Formalizing a Seligman-Style Tableau System for Hybrid Logic

Asta Halkjr From

October 24, 2020

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:

- Jrgen 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*›
  ⟨*proof*⟩

lemma *sub-id*: ‹*sub id p* = *p*›
  ⟨*proof*⟩

lemma *sub-upd-fresh*: ‹*i* ∉ *nominals p* ⟹ *sub* (*f*(*i* := *j*)) *p* = *sub f p*›
  ⟨*proof*⟩

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: ‹$'w \Rightarrow 'w$ *set*›) (V: ‹$'w \Rightarrow 'a \Rightarrow$ *bool*›)

primrec *semantics*
  :: ‹($'w$, $'a$) *model* $\Rightarrow$ ($'b \Rightarrow 'w$) $\Rightarrow$ $'w \Rightarrow$ ($'a$, $'b$) *fm* $\Rightarrow$ *bool*›
  (‹-, -, - $\models$ -› [*50*, *50*, *50*] *50*) **where**
  ‹$(M, \text{-}, w \models \textit{Pro } x) = V\ M\ w\ x$›
| ‹$(\text{-}, g, w \models \textit{Nom } i) = (w = g\ i)$›
| ‹$(M, g, w \models \neg\ p) = (\neg\ M, g, w \models p)$›
| ‹$(M, g, w \models (p \vee q)) = ((M, g, w \models p) \vee (M, g, w \models q))$›
| ‹$(M, g, w \models \Diamond\ p) = (\exists\, v \in R\ M\ w.\ M, g, v \models p)$›
| ‹$(M, g, \text{-} \models @\ i\ p) = (M, g, g\ i \models p)$›

lemma ‹$M, g, w \models \top$›
  ⟨*proof*⟩

lemma *semantics-fresh*:
  ‹$i \notin \textit{nominals } p \Longrightarrow (M, g, w \models p) = (M, g(i := v), w \models p)$›
  ⟨*proof*⟩

2.1 Examples

abbreviation *is-named* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*is-named* $M \equiv \forall\, w.\ \exists\, a.\ V\ M\ a = w$›

abbreviation *reflexive* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*reflexive* $M \equiv \forall\, w.\ w \in R\ M\ w$›

abbreviation *irreflexive* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*irreflexive* $M \equiv \forall\, w.\ w \notin R\ M\ w$›

abbreviation *symmetric* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*symmetric* $M \equiv \forall\, v\ w.\ w \in R\ M\ v \longleftrightarrow v \in R\ M\ w$›

abbreviation *asymmetric* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*asymmetric* $M \equiv \forall\, v\ w.\ \neg\ (w \in R\ M\ v \wedge v \in R\ M\ w)$›

abbreviation *transitive* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*transitive* $M \equiv \forall\, v\ w\ x.\ w \in R\ M\ v \wedge x \in R\ M\ w \longrightarrow x \in R\ M\ v$›

abbreviation *universal* :: ‹($'w$, $'b$) *model* $\Rightarrow$ *bool*› **where**
  ‹*universal* $M \equiv \forall\, v\ w.\ v \in R\ M\ w$›

lemma ‹*irreflexive M* ⟹ *M*, *g*, *w* ⊨ **@** *i* ¬ (◇ *Nom i*)›
⟨*proof*⟩

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

lemma ‹*irreflexive M* = (∀ *g w*. *M*, *g*, *w* ⊨ **@** *i* ¬ (◇ *Nom i*))›
⟨*proof*⟩

lemma ‹*asymmetric M* = (∀ *g w*. *M*, *g*, *w* ⊨ **@** *i* (□ ¬ (◇ *Nom i*)))›
⟨*proof*⟩

lemma ‹*universal M* = (∀ *g w*. *M*, *g*, *w* ⊨ ◇ *Nom i*)›
⟨*proof*⟩

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*)

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) ⟹
```

$A,\ Suc\ n \vdash (p\ \#\ ps,\ a)\ \#\ branch \Longrightarrow$
$A,\ n \vdash (ps,\ a)\ \#\ branch$›
| *SatN*:
‹$(\neg\ (@\ a\ p))\ at\ b\ in\ (ps,\ a)\ \#\ branch \Longrightarrow$
$new\ (\neg\ p)\ a\ ((ps,\ a)\ \#\ branch) \Longrightarrow$
$A,\ Suc\ n \vdash ((\neg\ p)\ \#\ ps,\ a)\ \#\ branch \Longrightarrow$
$A,\ n \vdash (ps,\ a)\ \#\ branch$›
| *GoTo*:
‹$i \in branch\text{-}nominals\ branch \Longrightarrow$
$A,\ n \vdash ([],\ i)\ \#\ branch \Longrightarrow$
$A,\ Suc\ n \vdash branch$›
| *Nom*:
‹$p\ at\ b\ in\ (ps,\ a)\ \#\ branch \Longrightarrow Nom\ a\ at\ b\ in\ (ps,\ a)\ \#\ branch \Longrightarrow$
$\forall\, i.\ p = Nom\ i \vee p = (\Diamond\ Nom\ i) \longrightarrow i \in A \Longrightarrow$
$new\ p\ a\ ((ps,\ a)\ \#\ branch) \Longrightarrow$
$A,\ Suc\ n \vdash (p\ \#\ ps,\ a)\ \#\ branch \Longrightarrow$
$A,\ n \vdash (ps,\ a)\ \#\ branch$›

abbreviation *STA-ex-potential* :: ‹$'b\ set \Rightarrow ('a,\ 'b)\ branch \Rightarrow bool$› (‹- ⊢ -› [50, 50] 50) **where**
‹$A \vdash branch \equiv \exists\, n.\ A,\ n \vdash branch$›

lemma *STA-Suc*: ‹$A,\ n \vdash branch \Longrightarrow A,\ Suc\ n \vdash branch$›
⟨*proof*⟩

A verified derivation in the calculus.

lemma
fixes *i*
defines ‹$p \equiv \neg\ (@\ i\ (Nom\ i))$›
shows ‹$A,\ Suc\ n \vdash [([p],\ a)]$›
⟨*proof*⟩

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* :: ‹$('w,\ 'a)\ model \Rightarrow ('b \Rightarrow 'w) \Rightarrow ('a,\ 'b)\ block \Rightarrow bool$›
(‹-, - $\models_B$ -› [50, 50] 50) **where**
‹$(M,\ g \models_B (ps,\ i)) = (\forall\, p\ on\ (ps,\ i).\ M,\ g,\ g\ i \models p)$›

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

lemma *block-nominals*:
‹$p\ on\ block \Longrightarrow i \in nominals\ p \Longrightarrow i \in block\text{-}nominals\ block$›

⟨*proof*⟩

lemma *block-sat-fresh*:
  assumes ‹M, g $\models_B$ *block*› ‹$i \notin$ *block-nominals block*›
  shows ‹M, $g(i := v)$ $\models_B$ *block*›
  ⟨*proof*⟩

lemma *branch-sat-fresh*:
  assumes ‹M, g $\models_\Theta$ *branch*› ‹$i \notin$ *branch-nominals branch*›
  shows ‹M, $g(i := v)$ $\models_\Theta$ *branch*›
  ⟨*proof*⟩

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

lemma *soundness'*: ‹A, $n \vdash$ *branch* $\Longrightarrow$ M, g $\models_\Theta$ *branch* $\Longrightarrow$ *False*›
⟨*proof*⟩

lemma *block-sat*: ‹$\forall$ p *on block*. M, g, $w \models p \Longrightarrow M$, g $\models_B$ *block*›
  ⟨*proof*⟩

lemma *branch-sat*:
  assumes ‹$\forall$ $(ps, i) \in$ *set branch*. $\forall$ p *on* (ps, i). M, g, $w \models p$›
  shows ‹M, g $\models_\Theta$ *branch*›
  ⟨*proof*⟩

lemma *soundness*:
  assumes ‹A, $n \vdash$ *branch*›
  shows ‹$\exists$ *block* $\in$ *set branch*. $\exists$ p *on block*. $\neg$ M, g, $w \models p$›
  ⟨*proof*⟩

corollary ‹$\neg$ A, $n \vdash$ []›
  ⟨*proof*⟩

theorem *soundness-fresh*:
  assumes ‹A, $n \vdash$ [([$\neg$ p], i)]› ‹$i \notin$ *nominals* p›
  shows ‹M, g, $w \models p$›
⟨*proof*⟩

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* $\Rightarrow$ *bool*› **where**
  ‹*nonempty* $(ps, i) = (ps \neq [])$›

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*⟩

lemma *STA-nonempty*:
‹*A*, *n* ⊢ *left* @ *right* ⟹ *A*, *Suc m* ⊢ *filter nonempty left* @ *right*›
⟨*proof*⟩

theorem *STA-potential*: ‹*A*, *n* ⊢ *branch* ⟹ *A*, *Suc m* ⊢ *branch*›
⟨*proof*⟩

corollary *STA-one*: ‹*A*, *n* ⊢ *branch* ⟹ *A*, *1* ⊢ *branch*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *rev-nth-zero*: ‹(*xs* @ [*x*]) !. *0* = *Some x*›
⟨*proof*⟩

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

lemma *rev-nth-Suc*: ‹$(xs\ @\ [x])\ !.\ Suc\ v = xs\ !.\ v$›
⟨*proof*⟩

lemma *rev-nth-bounded*: ‹$v < length\ xs \Longrightarrow \exists x.\ xs\ !.\ v = Some\ x$›
⟨*proof*⟩

lemma *rev-nth-Cons*: ‹$xs\ !.\ v = Some\ y \Longrightarrow (x\ \#\ xs)\ !.\ v = Some\ y$›
⟨*proof*⟩

lemma *rev-nth-append*: ‹$xs\ !.\ v = Some\ y \Longrightarrow (ys\ @\ xs)\ !.\ v = Some\ y$›
⟨*proof*⟩

lemma *rev-nth-mem*: ‹$block \in.\ branch \longleftrightarrow (\exists v.\ branch\ !.\ v = Some\ block)$›
⟨*proof*⟩

lemma *rev-nth-on*: ‹$p\ on\ (ps,\ i) \longleftrightarrow (\exists v.\ ps\ !.\ v = Some\ p) \lor p = Nom\ i$›
⟨*proof*⟩

lemma *rev-nth-Some*: ‹$xs\ !.\ v = Some\ y \Longrightarrow v < length\ xs$›
⟨*proof*⟩

lemma *index-Cons*:
assumes ‹$((ps,\ a)\ \#\ branch)\ !.\ v = Some\ (qs,\ b)$› ‹$qs\ !.\ v' = Some\ q$›
shows ‹$\exists qs'.\ ((p\ \#\ ps,\ a)\ \#\ branch)\ !.\ v = Some\ (qs',\ b) \land qs'\ !.\ v' = Some\ q$›
⟨*proof*⟩

6.2 Mapping

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

primrec *mapi-block* ::
‹$(nat \Rightarrow ('a,\ 'b)\ fm \Rightarrow ('a,\ 'b)\ fm) \Rightarrow (('a,\ 'b)\ block \Rightarrow ('a,\ 'b)\ block)$› **where**
‹$mapi\text{-}block\ f\ (ps,\ i) = (mapi\ f\ ps,\ i)$›

definition *mapi-branch* ::
‹$(nat \Rightarrow nat \Rightarrow ('a,\ 'b)\ fm \Rightarrow ('a,\ 'b)\ fm) \Rightarrow (('a,\ 'b)\ branch \Rightarrow ('a,\ 'b)\ branch)$›
where
‹$mapi\text{-}branch\ f\ branch \equiv mapi\ (\lambda v.\ mapi\text{-}block\ (f\ v))\ branch$›

abbreviation *mapper* ::
‹$(('a,\ 'b)\ fm \Rightarrow ('a,\ 'b)\ fm) \Rightarrow$
$(nat \times nat)\ set \Rightarrow nat \Rightarrow nat \Rightarrow ('a,\ 'b)\ fm \Rightarrow ('a,\ 'b)\ fm$› **where**
‹$mapper\ f\ xs\ v\ v'\ p \equiv (if\ (v,\ v') \in xs\ then\ f\ p\ else\ p)$›

lemma *mapi-block-add-oob*:
assumes ‹$length\ ps \le v'$›

shows
‹*mapi-block* (*mapper f* ({(*v*, *v*′)} ∪ *xs*) *v*) (*ps*, *i*) =
mapi-block (*mapper f xs v*) (*ps*, *i*)›
⟨*proof*⟩

lemma *mapi-branch-add-oob*:
assumes ‹*length branch* ≤ *v*›
shows
‹*mapi-branch* (*mapper f* ({(*v*, *v*′)} ∪ *xs*)) *branch* =
mapi-branch (*mapper f xs*) *branch*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *mapi-branch-mem*:
assumes ‹(*ps*, *i*) ∈. *branch*›
shows ‹∃ *v*. (*mapi* (*f v*) *ps*, *i*) ∈. *mapi-branch f branch*›
⟨*proof*⟩

lemma *rev-nth-mapi-branch*:
assumes ‹*branch* !. *v* = *Some* (*ps*, *a*)›
shows ‹(*mapi* (*f v*) *ps*, *a*) ∈. *mapi-branch f branch*›
⟨*proof*⟩

lemma *rev-nth-mapi-block*:
assumes ‹*ps* !. *v*′ = *Some p*›
shows ‹*f v*′ *p on* (*mapi f ps*, *a*)›
⟨*proof*⟩

lemma *mapi-append*:
‹*mapi f* (*xs* @ *ys*) = *mapi* (λ*v*. *f* (*v* + *length ys*)) *xs* @ *mapi f ys*›
⟨*proof*⟩

lemma *mapi-block-id*: ‹*mapi-block* (*mapper f* {} *v*) (*ps*, *i*) = (*ps*, *i*)›
⟨*proof*⟩

lemma *mapi-branch-id*: ‹*mapi-branch* (*mapper f* {}) *branch* = *branch*›
⟨*proof*⟩

lemma *length-mapi*: ‹*length* (*mapi f xs*) = *length xs*›
⟨*proof*⟩

lemma *mapi-rev-nth*:
assumes ‹*xs* !. *v* = *Some x*›
shows ‹*mapi f xs* !. *v* = *Some* (*f v x*)›

⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *Dup-branch*:
‹*Dup p i branch xs* ⟹ *Dup p i* (*extra* @ *branch*) *xs*›
⟨*proof*⟩

lemma *Dup-block*:
assumes ‹*Dup p i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹*Dup p i* ((*ps*′ @ *ps*, *a*) # *branch*) *xs*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *only-touches-opening*:

assumes ‹*only-touches i branch xs*› ‹(v, v') ∈ *xs*› ‹*branch* !. v = *Some* (*ps*, *a*)›
shows ‹$i = a$›
⟨*proof*⟩

lemma *Dup-head*:
‹*Dup p i* ((*ps*, *a*) # *branch*) *xs* ⟹ *Dup p i* ((*q* # *ps*, *a*) # *branch*) *xs*›
⟨*proof*⟩

lemma *Dup-head-oob'*:
assumes ‹*Dup p i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹(*length branch*, *k* + *length ps*) ∉ *xs*›
⟨*proof*⟩

lemma *Dup-head-oob*:
assumes ‹*Dup p i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹(*length branch*, *length ps*) ∉ *xs*›
⟨*proof*⟩

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*⟩

lemma *omit-id*: ‹*omit* {} *ps* = *ps*›
⟨*proof*⟩

lemma *omit-block-id*: ‹*omit-block* {} *block* = *block*›
⟨*proof*⟩

lemma *omit-branch-id*: ‹*omit-branch* {} *branch* = *branch*›
⟨*proof*⟩

lemma *omit-branch-mem-diff-opening*:
assumes ‹*only-touches i branch xs*› ‹(*ps*, *a*) ∈. *branch*› ‹$i \neq a$›
shows ‹(*ps*, *a*) ∈. *omit-branch xs branch*›
⟨*proof*⟩

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*⟩

lemma *omit-del*:
assumes ‹$p \in .\ ps$› ‹$p \notin$ *set* (*omit xs ps*)›
shows ‹$\exists\, v.\ ps\ !.\ v =$ *Some p* $\land\ v \in xs$›
⟨*proof*⟩

lemma *omit-all-is*:
assumes ‹*all-is p ps xs*› ‹$q \in .\ ps$› ‹$q \notin$ *set* (*omit xs ps*)›
shows ‹$q = p$›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *Dup-all-is-branch*: ‹*Dup p i branch xs* ⟹ *all-is-branch p i branch xs*›
⟨*proof*⟩

lemma *omit-branch-mem-diff-formula*:
assumes ‹*all-is-branch p i branch xs*› ‹*q at i in branch*› ‹$p \neq q$›
shows ‹*q at i in omit-branch xs branch*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *omit-set*: ‹*set* (*omit xs ps*) ⊆ *set ps*›
⟨*proof*⟩

lemma *on-omit*: ‹*p on* (*omit xs ps*, *i*) ⟹ *p on* (*ps*, *i*)›
⟨*proof*⟩

lemma *all-is-set*:
assumes ‹*all-is p ps xs*›
shows ‹{*p*} ∪ *set* (*omit xs ps*) = {*p*} ∪ *set ps*›
⟨*proof*⟩

lemma *all-is-list-nominals*:

assumes ‹*all-is p ps xs*›
shows ‹*nominals p* ∪ *list-nominals* (*omit xs ps*) = *nominals p* ∪ *list-nominals ps*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *Dup-branch-nominals*:
assumes ‹*Dup p i branch xs*›
shows ‹*branch-nominals* (*omit-branch xs branch*) = *branch-nominals branch*›
⟨*proof*⟩

lemma *omit-branch-mem-dual*:
assumes ‹*p at i in omit-branch xs branch*›
shows ‹*p at i in branch*›
⟨*proof*⟩

lemma *witnessed-omit-branch*:
assumes ‹*witnessed p a* (*omit-branch xs branch*)›
shows ‹*witnessed p a branch*›
⟨*proof*⟩

lemma *new-omit-branch*:
assumes ‹*new p a branch*›
shows ‹*new p a* (*omit-branch xs branch*)›
⟨*proof*⟩

lemma *omit-oob*:
assumes ‹*length ps* ≤ *v*›
shows ‹*omit* ({*v*} ∪ *xs*) *ps* = *omit xs ps*›
⟨*proof*⟩

lemma *omit-branch-oob*:
assumes ‹*length branch* ≤ *v*›
shows ‹*omit-branch* ({(*v*, *v'*)} ∪ *xs*) *branch* = *omit-branch xs branch*›
⟨*proof*⟩

7.3 Induction

lemma *STA-Dup*:
  assumes ‹A, n ⊢ branch› ‹Dup q i branch xs›
  shows ‹A, n ⊢ omit-branch xs branch›
  ⟨*proof*⟩

theorem *Dup*:
  assumes ‹A, n ⊢ (p # ps, a) # branch› ‹¬ new p a ((ps, a) # branch)›
  shows ‹A, n ⊢ (ps, a) # branch›
⟨*proof*⟩

7.4 Unrestricted rules

lemma *STA-add*: ‹A, n ⊢ branch ⟹ A, m + n ⊢ branch›
  ⟨*proof*⟩

lemma *STA-le*: ‹A, n ⊢ branch ⟹ n ≤ m ⟹ A, m ⊢ branch›
  ⟨*proof*⟩

lemma *Neg'*:
  assumes
    ‹(¬ ¬ p) at a in (ps, a) # branch›
    ‹A, n ⊢ (p # ps, a) # branch›
  shows ‹A, n ⊢ (ps, a) # branch›
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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›
⟨proof⟩

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›
⟨proof⟩

lemma *SatP'*:
assumes
‹(@ a p) at b in (ps, a) # branch›
‹A, n ⊢ (p # ps, a) # branch›
shows ‹A, n ⊢ (ps, a) # branch›
⟨proof⟩

lemma *SatN'*:
assumes
‹(¬ (@ a p)) at b in (ps, a) # branch›
‹A, n ⊢ ((¬ p) # ps, a) # branch›
shows ‹A, n ⊢ (ps, a) # branch›
⟨proof⟩

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⟩

8 Substitution

lemma *finite-nominals*: ‹finite (nominals p)›
⟨proof⟩

lemma *finite-block-nominals*: ‹finite (block-nominals block)›
⟨proof⟩

lemma *finite-branch-nominals*: ‹finite (branch-nominals branch)›
⟨proof⟩

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*›
⟨*proof*⟩

lemma *sub-branch-mem*:
assumes ‹(*ps*, *i*) ∈. *branch*›
shows ‹(*sub-list f ps*, *f i*) ∈. *sub-branch f branch*›
⟨*proof*⟩

lemma *sub-block-nominals*: ‹*block-nominals* (*sub-block f block*) = *f* ‘ *block-nominals block*›
⟨*proof*⟩

lemma *sub-branch-nominals*:
‹*branch-nominals* (*sub-branch f branch*) = *f* ‘ *branch-nominals branch*›
⟨*proof*⟩

lemma *sub-list-id*: ‹*sub-list id ps* = *ps*›
⟨*proof*⟩

lemma *sub-block-id*: ‹*sub-block id block* = *block*›
⟨*proof*⟩

lemma *sub-branch-id*: ‹*sub-branch id branch* = *branch*›
⟨*proof*⟩

lemma *sub-block-upd-fresh*:
assumes ‹*i* ∉ *block-nominals block*›
shows ‹*sub-block* (*f*(*i* := *j*)) *block* = *sub-block f block*›
⟨*proof*⟩

lemma *sub-branch-upd-fresh*:
assumes ‹*i* ∉ *branch-nominals branch*›
shows ‹*sub-branch* (*f*(*i* := *j*)) *branch* = *sub-branch f branch*›
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *sub-block-comp*: ‹*sub-block f (sub-block g block) = sub-block (f o g) block*›
⟨*proof*⟩

lemma *sub-branch-comp*:
‹*sub-branch f (sub-branch g branch) = sub-branch (f o g) branch*›
⟨*proof*⟩

lemma *swap-id*: ‹*(id(i := j, j := i)) o (id(i := j, j := i)) = id*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

lemma *ex-fresh-gt*:
fixes *f* :: ‹*'b* ⇒ *'c*›
assumes ‹∃ *g* :: *'c* ⇒ *'b. surj g*› ‹*finite A*› ‹*i* ∉ *A*›
shows ‹∃ *j. j* ∉ *f* ‘ *A*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*⟩

corollary *STA-sub*:
fixes f :: ‹′b ⇒ ′b›
assumes ‹*A* ⊢ *branch*› ‹*finite A*›
shows ‹*f* ‘ *A* ⊢ *sub-branch f branch*›
⟨*proof*⟩

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*⟩

9 Structural Properties

lemma *block-nominals-branch*:
assumes ‹*block* ∈. *branch*›
shows ‹*block-nominals block* ⊆ *branch-nominals branch*›
⟨*proof*⟩

lemma *sub-block-fresh*:
assumes ‹*i* ∉ *branch-nominals branch*› ‹*block* ∈. *branch*›
shows ‹*sub-block* (*f*(*i* := *j*)) *block* = *sub-block f block*›
⟨*proof*⟩

lemma *list-down-induct* [*consumes 1*, *case-names Start Cons*]:
assumes ‹∀ *y* ∈ *set ys*. *Q y*› ‹*P* (*ys* @ *xs*)›
‹⋀*y xs*. *Q y* ⟹ *P* (*y* # *xs*) ⟹ *P xs*›
shows ‹*P xs*›
⟨*proof*⟩

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* ⊢ (*ps* @ *ps′*, *a*) # *branch*›

shows ‹A, n ⊢ (ps', a) # *branch*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *STA-drop-block'*:
assumes ‹A, n ⊢ (ps, a) # *branch*› ‹(ps, a) ∈. *branch*›
shows ‹A, *Suc n* ⊢ *branch*›
⟨*proof*⟩

lemma *sub-branch-image*: ‹*set* (*sub-branch f branch*) = *sub-block f* ‘ *set branch*›
⟨*proof*⟩

lemma *sub-block-repl*:
assumes ‹j ∉ *block-nominals block*›
shows ‹i ∉ *block-nominals* (*sub-block* (*id*(i := j, j := i)) *block*)›
⟨*proof*⟩

lemma *sub-branch-repl*:
assumes ‹j ∉ *branch-nominals branch*›
shows ‹i ∉ *branch-nominals* (*sub-branch* (*id*(i := j, j := i)) *branch*)›
⟨*proof*⟩

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'*›
⟨*proof*⟩

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*›
⟨*proof*⟩

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*)›
⟨*proof*⟩

lemma *bridge′-nominals*:
‹*nominals* (*bridge′ k j p*) ∪ {*k*, *j*} = *nominals p* ∪ {*k*, *j*}›
⟨*proof*⟩

lemma *bridge-nominals*:
‹*nominals* (*bridge k j xs v v′ p*) ∪ {*k*, *j*} = *nominals p* ∪ {*k*, *j*}›
⟨*proof*⟩

lemma *bridge-block-nominals*:
‹*block-nominals* (*mapi-block* (*bridge k j xs v*) (*ps*, *a*)) ∪ {*k*, *j*} =
block-nominals (*ps*, *a*) ∪ {*k*, *j*}›
⟨*proof*⟩

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

lemma *at-in-mapi-branch*:
assumes ‹p *at* a *in branch*› ‹$p \neq$ *Nom* a›
shows ‹$\exists\, v\ v'.\ f\ v\ v'\ p$ *at* a *in mapi-branch* f *branch*›
⟨*proof*⟩

lemma *nom-at-in-bridge*:
fixes $k\ j\ xs$
defines ‹$f \equiv$ *bridge* $k\ j\ xs$›
assumes ‹*Nom* i *at* a *in branch*›
shows ‹*Nom* i *at* a *in mapi-branch* f *branch*›
⟨*proof*⟩

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*⟩

lemma *bridge-proper-dia*:
assumes ‹$\nexists\, a.\ p =$ *Nom* a›
shows ‹*bridge* $k\ j\ xs\ v\ v'$ ($\Diamond\ p$) = ($\Diamond\ p$)›
⟨*proof*⟩

lemma *bridge-compl-cases*:
fixes $k\ j\ xs\ v\ v'\ w\ w'\ p$
defines ‹$q \equiv$ *bridge* $k\ j\ xs\ v\ v'\ p$› **and** ‹$q' \equiv$ *bridge* $k\ j\ xs\ w\ w'$ ($\neg\ p$)›
shows
‹(q = ($\Diamond$ *Nom* j) $\wedge$ q' = ($\neg$ ($\Diamond$ *Nom* k))) $\vee$
($\exists\, r.\ q$ = ($\neg$ (**@** $j\ r$)) $\wedge$ q' = ($\neg\ \neg$ (**@** $k\ r$))) $\vee$
($\exists\, r.\ q$ = (**@** $k\ r$) $\wedge$ q' = ($\neg$ (**@** $j\ r$))) $\vee$
($q = p$ $\wedge$ q' = ($\neg\ p$))›
⟨*proof*⟩

10.2 Descendants

inductive *descendants* :: ‹$'b \Rightarrow 'b \Rightarrow ('a,\ 'b)$ *branch* $\Rightarrow$ (*nat* $\times$ *nat*) *set* $\Rightarrow$ *bool*›
where
Initial:
‹*branch* !. v = *Some* (qs, i) $\Longrightarrow$ qs !. v' = *Some* ($\Diamond$ *Nom* k) $\Longrightarrow$
descendants $k\ i$ *branch* $\{(v,\ v')\}$›
| *Derived*:
‹*branch* !. v = *Some* (qs, a) $\Longrightarrow$ qs !. v' = *Some* ($\neg$ (**@** $k\ p$)) $\Longrightarrow$
descendants $k\ i$ *branch* $xs \Longrightarrow (w,\ w') \in xs \Longrightarrow$
branch !. w = *Some* (rs, a) $\Longrightarrow$ rs !. w' = *Some* ($\Diamond$ *Nom* k) $\Longrightarrow$
descendants $k\ i$ *branch* ($\{(v,\ v')\} \cup xs$)›
| *Copied*:
‹*branch* !. v = *Some* (qs, a) $\Longrightarrow$ qs !. v' = *Some* p $\Longrightarrow$
descendants $k\ i$ *branch* $xs \Longrightarrow (w,\ w') \in xs \Longrightarrow$

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*)›
⟨*proof*⟩

lemma *descendants-bounds-fst*:
assumes ‹*descendants k i branch xs*› ‹(v, v') ∈ *xs*›
shows ‹v < *length branch*›
⟨*proof*⟩

lemma *descendants-bounds-snd*:
assumes ‹*descendants k i branch xs*› ‹(v, v') ∈ *xs*› ‹*branch* !. v = *Some* (*ps*, *a*)›
shows ‹v' < *length ps*›
⟨*proof*⟩

lemma *descendants-branch*:
‹*descendants k i branch xs* ⟹ *descendants k i* (*extra* @ *branch*) *xs*›
⟨*proof*⟩

lemma *descendants-block*:
assumes ‹*descendants k i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹*descendants k i* ((*ps'* @ *ps*, *a*) # *branch*) *xs*›
⟨*proof*⟩

lemma *descendants-no-head*:
assumes ‹*descendants k i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹*descendants k i* ((*p* # *ps*, *a*) # *branch*) *xs*›
⟨*proof*⟩

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*)))›
⟨*proof*⟩

lemma *descendants-oob-head'*:
assumes ‹*descendants k i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹(*length branch*, *m* + *length ps*) ∉ *xs*›
⟨*proof*⟩

lemma *descendants-oob-head*:
assumes ‹*descendants k i* ((*ps*, *a*) # *branch*) *xs*›
shows ‹(*length branch*, *length ps*) ∉ *xs*›

⟨*proof*⟩

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*)›
  ⟨*proof*⟩

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*⟩

10.4 Derivation

theorem *Bridge*:
  fixes *i* :: *′b*
  assumes *inf*: ‹*infinite* (*UNIV* :: *′b set*)› **and** *fin*: ‹*finite A*› **and** ‹*j* ∈ *A*›
    ‹*Nom k at j in* (*ps*, *i*) # *branch*› ‹(◇ *Nom j*) *at i in* (*ps*, *i*) # *branch*›
    ‹*A* ⊢ ((◇ *Nom k*) # *ps*, *i*) # *branch*›
  shows ‹*A* ⊢ (*ps*, *i*) # *branch*›
⟨*proof*⟩

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›
  ⟨*proof*⟩

lemma *hequiv-refl′*: ‹(ps, i) ∈ H ⟹ hequiv H i i›
  ⟨*proof*⟩

lemma *hequiv-sym′*:
  assumes ‹Hintikka A H› ‹i ∈ A› ‹hequiv H i j›
  shows ‹hequiv H j i›
⟨*proof*⟩

lemma *hequiv-sym*: ‹Hintikka A H ⟹ i ∈ A ⟹ j ∈ A ⟹ hequiv H i j ⟷ hequiv H j i›
  ⟨*proof*⟩

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⟩

lemma *hequiv-names*: ‹$hequiv\ H\ i\ j \Longrightarrow i \in names\ H$›
⟨proof⟩

lemma *hequiv-names-rel*:
assumes ‹$Hintikka\ A\ H$›
shows ‹$hequiv\text{-}rel\ A\ H \subseteq names\ H \times names\ H$›
⟨proof⟩

lemma *hequiv-refl-rel*:
assumes ‹$Hintikka\ A\ H$›
shows ‹$refl\text{-}on\ (names\ H \cap A)\ (hequiv\text{-}rel\ A\ H)$›
⟨proof⟩

lemma *hequiv-sym-rel*: ‹$Hintikka\ A\ H \Longrightarrow sym\ (hequiv\text{-}rel\ A\ H)$›
⟨proof⟩

lemma *hequiv-trans-rel*: ‹$Hintikka\ B\ A \Longrightarrow trans\ (hequiv\text{-}rel\ B\ A)$›
⟨proof⟩

lemma *hequiv-rel*: ‹$Hintikka\ A\ H \Longrightarrow equiv\ (names\ H \cap A)\ (hequiv\text{-}rel\ A\ H)$›
⟨proof⟩

lemma *nominal-in-names*:
assumes ‹$Hintikka\ A\ H$› ‹$\exists\, block \in H.\ i \in block\text{-}nominals\ block$›
shows ‹$i \in names\ H$›
⟨proof⟩

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 $\Diamond$ *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* :: ‹$'b\ set \Rightarrow ('a,\ 'b)\ block\ set \Rightarrow\ 'b \Rightarrow\ 'b\ set$› **where**
‹$assign\ A\ H\ i \equiv if\ \exists\, a.\ a \in A \land Nom\ a\ at\ i\ in'\ H$
$then\ proj\ (hequiv\text{-}rel\ A\ H)\ (SOME\ a.\ a \in A \land Nom\ a\ at\ i\ in'\ H)$
$else\ \{i\}$›

definition *reach* :: ‹$'b\ set \Rightarrow ('a,\ 'b)\ block\ set \Rightarrow\ 'b\ set \Rightarrow\ 'b\ set\ set$› **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*⟩

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*⟩

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*⟩

lemma *mem-hequiv-rel*: ‹*a* ∈ *proj* (*hequiv-rel A H*) *b* ⟹ *a* ∈ *A*›
⟨*proof*⟩

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*⟩

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*›
⟨*proof*⟩

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*⟩

lemma *assign-named*:
assumes ‹*Hintikka A H*› ‹*i* ∈ *proj* (*hequiv-rel A H*) *a*›
shows ‹*i* ∈ *names H*›
⟨*proof*⟩

lemma *assign-unique*:
assumes ‹*Hintikka A H*› ‹*a* ∈ *assign A H i*›
shows ‹*assign A H a* = *assign A H i*›

⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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)*›
  ⟨*proof*⟩

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))›
⟨proof⟩

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)›
⟨proof⟩

lemma *extend-mem*: ‹S ⊆ extend A S f n›
⟨proof⟩

lemma *Extend-mem*: ‹S ⊆ Extend A S f›
⟨proof⟩

11.2.1 Consistency

lemma *split-list*:
‹set A ⊆ {x} ∪ X ⟹ x ∈. A ⟹ ∃ B. set (x # B) = set A ∧ x ∉ set B›
⟨proof⟩

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)›
⟨proof⟩

lemma *consistent-drop-block*: ‹consistent A ({block} ∪ S) ⟹ consistent A S›
⟨proof⟩

lemma *inconsistent-weaken*: ‹¬ *consistent A S* ⟹ *S* ⊆ *S'* ⟹ ¬ *consistent A S'*›
⟨*proof*⟩

lemma *finite-nominals-set*: ‹*finite S* ⟹ *finite* (⋃ *block* ∈ *S*. *block-nominals block*)›
⟨*proof*⟩

lemma *witness-list-used*:
fixes *i* :: *'b*
assumes *inf*: ‹*infinite* (*UNIV* :: *'b set*)› **and** ‹*finite used*› ‹*i* ∉ *list-nominals ps*›
shows ‹*i* ∉ *list-nominals* (*witness-list ps* ({*i*} ∪ *used*))›
⟨*proof*⟩

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*))›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*)›
⟨*proof*⟩

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*⟩

lemma *finite-nominals-extend*:
assumes ‹*finite* (⋃ (*block-nominals* ‘ *S*))›
shows ‹*finite* (⋃ (*block-nominals* ‘ *extend A S f n*))›
⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma *UN-finite-bound*:
  assumes ‹*finite A*› ‹$A \subseteq (\bigcup n.\ f\ n)$›
  shows ‹$\exists m :: nat.\ A \subseteq (\bigcup n \leq m.\ f\ n)$›
  ⟨*proof*⟩

lemma *extend-bound*: ‹($\bigcup n \leq m$. *extend A S f n*) = *extend A S f m*›
⟨*proof*⟩

lemma *consistent-Extend*:
  fixes *S* :: ‹('*a*, '*b*) *block set*›
  assumes *inf*: ‹*infinite* (*UNIV* :: '*b set*)› **and** ‹*finite A*›
    ‹*consistent A S*› ‹*finite* ($\bigcup$ (*block-nominals* ‘ *S*))›
  shows ‹*consistent A* (*Extend A S f*)›
  ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma *maximal-Extend*:
  fixes *S* :: ‹('*a*, '*b*) *block set*›
  assumes *inf*: ‹*infinite* (*UNIV* :: '*b set*)› **and** ‹*finite A*›
    ‹*consistent A S*› ‹*finite* ($\bigcup$ (*block-nominals* ‘ *S*))› ‹*surj f*›
  shows ‹*maximal A* (*Extend A S f*)›
⟨*proof*⟩

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*⟩

lemma *ex-witness-list*:
  assumes ‹$p \in. ps$› ‹*proper-dia* p = *Some* q›
  shows ‹$\exists\, i.\ \{$@ $i\ q$, ◊ *Nom* $i\} \subseteq$ *set* (*witness-list ps used*)›
  ⟨*proof*⟩

lemma *saturated-Extend*:
  fixes S :: ‹('*a*, '*b*) *block set*›
  assumes *inf*: ‹*infinite* (*UNIV* :: '*b set*)› **and** *fin*: ‹*finite A*› **and**
    ‹*consistent A S*› ‹*finite* ($\bigcup$ (*block-nominals* ‘ S))› ‹*surj f*›
  shows ‹*saturated* (*Extend A S f*)›
  ⟨*proof*⟩

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*›
  ⟨*proof*⟩

11.4 Result

theorem *completeness*:
  fixes p :: ‹('*a* :: *countable*, '*b* :: *countable*) *fm*›
  assumes
    inf: ‹*infinite* (*UNIV* :: '*b set*)› **and**
    valid: ‹$\forall$ (*M* :: ('*b set*, '*a*) *model*) *g w*. *M*, *g*, *w* $\models$ *p*›
  shows ‹*nominals p*, *1* $\vdash$ [([¬ *p*], *i*)]›
⟨*proof*⟩

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* $\equiv \forall$ (*M* :: (*nat set*, *nat*) *model*) (*g* :: *nat* $\Rightarrow$ -) *w*. *M*, *g*, *w* $\models$ *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 ‹$i \notin$ *nominals p*›
  shows ‹*valid p* $\longleftrightarrow$ *nominals p*, *1* $\vdash$ [([¬ *p*], *i*)]›
⟨*proof*⟩

The restricted validity implies validity in general.

theorem *valid-semantics*:
$\langle$*valid p* $\longrightarrow M,\ g,\ w \models p\rangle$
$\langle$*proof*$\rangle$

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.