subseteq (\bigcup n \le (m + k).\ f\ n)$› **for** *k*
by *fastforce*
moreover obtain m' **where** ‹$x \in f\ m'$›
using *insert*(*4*) **by** *blast*
ultimately have ‹$\{x\} \cup A \subseteq (\bigcup n \le m + m'.\ f\ n)$›
by *auto*
then show *?case*
by *blast*
qed *simp*

lemma *extend-bound*: ‹$(\bigcup n \le m.\ extend\ A\ S\ f\ n) = extend\ A\ S\ f\ m$›
proof (*induct m*)
case (*Suc m*)
have ‹$\bigcup\ (extend\ A\ S\ f\ `\ \{..Suc\ m\}) = \bigcup\ (extend\ A\ S\ f\ `\ \{..m\}) \cup extend\ A\ S\ f$ $(Suc\ m)$›
using *atMost-Suc* **by** *auto*
also have ‹$\ldots = extend\ A\ S\ f\ m \cup extend\ A\ S\ f\ (Suc\ m)$›
using *Suc* **by** *blast*
also have ‹$\ldots = extend\ A\ S\ f\ (Suc\ m)$›
using *extend-chain* **by** *blast*
finally show *?case*
by *simp*
qed *simp*

lemma *consistent-Extend*:
fixes *S* :: ‹('*a*, '*b*) *block set*›
assumes *inf*: ‹*infinite* (*UNIV* :: '*b set*)› **and** ‹*finite A*›
‹*consistent A S*› ‹$finite\ (\bigcup\ (block\text{-}nominals\ `\ S))$›
shows ‹*consistent A* (*Extend A S f*)›
unfolding *Extend-def*
proof (*rule ccontr*)
assume ‹$\neg\ consistent\ A\ (\bigcup\ (range\ (extend\ A\ S\ f)))$›
then obtain S' *n* **where** ∗:
‹$A,\ n \vdash S'$›
‹$set\ S' \subseteq (\bigcup n.\ extend\ A\ S\ f\ n)$›
unfolding *consistent-def* **by** *blast*
moreover have ‹$finite\ (set\ S')$›
by *simp*
ultimately obtain *m* **where** ‹$set\ S' \subseteq (\bigcup n \le m.\ extend\ A\ S\ f\ n)$›
using *UN-finite-bound* **by** *metis*
then have ‹$set\ S' \subseteq extend\ A\ S\ f\ m$›
using *extend-bound* **by** *blast*

moreover have ‹*consistent A* (*extend A S f m*)›
using *assms consistent-extend′* **by** *blast*
ultimately show *False*
unfolding *consistent-def* **using** ∗ **by** *blast*
qed

11.2.2 Maximality

A set of blocks is maximally consistent if any proper extension makes it inconsistent.

definition *maximal* :: ‹′*b set* ⇒ (′*a*, ′*b*) *block set* ⇒ *bool*› **where**
‹*maximal A S* ≡ *consistent A S* ∧ (∀ *block*. *block* ∉ *S* ⟶ ¬ *consistent A* ({*block*} ∪ *S*))›

lemma *extend-not-mem*:
‹*f n* ∉ *extend A S f* (*Suc n*) ⟹ ¬ *consistent A* ({*f n*} ∪ *extend A S f n*)›
by (*metis Un-insert-left extend.simps*(*2*) *insertI1*)

lemma *maximal-Extend*:
fixes *S* :: ‹(′*a*, ′*b*) *block set*›
assumes *inf*: ‹*infinite* (*UNIV* :: ′*b set*)› **and** ‹*finite A*›
‹*consistent A S*› ‹*finite* (⋃ (*block-nominals* ‘ *S*))› ‹*surj f*›
shows ‹*maximal A* (*Extend A S f*)›
proof (*rule ccontr*)
assume ‹¬ *maximal A* (*Extend A S f*)›
then obtain *block* **where**
‹*block* ∉ *Extend A S f*› ‹*consistent A* ({*block*} ∪ *Extend A S f*)›
unfolding *maximal-def* **using** *assms consistent-Extend* **by** *metis*
obtain *n* **where** *n*: ‹*f n* = *block*›
using ‹*surj f*› **unfolding** *surj-def* **by** *metis*
then have ‹*block* ∉ *extend A S f* (*Suc n*)›
using ‹*block* ∉ *Extend A S f*› *extend-chain* **unfolding** *Extend-def* **by** *blast*
then have ‹¬ *consistent A* ({*block*} ∪ *extend A S f n*)›
using *n extend-not-mem* **by** *blast*
moreover have ‹*block* ∉ *extend A S f n*›
using ‹*block* ∉ *extend A S f* (*Suc n*)› *extend-chain* **by** *blast*
then have ‹{*block*} ∪ *extend A S f n* ⊆ {*block*} ∪ *Extend A S f*›
unfolding *Extend-def* **by** *blast*
ultimately have ‹¬ *consistent A* ({*block*} ∪ *Extend A S f*)›
using *inconsistent-weaken* **by** *blast*
then show *False*
using ‹*consistent A* ({*block*} ∪ *Extend A S f*)› **by** *simp*
qed

11.2.3 Saturation

A set of blocks is saturated if every ◊ *p* is witnessed.

definition *saturated* :: ‹(′*a*, ′*b*) *block set* ⇒ *bool*› **where**

‹saturated S ≡ ∀ p i. (◇ p) at i in' S ⟶ (∄ a. p = Nom a) ⟶
(∃ j. (**@** j p) at i in' S ∧ (◇ Nom j) at i in' S)›

lemma witness-list-append:
‹∃ extra. witness-list (ps @ qs) used = witness-list ps used @ witness-list qs (extra
∪ used)›
proof (induct ps arbitrary: used)
case Nil
then show ?case
by (metis Un-absorb append-self-conv2 witness-list.simps(1))
next
case (Cons p ps)
show ?case
proof (cases ‹∃ q. proper-dia p = Some q›)
case True
let ?i = ‹SOME i. i ∉ used›
from True **obtain** q **where** q: ‹proper-dia p = Some q›
by blast
moreover have ‹(p # ps) @ qs = p # (ps @ qs)›
by simp
ultimately have
‹witness-list ((p # ps) @ qs) used = (**@** ?i q) # (◇ Nom ?i) #
witness-list (ps @ qs) ({?i} ∪ used)›
using witness-list **by** metis
then have
‹∃ extra. witness-list ((p # ps) @ qs) used = (**@** ?i q) # (◇ Nom ?i) #
witness-list ps ({?i} ∪ used) @ witness-list qs (extra ∪ ({?i} ∪ used))›
using Cons **by** metis
moreover have ‹(**@** ?i q) # (◇ Nom ?i) # witness-list ps ({?i} ∪ used) =
witness-list (p # ps) used›
using q witness-list **by** metis
ultimately have ‹∃ extra. witness-list ((p # ps) @ qs) used =
witness-list (p # ps) used @ witness-list qs (extra ∪ ({?i} ∪ used))›
by (metis append-Cons)
then have ‹∃ extra. witness-list ((p # ps) @ qs) used =
witness-list (p # ps) used @ witness-list qs (({?i} ∪ extra) ∪ used)›
by simp
then show ?thesis
by blast
qed (simp add: Cons)
qed

lemma ex-witness-list:
assumes ‹p ∈. ps› ‹proper-dia p = Some q›
shows ‹∃ i. {**@** i q, ◇ Nom i} ⊆ set (witness-list ps used)›
using ‹p ∈. ps›
proof (induct ps arbitrary: used)
case (Cons a ps)
then show ?case

```
  proof (induct ‹a = p›)
    case True
    then have
      ‹∃ i. witness-list (a # ps) used = (@ i q) # (◇ Nom i) #
        witness-list ps ({i} ∪ used)›
      using ‹proper-dia p = Some q› witness-list by metis
    then show ?case
      by auto
  next
    case False
    then have ‹∃ i. {@ i q, ◇ Nom i} ⊆ set (witness-list ps (extra ∪ used))› for
extra
      by simp
    moreover have ‹∃ extra. witness-list (a # ps) used =
        witness-list [a] used @ witness-list ps (extra ∪ used)›
      using witness-list-append[where ps=‹[-]›] by simp
    ultimately show ?case
      by fastforce
  qed
qed simp

lemma saturated-Extend:
  fixes S :: ‹('a, 'b) block set›
  assumes inf: ‹infinite (UNIV :: 'b set)› and fin: ‹finite A› and
    ‹consistent A S› ‹finite (⋃ (block-nominals ‘ S))› ‹surj f›
  shows ‹saturated (Extend A S f)›
  unfolding saturated-def
proof safe
  fix ps i p
  assume ‹(ps, i) ∈ Extend A S f› ‹(◇ p) on (ps, i)› ‹∄ a. p = Nom a›
  obtain n where n: ‹f n = (ps, i)›
    using ‹surj f› unfolding surj-def by metis

  let ?used = ‹A ∪ (⋃ block ∈ {f n} ∪ extend A S f n. block-nominals block)›

  have ‹extend A S f n ⊆ Extend A S f›
    unfolding Extend-def by auto
  moreover have ‹consistent A (Extend A S f)›
    using assms consistent-Extend by blast
  ultimately have ‹consistent A ({(ps, i)} ∪ extend A S f n)›
    using ‹(ps, i) ∈ Extend A S f› inconsistent-weaken by blast
  then have ‹extend A S f (Suc n) = {f n, witness (f n) ?used} ∪ extend A S f n›
    using n ‹(◇ p) on (ps, i)› by auto
  then have ‹witness (f n) ?used ∈ Extend A S f›
    unfolding Extend-def by blast
  then have *: ‹(witness-list ps ?used, i) ∈ Extend A S f›
    using n by simp

  have ‹(◇ p) ∈. ps›
```

```
    using ‹(◇ p) on (ps, i)› by simp
  moreover have ‹proper-dia (◇ p) = Some p›
    unfolding proper-dia-def using ‹∄ a. p = Nom a› by simp
  ultimately have ‹∃ j.
      (@ j p) on (witness-list ps ?used, i) ∧
      (◇ Nom j) on (witness-list ps ?used, i)›
    using ex-witness-list by fastforce
  then show ‹∃ j.
      (∃ qs. (qs, i) ∈ Extend A S f ∧ (@ j p) on (qs, i)) ∧
      (∃ rs. (rs, i) ∈ Extend A S f ∧ (◇ Nom j) on (rs, i))›
    using ∗ by blast
qed
```

11.3 Smullyan-Fitting

```
lemma Hintikka-Extend:
  fixes S :: ‹('a, 'b) block set›
  assumes inf: ‹infinite (UNIV :: 'b set)› and fin: ‹finite A› and
    ‹maximal A S› ‹consistent A S› ‹saturated S›
  shows ‹Hintikka A S›
  unfolding Hintikka-def
proof safe
  fix x i j ps qs rs
  assume
    ps: ‹(ps, i) ∈ S› ‹Nom j on (ps, i)› and
    qs: ‹(qs, j) ∈ S› ‹Pro x on (qs, j)› and
    rs: ‹(rs, i) ∈ S› ‹(¬ Pro x) on (rs, i)›
  then have ‹¬ A, n ⊢ [(qs, j), (ps, i), (rs, i)]› for n
    using ‹consistent A S› unfolding consistent-def by simp
  moreover have ‹A, n ⊢ [((¬ Pro x) # qs, j), (ps, i), (rs, i)]› for n
    using qs(2) Close
   by (metis (no-types, lifting) list.set-intros(1) on.simps set-subset-Cons subsetD)
  then have ‹A, n ⊢ [(qs, j), (ps, i), (rs, i)]› for n
    using ps(2) rs(2)
   by (meson Nom' fm.distinct(21) fm.simps(18) list.set-intros(1) set-subset-Cons
subsetD)
  ultimately show False
    by blast
next
  fix a i ps qs
  assume
    ps: ‹(ps, i) ∈ S› ‹Nom a on (ps, i)› and
    qs: ‹(qs, i) ∈ S› ‹(¬ Nom a) on (qs, i)›
  then have ‹¬ A , n ⊢ [(ps, i), (qs, i)]› for n
    using ‹consistent A S› unfolding consistent-def by simp
  moreover have ‹A, n ⊢ [(ps, i), (qs, i)]› for n
      using ps(2) qs(2) by (meson Close list.set-intros(1) set-subset-Cons sub-
set-code(1))
  ultimately show False
```

    by *blast*
next
  fix *p i ps*
  assume *ps*: ‹(*ps*, *i*) ∈ *S*› ‹(¬ ¬ *p*) *on* (*ps*, *i*)›
  show ‹*p at i in′ S*›
  proof (*rule ccontr*)
    assume ‹¬ *p at i in′ S*›
    then obtain *S′ n* **where**
      ‹*A*, *n* ⊢ *S′*› **and** *S′*: ‹*set S′* ⊆ {(*p* # *ps*, *i*)} ∪ *S*› **and** ‹(*p* # *ps*, *i*) ∈. *S′*›
      using ‹*maximal A S*› **unfolding** *maximal-def consistent-def*
      by (*metis insert-is-Un list.set-intros*(*1*) *on.simps subset-insert*)
    then obtain *S′′* **where** *S′′*:
      ‹*set* ((*p* # *ps*, *i*) # *S′′*) = *set S′*› ‹(*p* # *ps*, *i*) ∉ *set S′′*›
      using *split-list*[**where** *x*=‹(*p* # *ps*, *i*)›] **by** *blast*
    then have ‹*A* ⊢ (*p* # *ps*, *i*) # *S′′*›
      using *inf fin STA-struct* ‹*A*, *n* ⊢ *S′*› **by** *blast*
    then have ‹*A* ⊢ (*ps*, *i*) # *S′′*›
      using *ps* **by** (*meson Neg′ list.set-intros*(*1*))
    moreover have ‹*set* ((*ps*, *i*) # *S′′*) ⊆ *S*›
      using *S′ S′′ ps* **by** *auto*
    ultimately show *False*
      using ‹*consistent A S*› **unfolding** *consistent-def* **by** *blast*
  qed
next
  fix *p q i ps*
  assume *ps*: ‹(*ps*, *i*) ∈ *S*› ‹(*p* ∨ *q*) *on* (*ps*, *i*)› **and** ∗: ‹¬ *q at i in′ S*›
  show ‹*p at i in′ S*›
  proof (*rule ccontr*)
    assume ‹¬ *p at i in′ S*›
    then obtain *Sp′ np* **where**
      ‹*A*, *np* ⊢ *Sp′*› **and** *Sp′*: ‹*set Sp′* ⊆ {(*p* # *ps*, *i*)} ∪ *S*› **and** ‹(*p* # *ps*, *i*) ∈. *Sp′*›
      using ‹*maximal A S*› **unfolding** *maximal-def consistent-def*
      by (*metis insert-is-Un list.set-intros*(*1*) *on.simps subset-insert*)
    then obtain *Sp′′* **where** *Sp′′*:
      ‹*set* ((*p* # *ps*, *i*) # *Sp′′*) = *set Sp′*› ‹(*p* # *ps*, *i*) ∉ *set Sp′′*›
      using *split-list*[**where** *x*=‹(*p* # *ps*, *i*)›] **by** *blast*
    then have ‹*A* ⊢ (*p* # *ps*, *i*) # *Sp′′*›
      using ‹*A*, *np* ⊢ *Sp′*› *inf fin STA-struct* **by** *blast*

    obtain *Sq′ nq* **where**
      ‹*A*, *nq* ⊢ *Sq′*› **and** *Sq′*: ‹*set Sq′* ⊆ {(*q* # *ps*, *i*)} ∪ *S*› **and** ‹(*q* # *ps*, *i*) ∈. *Sq′*›
      using ∗ ‹*maximal A S*› **unfolding** *maximal-def consistent-def*
      by (*metis insert-is-Un list.set-intros*(*1*) *on.simps subset-insert*)
    then obtain *Sq′′* **where** *Sq′′*:
      ‹*set* ((*q* # *ps*, *i*) # *Sq′′*) = *set Sq′*› ‹(*q* # *ps*, *i*) ∉ *set Sq′′*›
      using *split-list*[**where** *x*=‹(*q* # *ps*, *i*)›] **by** *blast*
    then have ‹*A* ⊢ (*q* # *ps*, *i*) # *Sq′′*›
      using ‹*A*, *nq* ⊢ *Sq′*› *inf fin STA-struct* **by** *blast*

obtain S'' **where** S'': ‹*set* S'' = *set* Sp'' ∪ *set* Sq''›
by (*meson set-union*)
then have
‹*set* ((*p* # *ps*, *i*) # Sp'') ⊆ *set* ((*p* # *ps*, *i*) # S'')›
‹*set* ((*q* # *ps*, *i*) # Sq'') ⊆ *set* ((*q* # *ps*, *i*) # S'')›
by *auto*
then have ‹*A* ⊢ (*p* # *ps*, *i*) # S''› ‹*A* ⊢ (*q* # *ps*, *i*) # S''›
using ‹*A* ⊢ (*p* # *ps*, *i*) # Sp''› ‹*A* ⊢ (*q* # *ps*, *i*) # Sq''› *inf fin STA-struct*
by *blast+*
then have ‹*A* ⊢ (*ps*, *i*) # S''›
using *ps* **by** (*meson DisP''* *list.set-intros*(*1*))
moreover have ‹*set* ((*ps*, *i*) # S'') ⊆ *S*›
using *ps* Sp' Sp'' Sq' Sq'' S'' **by** *auto*
ultimately show *False*
using ‹*consistent A S*› **unfolding** *consistent-def* **by** *blast*
qed
next
fix *p q i ps*
assume *ps*: ‹(*ps*, *i*) ∈ *S*› ‹(¬ (*p* ∨ *q*)) *on* (*ps*, *i*)›
show ‹(¬ *p*) *at i in'* *S*›
proof (*rule ccontr*)
assume ‹¬ (¬ *p*) *at i in'* *S*›
then obtain S' **where**
‹*A* ⊢ S'› **and**
S': ‹*set* S' ⊆ {((¬ *q*) # (¬ *p*) # *ps*, *i*)} ∪ *S*› **and**
‹((¬ *q*) # (¬ *p*) # *ps*, *i*) ∈. S'›
using ‹*maximal A S*› **unfolding** *maximal-def consistent-def*
by (*metis* (*mono-tags*, *lifting*) *insert-is-Un insert-subset list.simps*(*15*) *on.simps*
set-subset-Cons subset-insert)
then obtain S'' **where** S'':
‹*set* (((¬ *q*) # (¬ *p*) # *ps*, *i*) # S'') = *set* S'›
‹((¬ *q*) # (¬ *p*) # *ps*, *i*) ∉ *set* S''›
using *split-list*[**where** *x*=‹((¬ *q*) # (¬ *p*) # *ps*, *i*)›] **by** *blast*
then have ‹*A* ⊢ ((¬ *q*) # (¬ *p*) # *ps*, *i*) # S''›
using *inf fin STA-struct* ‹*A* ⊢ S'› **by** *blast*
then have ‹*A* ⊢ (*ps*, *i*) # S''›
using *ps* **by** (*meson DisN'* *list.set-intros*(*1*))
moreover have ‹*set* ((*ps*, *i*) # S'') ⊆ *S*›
using S' S'' *ps* **by** *auto*
ultimately show *False*
using ‹*consistent A S*› **unfolding** *consistent-def* **by** *blast*
qed
next
fix *p q i ps*
assume *ps*: ‹(*ps*, *i*) ∈ *S*› ‹(¬ (*p* ∨ *q*)) *on* (*ps*, *i*)›
show ‹(¬ *q*) *at i in'* *S*›
proof (*rule ccontr*)
assume ‹¬ (¬ *q*) *at i in'* *S*›
then obtain S' **where**

‹A ⊢ S′› **and**
S′: ‹set S′ ⊆ {((¬ q) # (¬ p) # ps, i)} ∪ S› **and**
‹((¬ q) # (¬ p) # ps, i) ∈. S′›
using ‹maximal A S› **unfolding** maximal-def consistent-def
by (metis (mono-tags, lifting) insert-is-Un insert-subset list.simps(15) on.simps set-subset-Cons subset-insert)
then obtain S′′ **where** S′′:
‹set (((¬ q) # (¬ p) # ps, i) # S′′) = set S′›
‹((¬ q) # (¬ p) # ps, i) ∉ set S′′›
using split-list[**where** x=‹((¬ q) # (¬ p) # ps, i)›] **by** blast
then have ‹A ⊢ ((¬ q) # (¬ p) # ps, i) # S′′›
using inf fin STA-struct ‹A ⊢ S′› **by** blast
then have ‹A ⊢ (ps, i) # S′′›
using ps **by** (meson DisN′ list.set-intros(1))
moreover have ‹set ((ps, i) # S′′) ⊆ S›
using S′ S′′ ps **by** auto
ultimately show False
using ‹consistent A S› **unfolding** consistent-def **by** blast
qed
next
fix p i ps
assume ‹∄ a. p = Nom a› ‹(ps, i) ∈ S› ‹(◇ p) on (ps, i)›
then show ‹∃ j. (◇ Nom j) at i in′ S ∧ (@ j p) at i in′ S›
using ‹saturated S› **unfolding** saturated-def **by** blast
next
fix p i j ps qs
assume
ps: ‹(ps, i) ∈ S› ‹(¬ (◇ p)) on (ps, i)› **and**
qs: ‹(qs, i) ∈ S› ‹(◇ Nom j) on (qs, i)›
show ‹(¬ (@ j p)) at i in′ S›
proof (rule ccontr)
assume ‹¬ (¬ (@ j p)) at i in′ S›
then obtain S′ n **where**
‹A, n ⊢ S′› **and** S′: ‹set S′ ⊆ {([¬ (@ j p)], i)} ∪ S› **and** ‹([¬ (@ j p)], i) ∈. S′›
using ‹maximal A S› **unfolding** maximal-def consistent-def
by (metis insert-is-Un list.set-intros(1) on.simps subset-insert)
then obtain S′′ **where** S′′:
‹set (([¬ (@ j p)], i) # S′′) = set S′› ‹([¬ (@ j p)], i) ∉ set S′′›
using split-list[**where** x=‹([¬ (@ j p)], i)›] **by** blast
then have ‹A ⊢ ([¬ (@ j p)], i) # S′′›
using inf fin STA-struct ‹A, n ⊢ S′› **by** blast
then have ‹A ⊢ ([¬ (@ j p)], i) # (ps, i) # (qs, i) # S′′›
using inf fin STA-struct[**where** branch′=‹([-], -) # (ps, i) # (qs, i) # S′′›] ‹A, n ⊢ S′›
by fastforce
then have ‹A ⊢ ([], i) # (ps, i) # (qs, i) # S′′›
using ps(2) qs(2) **by** (meson DiaN′ list.set-intros(1) set-subset-Cons subset-iff)

```
    moreover have ‹i ∈ branch-nominals ((ps, i) # (qs, i) # S'')›
      unfolding branch-nominals-def by simp
    ultimately have ‹A ⊢ (ps, i) # (qs, i) # S''›
      using GoTo by fast
    moreover have ‹set ((ps, i) # (qs, i) # S'') ⊆ S›
      using S' S'' ps qs by auto
    ultimately show False
      using ‹consistent A S› unfolding consistent-def by blast
  qed
next
  fix p i ps a
  assume ps: ‹(ps, a) ∈ S› ‹(@ i p) on (ps, a)›
  show ‹p at i in' S›
  proof (rule ccontr)
    assume ‹¬ p at i in' S›
    then obtain S' n where
      ‹A, n ⊢ S'› and S': ‹set S' ⊆ {([p], i)} ∪ S› and ‹([p], i) ∈. S'›
      using ‹maximal A S› unfolding maximal-def consistent-def
      by (metis insert-is-Un list.set-intros(1) on.simps subset-insert)
    then obtain S'' where S'':
      ‹set (([p], i) # S'') = set S'› ‹([p], i) ∉ set S''›
      using split-list[where x=‹([p], i)›] by blast
    then have ‹A ⊢ ([p], i) # S''›
      using inf fin STA-struct ‹A, n ⊢ S'› by blast
    moreover have ‹set (([p], i) # S'') ⊆ set (([p], i) # (ps, a) # S'')›
      by auto
    ultimately have ‹A ⊢ ([p], i) # (ps, a) # S''›
      using inf fin STA-struct ‹A, n ⊢ S'› by blast
    then have ‹A ⊢ ([], i) # (ps, a) # S''›
      using ps by (metis SatP' insert-iff list.simps(15))
    moreover have ‹i ∈ branch-nominals ((ps, a) # S'')›
      using ps unfolding branch-nominals-def by fastforce
    ultimately have ‹A ⊢ (ps, a) # S''›
      using GoTo by fast
    moreover have ‹set ((ps, a) # S'') ⊆ S›
      using S' S'' ps by auto
    ultimately show False
      using ‹consistent A S› unfolding consistent-def by blast
  qed
next
  fix p i ps a
  assume ps: ‹(ps, a) ∈ S› ‹(¬ (@ i p)) on (ps, a)›
  show ‹(¬ p) at i in' S›
  proof (rule ccontr)
    assume ‹¬ (¬ p) at i in' S›
    then obtain S' n where
      ‹A, n ⊢ S'› and S': ‹set S' ⊆ {([¬ p], i)} ∪ S› and ‹([¬ p], i) ∈. S'›
      using ‹maximal A S› unfolding maximal-def consistent-def
      by (metis insert-is-Un list.set-intros(1) on.simps subset-insert)
```

then obtain S'' **where** S'':
‹*set* (([¬ *p*], *i*) # S'') = *set* S'› ‹([¬ *p*], *i*) ∉ *set* S''›
using *split-list*[**where** *x*=‹([¬ *p*], *i*)›] **by** *blast*
then have ‹*A* ⊢ ([¬ *p*], *i*) # S''›
using *inf fin STA-struct* ‹*A*, *n* ⊢ S'› **by** *blast*
then have ‹*A* ⊢ ([¬ *p*], *i*) # (*ps*, *a*) # S''›
using *inf fin STA-struct*[**where** *branch′*=‹([¬ *p*], *i*) # - # S''›] ‹*A*, *n* ⊢ S'›
by *fastforce*
then have ‹*A* ⊢ ([], *i*) # (*ps*, *a*) # S''›
using *ps* **by** (*metis SatN′ insert-iff list.simps*(*15*))
moreover have ‹*i* ∈ *branch-nominals* ((*ps*, *a*) # S'')›
using *ps* **unfolding** *branch-nominals-def* **by** *fastforce*
ultimately have ‹*A* ⊢ (*ps*, *a*) # S''›
using *GoTo* **by** *fast*
moreover have ‹*set* ((*ps*, *a*) # S'') ⊆ *S*›
using S' S'' *ps* **by** *auto*
ultimately show *False*
using ‹*consistent A S*› **unfolding** *consistent-def* **by** *blast*
qed
next
fix *p i ps a*
assume *i*: ‹*i* ∈ *nominals p*› **and** *ps*: ‹(*ps*, *a*) ∈ *S*› ‹*p on* (*ps*, *a*)›
show ‹∃ *qs*. (*qs*, *i*) ∈ *S*›
proof (*rule ccontr*)
assume ‹∄ *qs*. (*qs*, *i*) ∈ *S*›
then obtain S' *n* **where**
‹*A*, *n* ⊢ S'› **and** S': ‹*set* S' ⊆ {([], *i*)} ∪ *S*› **and** ‹([], *i*) ∈. S'›
using ‹*maximal A S*› **unfolding** *maximal-def consistent-def*
by (*metis insert-is-Un subset-insert*)
then obtain S'' **where** S'':
‹*set* (([], *i*) # S'') = *set* S'› ‹([], *i*) ∉ *set* S''›
using *split-list*[**where** *x*=‹([], *i*)›] **by** *blast*
then have ‹*A* ⊢ ([], *i*) # (*ps*, *a*) # S''›
using *inf fin STA-struct*[**where** *branch′*=‹([], *i*) # (*ps*, *a*) # S''›] ‹*A*, *n* ⊢ S'›
by *fastforce*
moreover have ‹*i* ∈ *branch-nominals* ((*ps*, *a*) # S'')›
using *i ps* **unfolding** *branch-nominals-def* **by** *auto*
ultimately have ‹*A* ⊢ (*ps*, *a*) # S''›
using *GoTo* **by** *fast*
moreover have ‹*set* ((*ps*, *a*) # S'') ⊆ *S*›
using S' S'' *ps* **by** *auto*
ultimately show *False*
using ‹*consistent A S*› **unfolding** *consistent-def* **by** *blast*
qed
next
fix *p i j ps qs*
assume
p: ‹∀ *a*. *p* = *Nom a* ∨ *p* = (◇ *Nom a*) ⟶ *a* ∈ *A*› **and**
ps: ‹(*ps*, *i*) ∈ *S*› ‹*p on* (*ps*, *i*)› **and**

```
    qs: ‹(qs, i) ∈ S› ‹Nom j on (qs, i)›

  show ‹p at j in′ S›
  proof (rule ccontr)
    assume ‹∄ rs. (rs, j) ∈ S ∧ p on (rs, j)›
    then obtain S′ n where
      ‹A, n ⊢ S′› and S′: ‹set S′ ⊆ {([p], j)} ∪ S› and ‹([p], j) ∈. S′›
      using ‹maximal A S› unfolding maximal-def consistent-def
      by (metis insert-is-Un list.set-intros(1) on.simps subset-insert)
    then obtain S″ where S″:
      ‹set (([p], j) # S″) = set S′› ‹([p], j) ∉ set S″›
      using split-list[where x=‹([p], j)›] by blast
    then have ‹A ⊢ ([p], j) # S″›
      using inf fin STA-struct ‹A, n ⊢ S′› by blast
    then have ‹A ⊢ ([p], j) # (ps, i) # (qs, i) # S″›
      using inf fin STA-struct[where branch′=‹([-], -) # (ps, i) # (qs, i) # S″›]
‹A, n ⊢ S′›
      by fastforce
    then have ‹A ⊢ ([], j) # (ps, i) # (qs, i) # S″›
     using ps(2) qs(2) p by (meson Nom′ in-mono list.set-intros(1) set-subset-Cons)
    moreover have ‹j ∈ branch-nominals ((ps, i) # (qs, i) # S″)›
      using qs(2) unfolding branch-nominals-def by fastforce
    ultimately have ‹A ⊢ (ps, i) # (qs, i) # S″›
      using GoTo by fast
    moreover have ‹set ((ps, i) # (qs, i) # S″) ⊆ S›
      using S′ S″ ps qs by auto
    ultimately show False
      using ‹consistent A S› unfolding consistent-def by blast
  qed
qed
```

11.4 Result

```
theorem completeness:
  fixes p :: ‹('a :: countable, 'b :: countable) fm›
  assumes
    inf: ‹infinite (UNIV :: 'b set)› and
    valid: ‹∀(M :: ('b set, 'a) model) g w. M, g, w ⊨ p›
  shows ‹nominals p, 1 ⊢ [([¬ p], i)]›
proof −
  let ?A = ‹nominals p›

  have ‹?A ⊢ [([¬ p], i)]›
  proof (rule ccontr)
    assume ‹¬ ?A ⊢ [([¬ p], i)]›
    moreover have ‹finite ?A›
      using finite-nominals by blast
    ultimately have ∗: ‹consistent ?A {([¬ p], i)}›
      unfolding consistent-def using STA-struct inf
```

```
    by (metis empty-set list.simps(15))

  let ?S = ‹Extend ?A {([¬ p], i)} from-nat›
  have ‹finite {([¬ p], i)}›
    by simp
  then have fin: ‹finite (⋃ (block-nominals ` {([¬ p], i)}))›
    using finite-nominals-set by blast

  have ‹consistent ?A ?S›
    using consistent-Extend inf * fin ‹finite ?A› by blast
  moreover have ‹maximal ?A ?S›
    using maximal-Extend inf * fin by fastforce
  moreover have ‹saturated ?S›
    using saturated-Extend inf * fin by fastforce
  ultimately have ‹Hintikka ?A ?S›
    using Hintikka-Extend inf ‹finite ?A› by blast
  moreover have ‹([¬ p], i) ∈ ?S›
    using Extend-mem by blast
  moreover have ‹(¬ p) on ([¬ p], i)›
    by simp
  ultimately have ‹¬ Model (reach ?A ?S) (val ?S), assign ?A ?S, assign ?A
?S i ⊨ p›
    using Hintikka-model(2) by fast
  then show False
    using valid by blast
 qed
 then show ?thesis
  using STA-one by fast
qed
```

We arbitrarily fix nominal and propositional symbols to be natural numbers (any countably infinite type suffices) and define validity as truth in all models with sets of natural numbers as worlds. We show below that this implies validity for any type of worlds.

```
abbreviation
 ‹valid p ≡ ∀(M :: (nat set, nat) model) (g :: nat ⇒ -) w. M, g, w ⊨ p›
```

A formula is valid iff its negation has a closing tableau from a fresh world. We can assume a single unit of potential and take the allowed nominals to be the root nominals.

```
theorem main:
 assumes ‹i ∉ nominals p›
 shows ‹valid p ⟷ nominals p, 1 ⊢ [([¬ p], i)]›
proof
 assume ‹valid p›
 then show ‹nominals p, 1 ⊢ [([¬ p], i)]›
  using completeness by blast
next
```

assume ‹*nominals p*, *1* ⊢ [([¬ *p*], *i*)]›
then show ‹*valid p*›
using *assms soundness-fresh* **by** *fast*
qed

The restricted validity implies validity in general.

theorem *valid-semantics*:
‹*valid p* ⟶ *M*, *g*, *w* ⊨ *p*›
proof
assume ‹*valid p*›
then have ‹*i* ∉ *nominals p* ⟹ *nominals p* ⊢ [([¬ *p*], *i*)]› **for** *i*
using *main* **by** *blast*
moreover have ‹∃ *i*. *i* ∉ *nominals p*›
by (*simp add*: *finite-nominals ex-new-if-finite*)
ultimately show ‹*M*, *g*, *w* ⊨ *p*›
using *soundness-fresh* **by** *fast*
qed

end

References

[1] P. Blackburn, T. Bolander, T. Braüner, and K. F. Jørgensen. Completeness and Termination for a Seligman-style Tableau System. *Journal of Logic and Computation*, 27(1):81–107, 2017.

[2] K. F. Jørgensen, P. Blackburn, T. Bolander, and T. Braüner. Synthetic Completeness Proofs for Seligman-style Tableau Systems. In *Advances in Modal Logic*, volume 11, pages 302–321, 2016.