

Modal Logics for Nominal Transition Systems

Tjark Weber et al.

October 21, 2018

Abstract

These Isabelle theories formalize a modal logic for nominal transition systems, as presented in the paper *Modal Logics for Nominal Transition Systems* by Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramūnas Gutkovas, and Tjark Weber [1].

Contents

1	Bounded Sets Equipped With a Permutation Action	4
2	Lemmas about Well-Foundedness and Permutations	5
2.1	Hull and well-foundedness	5
3	Residuals	7
3.1	Binding names	7
3.2	Raw residuals and α -equivalence	7
3.3	Residuals	8
3.4	Notation for pairs as residuals	9
3.5	Support of residuals	9
3.6	Equality between residuals	9
3.7	Strong induction	10
3.8	Other lemmas	11
4	Nominal Transition Systems and Bisimulations	11
4.1	Basic Lemmas	11
4.2	Nominal transition systems	12
4.3	Bisimulations	12
5	Infinitary Formulas	16
5.1	Infinitely branching trees	16
5.2	Trees modulo α -equivalence	18
5.3	Constructors for trees modulo α -equivalence	32
5.4	Induction over trees modulo α -equivalence	35
5.5	Hereditarily finitely supported trees	36

5.6	Infinitary formulas	37
5.7	Constructors for infinitary formulas	39
5.8	Induction over infinitary formulas	43
5.9	Strong induction over infinitary formulas	44
6	Validity	46
6.1	Validity for infinitely branching trees	48
6.2	Validity for trees modulo α -equivalence	50
6.3	Validity for infinitary formulas	51
7	(Strong) Logical Equivalence	54
8	Bisimilarity Implies Logical Equivalence	54
9	Logical Equivalence Implies Bisimilarity	56
10	Finitely Supported Sets	60
11	Nominal Transition Systems with Effects and F/L-Bisimilarity	61
11.1	Nominal transition systems with effects	61
11.2	L -bisimulations and F/L -bisimilarity	62
12	Infinitary Formulas With Effects	67
12.1	Infinitely branching trees	67
12.2	Trees modulo α -equivalence	69
12.3	Constructors for trees modulo α -equivalence	84
12.4	Induction over trees modulo α -equivalence	87
12.5	Hereditarily finitely supported trees	88
12.6	Infinitary formulas	89
12.7	Constructors for infinitary formulas	91
12.8	F/L -formulas	95
12.9	Induction over infinitary formulas	96
12.10	Strong induction over infinitary formulas	96
13	Validity With Effects	96
13.1	Validity for infinitely branching trees	98
13.2	Validity for trees modulo α -equivalence	101
13.3	Validity for infinitary formulas	102
14	(Strong) Logical Equivalence	105
15	F/L-Bisimilarity Implies Logical Equivalence	105
16	Logical Equivalence Implies F/L-Bisimilarity	107

17	<i>L</i>-Transform	112
17.1	States	112
17.2	Actions and binding names	113
17.3	Satisfaction	115
17.4	Transitions	115
17.5	Translation of <i>F/L</i> -formulas into formulas without effects . .	118
17.6	Bisimilarity in the <i>L</i> -transform	125
18	Nominal Transition Systems and Bisimulations with Unobservable Transitions	131
18.1	Nominal transition systems with unobservable transitions . .	131
18.2	Weak bisimulations	133
19	Weak Formulas	139
19.1	Disjunction	139
19.2	Lemmas about α -equivalence involving τ	139
19.3	Weak action modality	140
19.4	Weak formulas	143
20	Weak Validity	145
21	Weak Logical Equivalence	149
22	Weak Bisimilarity Implies Weak Logical Equivalence	150
23	Weak Logical Equivalence Implies Weak Bisimilarity	152

```

theory Nominal-Bounded-Set
imports
  Nominal2.Nominal2
  HOL-Cardinals.Bounded-Set
begin

```

1 Bounded Sets Equipped With a Permutation Action

Additional lemmas about bounded sets.

```

interpretation bset-lifting: bset-lifting .

```

```

lemma Abs-bset-inverse' [simp]:
  assumes |A| <o natLeq +c |UNIV :: 'k set|
  shows set-bset (Abs-bset A :: 'a set['k]) = A
by (metis Abs-bset-inverse assms mem-Collect-eq)

```

Bounded sets are equipped with a permutation action, provided their elements are.

```

instantiation bset :: (pt,type) pt
begin

```

```

  lift-definition permute-bset :: perm  $\Rightarrow$  'a set['b]  $\Rightarrow$  'a set['b] is
    permute

```

```

proof -

```

```

  fix p and A :: 'a set

```

```

  have |p · A| ≤o |A| by (simp add: permute-set-eq-image)

```

```

  also assume |A| <o natLeq +c |UNIV :: 'b set|

```

```

  finally show |p · A| <o natLeq +c |UNIV :: 'b set| .

```

```

qed

```

```

instance

```

```

by standard (transfer, simp)+

```

```

end

```

```

lemma Abs-bset-eqvt [simp]:
  assumes |A| <o natLeq +c |UNIV :: 'k set|
  shows p · (Abs-bset A :: 'a::pt set['k]) = Abs-bset (p · A)
by (simp add: permute-bset-def map-bset-def image-def permute-set-def) (metis
  (no-types, lifting) Abs-bset-inverse' assms)

```

```

lemma supp-Abs-bset [simp]:

```

```

  assumes |A| <o natLeq +c |UNIV :: 'k set|

```

```

  shows supp (Abs-bset A :: 'a::pt set['k]) = supp A

```

```

proof -

```

```

  from assms have  $\bigwedge p. p \cdot (\text{Abs-bset } A :: 'a::\text{pt set}['k]) \neq \text{Abs-bset } A \iff p \cdot A$ 

```

$\neq A$
by *simp* (*metis map-bset.rep-eq permute-set-eq-image set-bset-inverse set-bset-to-set-bset*)
then show *?thesis*
unfolding *supp-def* **by** *simp*
qed

lemma *map-bset-permute*: $p \cdot B = \text{map-bset } (\text{permute } p) B$
by *transfer* (*auto simp add: image-def permute-set-def*)

lemma *set-bset-eqv* [*eqvt*]:
 $p \cdot \text{set-bset } B = \text{set-bset } (p \cdot B)$
by *transfer simp*

lemma *map-bset-eqv* [*eqvt*]:
 $p \cdot \text{map-bset } f B = \text{map-bset } (p \cdot f) (p \cdot B)$
by *transfer simp*

lemma *bempty-eqv* [*eqvt*]: $p \cdot \text{bempty} = \text{bempty}$
by *transfer simp*

lemma *binsert-eqv* [*eqvt*]: $p \cdot (\text{binsert } x B) = \text{binsert } (p \cdot x) (p \cdot B)$
by *transfer simp*

lemma *bsingleton-eqv* [*eqvt*]: $p \cdot \text{bsingleton } x = \text{bsingleton } (p \cdot x)$
by (*simp add: map-bset-permute*)

end
theory *Nominal-Wellfounded*
imports
Nominal2.Nominal2
begin

2 Lemmas about Well-Foundedness and Permutations

definition *less-bool-rel* :: *bool rel* **where**
 $\text{less-bool-rel} \equiv \{(x,y). x < y\}$

lemma *less-bool-rel-iff* [*simp*]:
 $(a,b) \in \text{less-bool-rel} \longleftrightarrow \neg a \wedge b$
by (*metis less-bool-def less-bool-rel-def mem-Collect-eq split-conv*)

lemma *wf-less-bool-rel*: *wf less-bool-rel*
by (*metis less-bool-rel-iff wfUNIVI*)

2.1 Hull and well-foundedness

inductive-set *hull-rel* **where**

$(p \cdot x, x) \in \text{hull-rel}$

lemma *hull-relp-reflp*: *reflp hull-relp*
by (*metis hull-relp.intros permute-zero reflpI*)

lemma *hull-relp-symp*: *symp hull-relp*
by (*metis (poly-guards-query) hull-relp.simps permute-minus-cancel(2) sympI*)

lemma *hull-relp-transp*: *transp hull-relp*
by (*metis (full-types) hull-relp.simps permute-plus transpI*)

lemma *hull-relp-equivp*: *equivp hull-relp*
by (*metis equivpI hull-relp-reflp hull-relp-symp hull-relp-transp*)

lemma *hull-rel-relcomp-subset*:

assumes *eqvt R*

shows $R \circ \text{hull-rel} \subseteq \text{hull-rel} \circ R$

proof

fix *x*

assume $x \in R \circ \text{hull-rel}$

then obtain $x1\ x2\ y$ **where** $x = (x1, x2)$ **and** $R: (x1, y) \in R$ **and** $(y, x2) \in \text{hull-rel}$

by *auto*

then obtain p **where** $y = p \cdot x2$

by (*metis hull-rel.simps*)

then have $-p \cdot y = x2$

by (*metis permute-minus-cancel(2)*)

then have $(-p \cdot x1, x2) \in R$

using R **assms** **by** (*metis Pair-eqvt eqvt-def mem-permute-iff*)

moreover have $(x1, -p \cdot x1) \in \text{hull-rel}$

by (*metis hull-rel.intros permute-minus-cancel(2)*)

ultimately show $x \in \text{hull-rel} \circ R$

using x **by** *auto*

qed

lemma *wf-hull-rel-relcomp*:

assumes *wf R and eqvt R*

shows *wf (hull-rel O R)*

using *assms* **by** (*metis hull-rel-relcomp-subset wf-relcomp-compatible*)

lemma *hull-rel-relcompI [simp]*:

assumes $(x, y) \in R$

shows $(p \cdot x, y) \in \text{hull-rel} \circ R$

using *assms* **by** (*metis hull-rel.intros relcomp.relcompI*)

lemma *hull-rel-relcomp-trivialI [simp]*:

assumes $(x, y) \in R$

shows $(x, y) \in \text{hull-rel} \circ R$

using *assms* **by** (*metis hull-rel-relcompI permute-zero*)

```

end
theory Residual
imports
  Nominal2.Nominal2
begin

```

3 Residuals

3.1 Binding names

To define α -equivalence, we require actions to be equipped with an equivariant function bn that gives their binding names. Actions may only bind finitely many names. This is necessary to ensure that we can use a finite permutation to rename the binding names in an action.

```

class bn = fs +
  fixes bn :: 'a  $\Rightarrow$  atom set
  assumes bn-eqvt:  $p \cdot (bn \alpha) = bn (p \cdot \alpha)$ 
  and bn-finite: finite (bn  $\alpha$ )

```

```

lemma bn-subset-supp:  $bn \alpha \subseteq supp \alpha$ 
by (metis (erased, hide-lams) bn-eqvt bn-finite eqvt-at-def finite-supp supp-eqvt-at
supp-finite-atom-set)

```

3.2 Raw residuals and α -equivalence

Raw residuals are simply pairs of actions and states. Binding names in the action bind into (the action and) the state.

```

fun alpha-residual :: ('act::bn  $\times$  'state::pt)  $\Rightarrow$  ('act  $\times$  'state)  $\Rightarrow$  bool where
  alpha-residual ( $\alpha 1, P1$ ) ( $\alpha 2, P2$ )  $\longleftrightarrow$   $[bn \alpha 1]set. (\alpha 1, P1) = [bn \alpha 2]set. (\alpha 2,$ 
 $P2)$ 

```

α -equivalence is equivariant.

```

lemma alpha-residual-eqvt [eqvt]:
  assumes alpha-residual r1 r2
  shows alpha-residual (p  $\cdot$  r1) (p  $\cdot$  r2)
using assms by (cases r1, cases r2) (simp, metis Pair-eqvt bn-eqvt permute-Abs-set)

```

α -equivalence is an equivalence relation.

```

lemma alpha-residual-reflp: reflp alpha-residual
by (metis alpha-residual.simps prod.exhaust reflpI)

```

```

lemma alpha-residual-symp: symp alpha-residual
by (metis alpha-residual.simps prod.exhaust sympI)

```

```

lemma alpha-residual-transp: transp alpha-residual
by (rule transpI) (metis alpha-residual.simps prod.exhaust)

```

lemma *alpha-residual-equivp*: *equivp alpha-residual*
by (*metis alpha-residual-reflp alpha-residual-symp alpha-residual-transp equivpI*)

3.3 Residuals

Residuals are raw residuals quotiented by α -equivalence.

quotient-type

('act,'state) residual = 'act::bn × 'state::pt / alpha-residual
by (*fact alpha-residual-equivp*)

lemma *residual-abs-rep* [*simp*]: *abs-residual (rep-residual res) = res*
by (*metis Quotient-residual Quotient-abs-rep*)

lemma *residual-rep-abs* [*simp*]: *alpha-residual (rep-residual (abs-residual r)) r*
by (*metis residual.abs-eq-iff residual-abs-rep*)

The permutation operation is lifted from raw residuals.

instantiation *residual* :: (*bn,pt*) *pt*
begin

lift-definition *permute-residual* :: *perm* \Rightarrow (*'a,'b*) *residual* \Rightarrow (*'a,'b*) *residual*
is *permute*
by (*fact alpha-residual-eqvt*)

instance

proof

fix *res* :: (*-,-*) *residual*

show $0 \cdot res = res$

by *transfer (metis alpha-residual-equivp equivp-reflp permute-zero)*

next

fix *p q* :: *perm* **and** *res* :: (*-,-*) *residual*

show $(p + q) \cdot res = p \cdot q \cdot res$

by *transfer (metis alpha-residual-equivp equivp-reflp permute-plus)*

qed

end

The abstraction function from raw residuals to residuals is equivariant. The representation function is equivariant modulo α -equivalence.

lemmas *permute-residual.abs-eq* [*eqvt, simp*]

lemma *alpha-residual-permute-rep-commute* [*simp*]: *alpha-residual (p · rep-residual res) (rep-residual (p · res))*
by (*metis residual.abs-eq-iff residual-abs-rep permute-residual.abs-eq*)

3.4 Notation for pairs as residuals

abbreviation *abs-residual-pair* :: 'act::bn \Rightarrow 'state::pt \Rightarrow ('act,'state) residual
 $\langle \langle -, - \rangle [0,0] 1000 \rangle$

where

$\langle \alpha, P \rangle == \text{abs-residual } (\alpha, P)$

lemma *abs-residual-pair-eqvt* [simp]: $p \cdot \langle \alpha, P \rangle = \langle p \cdot \alpha, p \cdot P \rangle$

by (*metis Pair-eqvt permute-residual.abs-eq*)

3.5 Support of residuals

We only consider finitely supported states now.

lemma *supp-abs-residual-pair*: $\text{supp } \langle \alpha, P :: 'state::fs \rangle = \text{supp } (\alpha, P) - \text{bn } \alpha$

proof –

have $\text{supp } \langle \alpha, P \rangle = \text{supp } ([\text{bn } \alpha] \text{set. } (\alpha, P))$

by (*simp add: supp-def residual.abs-eq-iff bn-eqvt*)

then show *?thesis* **by** (*simp add: supp-Abs*)

qed

lemma *bn-abs-residual-fresh* [simp]: $\text{bn } \alpha \#* \langle \alpha, P :: 'state::fs \rangle$

by (*simp add: fresh-star-def fresh-def supp-abs-residual-pair*)

lemma *finite-supp-abs-residual-pair* [simp]: $\text{finite } (\text{supp } \langle \alpha, P :: 'state::fs \rangle)$

by (*metis finite-Diff finite-supp supp-abs-residual-pair*)

3.6 Equality between residuals

lemma *residual-eq-iff-perm*: $\langle \alpha 1, P 1 \rangle = \langle \alpha 2, P 2 \rangle \longleftrightarrow$

$(\exists p. \text{supp } (\alpha 1, P 1) - \text{bn } \alpha 1 = \text{supp } (\alpha 2, P 2) - \text{bn } \alpha 2 \wedge (\text{supp } (\alpha 1, P 1) - \text{bn } \alpha 1) \#* p \wedge p \cdot (\alpha 1, P 1) = (\alpha 2, P 2) \wedge p \cdot \text{bn } \alpha 1 = \text{bn } \alpha 2)$

(**is** *?l* \longleftrightarrow *?r*)

proof

assume *: *?l*

then have $[\text{bn } \alpha 1] \text{set. } (\alpha 1, P 1) = [\text{bn } \alpha 2] \text{set. } (\alpha 2, P 2)$

by (*simp add: residual.abs-eq-iff*)

then obtain *p* **where** $(\text{bn } \alpha 1, (\alpha 1, P 1)) \approx_{\text{set}} ((=)) \text{supp } p (\text{bn } \alpha 2, (\alpha 2, P 2))$

using *Abs-eq-iff(1)* **by** *blast*

then show *?r*

by (*metis (mono-tags, lifting) alpha-set.simps*)

next

assume *: *?r*

then obtain *p* **where** $(\text{bn } \alpha 1, (\alpha 1, P 1)) \approx_{\text{set}} ((=)) \text{supp } p (\text{bn } \alpha 2, (\alpha 2, P 2))$

using *alpha-set.simps* **by** *blast*

then have $[\text{bn } \alpha 1] \text{set. } (\alpha 1, P 1) = [\text{bn } \alpha 2] \text{set. } (\alpha 2, P 2)$

using *Abs-eq-iff(1)* **by** *blast*

then show *?l*

by (*simp add: residual.abs-eq-iff*)

qed

lemma *residual-eq-iff-perm-renaming*: $\langle \alpha 1, P1 \rangle = \langle \alpha 2, P2 \rangle \longleftrightarrow$
 $(\exists p. \text{supp } (\alpha 1, P1) - \text{bn } \alpha 1 = \text{supp } (\alpha 2, P2) - \text{bn } \alpha 2 \wedge (\text{supp } (\alpha 1, P1) - \text{bn } \alpha 1) \#* p \wedge p \cdot (\alpha 1, P1) = (\alpha 2, P2) \wedge p \cdot \text{bn } \alpha 1 = \text{bn } \alpha 2 \wedge \text{supp } p \subseteq \text{bn } \alpha 1 \cup p \cdot \text{bn } \alpha 1)$
(is ?l \longleftrightarrow ?r)

proof
assume ?l
then obtain p **where** $p: \text{supp } (\alpha 1, P1) - \text{bn } \alpha 1 = \text{supp } (\alpha 2, P2) - \text{bn } \alpha 2 \wedge (\text{supp } (\alpha 1, P1) - \text{bn } \alpha 1) \#* p \wedge p \cdot (\alpha 1, P1) = (\alpha 2, P2) \wedge p \cdot \text{bn } \alpha 1 = \text{bn } \alpha 2$
by (*metis residual-eq-iff-perm*)
moreover obtain q **where** $q-p: \forall b \in \text{bn } \alpha 1. q \cdot b = p \cdot b$ **and** $\text{supp } q: \text{supp } q \subseteq \text{bn } \alpha 1 \cup p \cdot \text{bn } \alpha 1$
by (*metis set-renaming-perm2*)
have $\text{supp } q \subseteq \text{supp } p$
proof
fix a **assume** $*$: $a \in \text{supp } q$ **then show** $a \in \text{supp } p$
proof (*cases* $a \in \text{bn } \alpha 1$)
case *True* **then show** ?thesis
using $*$ $q-p$ **by** (*metis mem-Collect-eq supp-perm*)
next
case *False* **then have** $a \in p \cdot \text{bn } \alpha 1$
using $*$ $\text{supp } q$ **using** *UnE subsetCE* **by** *blast*
with *False* **have** $p \cdot a \neq a$
by (*metis mem-permute-iff*)
then show ?thesis
using *fresh-def fresh-perm* **by** *blast*
qed
qed
with p **have** $(\text{supp } (\alpha 1, P1) - \text{bn } \alpha 1) \#* q$
by (*meson fresh-def fresh-star-def subset-iff*)
moreover with p **and** $q-p$ **have** $\bigwedge a. a \in \text{supp } \alpha 1 \implies q \cdot a = p \cdot a$ **and** $\bigwedge a. a \in \text{supp } P1 \implies q \cdot a = p \cdot a$
by (*metis Diff-iff fresh-perm fresh-star-def UnCI supp-Pair*)+
then have $q \cdot \alpha 1 = p \cdot \alpha 1$ **and** $q \cdot P1 = p \cdot P1$
by (*metis supp-perm-perm-eq*)+
ultimately show ?r
using $\text{supp } q$ **by** (*metis Pair-eqvt bn-eqvt*)
next
assume ?r **then show** ?l
by (*meson residual-eq-iff-perm*)
qed

3.7 Strong induction

lemma *residual-strong-induct*:

assumes $\bigwedge (\text{act}::'act::\text{bn}) (\text{state}::'state::\text{fs}) (c::'a::\text{fs}). \text{bn } \text{act} \#* c \implies P c \langle \text{act}, \text{state} \rangle$
shows $P c$ *residual*

proof (*rule residual.abs-induct, clarify*)

```

fix act :: 'act and state :: 'state
obtain p where 1: (p · bn act) #* c and 2: supp ⟨act,state⟩ #* p
  proof (rule at-set-avoiding2[of bn act c ⟨act,state⟩, THEN exE])
    show finite (bn act) by (fact bn-finite)
  next
    show finite (supp c) by (fact finite-supp)
  next
    show finite (supp ⟨act,state⟩) by (fact finite-supp-abs-residual-pair)
  next
    show bn act #* ⟨act,state⟩ by (fact bn-abs-residual-fresh)
qed metis
from 2 have ⟨p · act, p · state⟩ = ⟨act,state⟩
  using supp-perm-eq by fastforce
then show P c ⟨act,state⟩
  using assms 1 by (metis bn-eqvt)
qed

```

3.8 Other lemmas

```

lemma residual-empty-bn-eq-iff:
  assumes bn α1 = {}
  shows ⟨α1,P1⟩ = ⟨α2,P2⟩ ⟷ α1 = α2 ∧ P1 = P2
proof
  assume ⟨α1,P1⟩ = ⟨α2,P2⟩
  with assms have [{}]set. (α1, P1) = [bn α2]set. (α2, P2)
    by (simp add: residual.abs-eq-iff)
  then obtain p where ({}, (α1, P1)) ≈set ((=)) supp p (bn α2, (α2, P2))
    using Abs-eq-iff(1) by blast
  then show α1 = α2 ∧ P1 = P2
    unfolding alpha-set using supp-perm-eq by fastforce
next
  assume α1 = α2 ∧ P1 = P2 then show ⟨α1,P1⟩ = ⟨α2,P2⟩
    by simp
qed

end
theory Transition-System
imports
  Residual
begin

```

4 Nominal Transition Systems and Bisimulations

4.1 Basic Lemmas

```

lemma symp-eqvt [eqvt]:
  assumes symp R shows symp (p · R)
  using assms unfolding symp-def by (subst permute-fun-def)+ (simp add: permute-pure)

```

4.2 Nominal transition systems

```

locale nominal-ts =
  fixes satisfies :: 'state::fs  $\Rightarrow$  'pred::fs  $\Rightarrow$  bool (infix  $\vdash$  70)
    and transition :: 'state  $\Rightarrow$  ('act::bn,'state) residual  $\Rightarrow$  bool (infix  $\rightarrow$  70)
  assumes satisfies-eqvt [eqvt]:  $P \vdash \varphi \Longrightarrow p \cdot P \vdash p \cdot \varphi$ 
    and transition-eqvt [eqvt]:  $P \rightarrow \alpha Q \Longrightarrow p \cdot P \rightarrow p \cdot \alpha Q$ 
begin

  lemma transition-eqvt':
    assumes  $P \rightarrow \langle \alpha, Q \rangle$  shows  $p \cdot P \rightarrow \langle p \cdot \alpha, p \cdot Q \rangle$ 
    using assms by (metis abs-residual-pair-eqvt transition-eqvt)

end

```

4.3 Bisimulations

```

context nominal-ts
begin

```

```

definition is-bisimulation :: ('state  $\Rightarrow$  'state  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  is-bisimulation  $R \equiv$ 
    symp  $R \wedge$ 
    ( $\forall P Q. R P Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)$ )  $\wedge$ 
    ( $\forall P Q. R P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \rightarrow$ 
 $\langle \alpha, Q' \rangle \wedge R P' Q'))$ )

```

```

definition bisimilar :: 'state  $\Rightarrow$  'state  $\Rightarrow$  bool (infix  $\sim$  100) where
   $P \sim Q \equiv \exists R. \text{is-bisimulation } R \wedge R P Q$ 

```

(\sim) is an equivariant equivalence relation.

```

lemma is-bisimulation-eqvt :
  assumes is-bisimulation  $R$  shows is-bisimulation ( $p \cdot R$ )
  using assms unfolding is-bisimulation-def
  proof (clarify)
    assume 1: symp  $R$ 
    assume 2:  $\forall P Q. R P Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)$ 
    assume 3:  $\forall P Q. R P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'.$ 
 $Q \rightarrow \langle \alpha, Q' \rangle \wedge R P' Q'))$ 
    have symp ( $p \cdot R$ ) (is ?S)
    using 1 by (simp add: symp-eqvt)
    moreover have  $\forall P Q. (p \cdot R) P Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)$  (is ?T)
    proof (clarify)
      fix  $P Q \varphi$ 
      assume *: ( $p \cdot R$ )  $P Q$  and **:  $P \vdash \varphi$ 
      from * have  $R (-p \cdot P) (-p \cdot Q)$ 
      by (simp add: eqvt-lambda permute-bool-def unpermute-def)
      then show  $Q \vdash \varphi$ 
      using 2 ** by (metis permute-minus-cancel(1) satisfies-eqvt)
    qed

```

moreover have $\forall P Q. (p \cdot R) P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P \rangle)$
 $\longrightarrow (\exists Q'. Q \rightarrow \langle \alpha, Q \rangle \wedge (p \cdot R) P' Q')$ (**is** ?U)
proof (*clarify*)
fix $P Q \alpha P'$
assume *: $(p \cdot R) P Q$ **and** **: $\text{bn } \alpha \#* Q$ **and** ***: $P \rightarrow \langle \alpha, P \rangle$
from * **have** $R (-p \cdot P) (-p \cdot Q)$
by (*simp add: eqvt-lambda permute-bool-def unpermute-def*)
moreover have $\text{bn } (-p \cdot \alpha) \#* (-p \cdot Q)$
using ** **by** (*metis bn-eqvt fresh-star-permute-iff*)
moreover have $-p \cdot P \rightarrow \langle -p \cdot \alpha, -p \cdot P \rangle$
using *** **by** (*metis transition-eqvt'*)
ultimately obtain Q' **where** $T: -p \cdot Q \rightarrow \langle -p \cdot \alpha, Q \rangle$ **and** $R: R (-p \cdot P) Q'$
using \exists **by** *metis*
from T **have** $Q \rightarrow \langle \alpha, p \cdot Q \rangle$
by (*metis permute-minus-cancel(1) transition-eqvt'*)
moreover from R **have** $(p \cdot R) P' (p \cdot Q')$
by (*metis eqvt-apply eqvt-lambda permute-bool-def unpermute-def*)
ultimately show $\exists Q'. Q \rightarrow \langle \alpha, Q \rangle \wedge (p \cdot R) P' Q'$
by *metis*
qed
ultimately show ?S \wedge ?T \wedge ?U **by** *simp*
qed

lemma *bisimilar-eqvt* :

assumes $P \sim Q$ **shows** $(p \cdot P) \sim (p \cdot Q)$

proof –

from *assms* **obtain** R **where** *: *is-bisimulation* $R \wedge R P Q$

unfolding *bisimilar-def* ..

then have *is-bisimulation* $(p \cdot R)$

by (*simp add: is-bisimulation-eqvt*)

moreover from * **have** $(p \cdot R) (p \cdot P) (p \cdot Q)$

by (*metis eqvt-apply permute-boolI*)

ultimately show $(p \cdot P) \sim (p \cdot Q)$

unfolding *bisimilar-def* **by** *auto*

qed

lemma *bisimilar-reflp*: *reflp bisimilar*

proof (*rule reflpI*)

fix x

have *is-bisimulation* $(=)$

unfolding *is-bisimulation-def* **by** (*simp add: symp-def*)

then show $x \sim x$

unfolding *bisimilar-def* **by** *auto*

qed

lemma *bisimilar-symp*: *symp bisimilar*

proof (*rule sympI*)

fix $P Q$

```

assume  $P \sim Q$ 
then obtain  $R$  where  $*$ : is-bisimulation  $R \wedge R P Q$ 
  unfolding bisimilar-def ..
then have  $R Q P$ 
  unfolding is-bisimulation-def by (simp add: symp-def)
with  $*$  show  $Q \sim P$ 
  unfolding bisimilar-def by auto
qed

lemma bisimilar-is-bisimulation: is-bisimulation bisimilar
unfolding is-bisimulation-def proof
  show symp ( $\sim$ )
  by (fact bisimilar-symp)
next
  show  $(\forall P Q. P \sim Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)) \wedge$ 
     $(\forall P Q. P \sim Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P \rangle \longrightarrow (\exists Q'. Q \rightarrow$ 
 $\langle \alpha, Q \rangle \wedge P' \sim Q')))$ 
  by (auto simp add: is-bisimulation-def bisimilar-def) blast
qed

lemma bisimilar-transp: transp bisimilar
proof (rule transpI)
  fix  $P Q R$ 
  assume  $PQ: P \sim Q$  and  $QR: Q \sim R$ 
  let  $?bisim = \text{bisimilar } OO \text{ bisimilar}$ 
  have symp  $?bisim$ 
  proof (rule sympI)
    fix  $P R$ 
    assume  $?bisim P R$ 
    then obtain  $Q$  where  $P \sim Q$  and  $Q \sim R$ 
    by blast
    then have  $R \sim Q$  and  $Q \sim P$ 
    by (metis bisimilar-symp sympE)+
    then show  $?bisim R P$ 
    by blast
  qed
  moreover have  $\forall P Q. ?bisim P Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)$ 
  using bisimilar-is-bisimulation is-bisimulation-def by auto
  moreover have  $\forall P Q. ?bisim P Q \longrightarrow$ 
     $(\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P \rangle \longrightarrow (\exists Q'. Q \rightarrow \langle \alpha, Q \rangle \wedge ?bisim P'$ 
 $Q'))$ 
  proof (clarify)
    fix  $P R Q \alpha P'$ 
    assume  $PR: P \sim R$  and  $RQ: R \sim Q$  and fresh:  $\text{bn } \alpha \#* Q$  and trans:  $P$ 
 $\rightarrow \langle \alpha, P \rangle$ 
    — rename  $\langle \alpha, P \rangle$  to avoid  $R$ , without touching  $Q$ 
    obtain  $p$  where  $1: (p \cdot \text{bn } \alpha) \#* R$  and  $2: \text{supp } (\langle \alpha, P \rangle, Q) \#* p$ 
    proof (rule at-set-avoiding2[of  $\text{bn } \alpha R (\langle \alpha, P \rangle, Q)$ , THEN exE])
      show finite  $(\text{bn } \alpha)$  by (fact bn-finite)

```

```

next
  show finite (supp R) by (fact finite-supp)
next
  show finite (supp (( $\alpha, P^\wedge$ ), Q)) by (simp add: finite-supp supp-Pair)
next
  show bn  $\alpha \#^* ((\alpha, P^\wedge), Q)$  by (simp add: fresh fresh-star-Pair)
qed metis
from 2 have 3: supp  $\langle \alpha, P^\wedge \rangle \#^* p$  and 4: supp Q  $\#^* p$ 
  by (simp add: fresh-star-Un supp-Pair)+
from 3 have  $\langle p \cdot \alpha, p \cdot P^\wedge \rangle = \langle \alpha, P^\wedge \rangle$ 
  using supp-perm-eq by fastforce
then obtain  $pR'$  where 5:  $R \rightarrow \langle p \cdot \alpha, pR^\wedge \rangle$  and 6:  $(p \cdot P^\wedge) \sim pR'$ 
  using PR trans 1 by (metis (mono-tags, lifting) bisimilar-is-bisimulation
bn-eqt is-bisimulation-def)
  from fresh and 4 have bn  $(p \cdot \alpha) \#^* Q$ 
    by (metis bn-eqt fresh-star-permute-iff supp-perm-eq)
  then obtain  $pQ'$  where 7:  $Q \rightarrow \langle p \cdot \alpha, pQ^\wedge \rangle$  and 8:  $pR' \sim pQ'$ 
  using RQ 5 by (metis (full-types) bisimilar-is-bisimulation is-bisimulation-def)
  from 7 have  $Q \rightarrow \langle \alpha, -p \cdot pQ^\wedge \rangle$ 
    using 4 by (metis permute-minus-cancel(2) supp-perm-eq transition-eqt')
  moreover from 6 and 8 have  $?bisim P' (-p \cdot pQ^\wedge)$ 
    by (metis (no-types, hide-lams) bisimilar-eqt permute-minus-cancel(2)
relcompp.simps)
  ultimately show  $\exists Q'. Q \rightarrow \langle \alpha, Q^\wedge \rangle \wedge ?bisim P' Q'$ 
    by metis
  qed
ultimately have is-bisimulation  $?bisim$ 
  unfolding is-bisimulation-def by metis
moreover have  $?bisim P R$ 
  using PQ QR by blast
ultimately show  $P \sim R$ 
  unfolding bisimilar-def by meson
qed

lemma bisimilar-equivp: equivp bisimilar
by (metis bisimilar-reflp bisimilar-symp bisimilar-transp equivp-reflp-symp-transp)

lemma bisimilar-simulation-step:
  assumes  $P \sim Q$  and bn  $\alpha \#^* Q$  and  $P \rightarrow \langle \alpha, P^\wedge \rangle$ 
  obtains  $Q'$  where  $Q \rightarrow \langle \alpha, Q^\wedge \rangle$  and  $P' \sim Q'$ 
  using assms by (metis (poly-guards-query) bisimilar-is-bisimulation is-bisimulation-def)

end

end
theory Formula
imports
  Nominal-Bounded-Set
  Nominal-Wellfounded

```

Residual
begin

5 Infinitary Formulas

5.1 Infinitely branching trees

First, we define a type of trees, with a constructor $tConj$ that maps (potentially infinite) sets of trees into trees. To avoid paradoxes (note that there is no injection from the powerset of trees into the set of trees), the cardinality of the argument set must be bounded.

```
datatype ('idx,'pred,'act) Tree =
  tConj ('idx,'pred,'act) Tree set['idx] — potentially infinite sets of trees
| tNot ('idx,'pred,'act) Tree
| tPred 'pred
| tAct 'act ('idx,'pred,'act) Tree
```

The (automatically generated) induction principle for trees allows us to prove that the following relation over trees is well-founded. This will be useful for termination proofs when we define functions by recursion over trees.

```
inductive-set Tree-wf :: ('idx,'pred,'act) Tree rel where
  t ∈ set-bset tset ⇒ (t, tConj tset) ∈ Tree-wf
| (t, tNot t) ∈ Tree-wf
| (t, tAct α t) ∈ Tree-wf
```

lemma wf-Tree-wf: wf Tree-wf

unfolding wf-def

proof (rule allI, rule impI, rule allI)

fix P :: ('idx,'pred,'act) Tree ⇒ bool **and** t

assume ∀ x. (∀ y. (y, x) ∈ Tree-wf ⇒ P y) ⇒ P x

then show P t

proof (induction t)

case tConj **then show** ?case

by (metis Tree.distinct(2) Tree.distinct(5) Tree.inject(1) Tree-wf.cases)

next

case tNot **then show** ?case

by (metis Tree.distinct(1) Tree.distinct(9) Tree.inject(2) Tree-wf.cases)

next

case tPred **then show** ?case

by (metis Tree.distinct(11) Tree.distinct(3) Tree.distinct(7) Tree-wf.cases)

next

case tAct **then show** ?case

by (metis Tree.distinct(10) Tree.distinct(6) Tree.inject(4) Tree-wf.cases)

qed

qed

We define a permutation operation on the type of trees.

instantiation *Tree* :: (*type*, *pt*, *pt*) *pt*
begin

primrec *permute-Tree* :: *perm* \Rightarrow (*-, -, -*) *Tree* \Rightarrow (*-, -, -*) *Tree* **where**
 $p \cdot (tConj\ tset) = tConj\ (map-bset\ (permute\ p)\ tset)$ — neat trick to get
around the fact that *tset* is not of permutation type yet
| $p \cdot (tNot\ t) = tNot\ (p \cdot t)$
| $p \cdot (tPred\ \varphi) = tPred\ (p \cdot \varphi)$
| $p \cdot (tAct\ \alpha\ t) = tAct\ (p \cdot \alpha)\ (p \cdot t)$

instance

proof

fix *t* :: (*-, -, -*) *Tree*
show $0 \cdot t = t$
proof (*induction t*)
 case *tConj* **then show** ?*case*
 by (*simp*, *transfer*) (*auto simp: image-def*)
qed *simp-all*
next
fix *p q* :: *perm* **and** *t* :: (*-, -, -*) *Tree*
show $(p + q) \cdot t = p \cdot q \cdot t$
proof (*induction t*)
 case *tConj* **then show** ?*case*
 by (*simp*, *transfer*) (*auto simp: image-def*)
qed *simp-all*
qed

end

Now that the type of trees—and hence the type of (bounded) sets of trees—is a permutation type, we can massage the definition of $p \cdot tConj\ tset$ into its more usual form.

lemma *permute-Tree-tConj* [*simp*]: $p \cdot tConj\ tset = tConj\ (p \cdot tset)$
by (*simp add: map-bset-permute*)

declare *permute-Tree.simps(1)* [*simp del*]

The relation *Tree-wf* is equivariant.

lemma *Tree-wf-eqt-aux*:

assumes $(t1, t2) \in Tree-wf$ **shows** $(p \cdot t1, p \cdot t2) \in Tree-wf$
using *assms* **proof** (*induction rule: Tree-wf.induct*)
fix *t* :: (*'a, 'b, 'c*) *Tree* **and** *tset* :: (*'a, 'b, 'c*) *Tree set['a]*
assume $t \in set-bset\ tset$ **then show** $(p \cdot t, p \cdot tConj\ tset) \in Tree-wf$
 by (*metis Tree-wf.intros(1) mem-permute-iff permute-Tree-tConj set-bset-eqt*)
next
fix *t* :: (*'a, 'b, 'c*) *Tree*
show $(p \cdot t, p \cdot tNot\ t) \in Tree-wf$
 by (*metis Tree-wf.intros(2) permute-Tree.simps(2)*)
next

```

fix t :: ('a,'b,'c) Tree and  $\alpha$ 
show (p · t, p · tAct  $\alpha$  t) ∈ Tree-wf
  by (metis Tree-wf.intros(3) permute-Tree.simps(4))
qed

```

```

lemma Tree-wf-eqt [eqvt, simp]: p · Tree-wf = Tree-wf
proof
  show p · Tree-wf ⊆ Tree-wf
    by (auto simp add: permute-set-def) (rule Tree-wf-eqt-aux)
next
  show Tree-wf ⊆ p · Tree-wf
    by (auto simp add: permute-set-def) (metis Tree-wf-eqt-aux permute-minus-cancel(1))
qed

```

```

lemma Tree-wf-eqt': eqvt Tree-wf
by (metis Tree-wf-eqt eqvtI)

```

The definition of *permute* for trees gives rise to the usual notion of support. The following lemmas, one for each constructor, describe the support of trees.

```

lemma supp-tConj [simp]: supp (tConj tset) = supp tset
unfolding supp-def by simp

```

```

lemma supp-tNot [simp]: supp (tNot t) = supp t
unfolding supp-def by simp

```

```

lemma supp-tPred [simp]: supp (tPred  $\varphi$ ) = supp  $\varphi$ 
unfolding supp-def by simp

```

```

lemma supp-tAct [simp]: supp (tAct  $\alpha$  t) = supp  $\alpha$  ∪ supp t
unfolding supp-def by (simp add: Collect-imp-eq Collect-neg-eq)

```

5.2 Trees modulo α -equivalence

We generalize the notion of support, which considers whether a permuted element is *equal* to itself, to arbitrary endorelations. This is available as *supp-rel* in Nominal Isabelle.

```

lemma supp-rel-eqvt [eqvt]:
  p · supp-rel R x = supp-rel (p · R) (p · x)
by (simp add: supp-rel-def)

```

Usually, the definition of α -equivalence presupposes a notion of free variables. However, the variables that are “free” in an infinitary conjunction are not necessarily those that are free in one of the conjuncts. For instance, consider a conjunction over *all* names. Applying any permutation will yield the same conjunction, i.e., this conjunction has *no* free variables.

To obtain the correct notion of free variables for infinitary conjunctions, we initially defined α -equivalence and free variables via mutual recursion. In

particular, we defined the free variables of a conjunction as term *fv-Tree* ($tConj\ tset$) = *supp-rel alpha-Tree* ($tConj\ tset$).

We then realized that it is not necessary to define the concept of “free variables” at all, but the definition of α -equivalence becomes much simpler (in particular, it is no longer mutually recursive) if we directly use the support modulo α -equivalence.

The following lemmas and constructions are used to prove termination of our definition.

lemma *supp-rel-cong* [*fundef-cong*]:

$\llbracket x=x'; \bigwedge a\ b.\ R\ ((a \Rightarrow b) \cdot x')\ x' \longleftrightarrow R'\ ((a \Rightarrow b) \cdot x')\ x' \rrbracket \Longrightarrow \text{supp-rel } R\ x = \text{supp-rel } R'\ x'$

by (*simp add: supp-rel-def*)

lemma *rel-bset-cong* [*fundef-cong*]:

$\llbracket x=x'; y=y'; \bigwedge a\ b.\ a \in \text{set-bset } x' \Longrightarrow b \in \text{set-bset } y' \Longrightarrow R\ a\ b \longleftrightarrow R'\ a\ b \rrbracket \Longrightarrow \text{rel-bset } R\ x\ y \longleftrightarrow \text{rel-bset } R'\ x'\ y'$

by (*simp add: rel-bset-def rel-set-def*)

lemma *alpha-set-cong* [*fundef-cong*]:

$\llbracket bs=bs'; x=x'; R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y'; f\ x' = f'\ x'; f\ y' = f'\ y'; p=p'; cs=cs'; y=y' \rrbracket \Longrightarrow$

$\text{alpha-set } (bs, x)\ R\ f\ p\ (cs, y) \longleftrightarrow \text{alpha-set } (bs', x')\ R'\ f'\ p'\ (cs', y')$

by (*simp add: alpha-set*)

quotient-type

$(\text{'idx, 'pred, 'act})\ \text{Tree}_p = (\text{'idx, 'pred::pt, 'act::bn})\ \text{Tree} / \text{hull-relp}$

by (*fact hull-relp-equivp*)

lemma *abs-Tree_p-eq* [*simp*]: $\text{abs-Tree}_p\ (p \cdot t) = \text{abs-Tree}_p\ t$

by (*metis hull-relp.simps Tree_p.abs-eq-iff*)

lemma *permute-rep-abs-Tree_p*:

obtains p **where** $\text{rep-Tree}_p\ (\text{abs-Tree}_p\ t) = p \cdot t$

by (*metis Quotient3-Tree_p Tree_p.abs-eq-iff rep-abs-rsp hull-relp.simps*)

lift-definition $\text{Tree-wf}_p :: (\text{'idx, 'pred::pt, 'act::bn})\ \text{Tree}_p\ \text{rel is}$

Tree-wf .

lemma *Tree-wf_pI* [*simp*]:

assumes $(a, b) \in \text{Tree-wf}$

shows $(\text{abs-Tree}_p\ (p \cdot a), \text{abs-Tree}_p\ b) \in \text{Tree-wf}_p$

using *assms* **by** (*metis (erased, lifting) Tree_p.abs-eq-iff Tree-wf_p.abs-eq hull-relp.intros map-prod-simp rev-image-eqI*)

lemma *Tree-wf_p-trivialI* [*simp*]:

assumes $(a, b) \in \text{Tree-wf}$

shows $(\text{abs-Tree}_p\ a, \text{abs-Tree}_p\ b) \in \text{Tree-wf}_p$

using *assms* by (*metis Tree-wf_pI permute-zero*)

lemma *Tree-wf_pE*:

assumes $(a_p, b_p) \in \text{Tree-wf}_p$

obtains *a b* where $a_p = \text{abs-Tree}_p a$ and $b_p = \text{abs-Tree}_p b$ and $(a, b) \in \text{Tree-wf}$

using *assms* by (*metis Pair-inject Tree-wf_p.abs-eq prod-fun-imageE*)

lemma *wf-Tree-wf_p*: *wf Tree-wf_p*

proof (*rule wf-subset[of inv-image (hull-rel O Tree-wf) rep-Tree_p]*)

show *wf (inv-image (hull-rel O Tree-wf) rep-Tree_p)*

by (*metis Tree-wf-eqvt' wf-Tree-wf wf-hull-rel-relcomp wf-inv-image*)

next

show $\text{Tree-wf}_p \subseteq \text{inv-image (hull-rel O Tree-wf) rep-Tree}_p$

proof (*standard, case-tac x, clarify*)

fix $a_p b_p :: ('d, 'e, 'f) \text{Tree}_p$

assume $(a_p, b_p) \in \text{Tree-wf}_p$

then obtain *a b* where 1: $a_p = \text{abs-Tree}_p a$ and 2: $b_p = \text{abs-Tree}_p b$ and 3:
 $(a, b) \in \text{Tree-wf}$

by (*rule Tree-wf_pE*)

from 1 obtain *p* where 4: $\text{rep-Tree}_p a_p = p \cdot a$

by (*metis permute-rep-abs-Tree_p*)

from 2 obtain *q* where 5: $\text{rep-Tree}_p b_p = q \cdot b$

by (*metis permute-rep-abs-Tree_p*)

have $(p \cdot a, q \cdot a) \in \text{hull-rel}$

by (*metis hull-rel.simps permute-minus-cancel(2) permute-plus*)

moreover from 3 have $(q \cdot a, q \cdot b) \in \text{Tree-wf}$

by (*rule Tree-wf-eqvt-aux*)

ultimately show $(a_p, b_p) \in \text{inv-image (hull-rel O Tree-wf) rep-Tree}_p$

using 4 5 by *auto*

qed

qed

fun *alpha-Tree-termination* :: $('a, 'b, 'c) \text{Tree} \times ('a, 'b, 'c) \text{Tree} \Rightarrow ('a, 'b::\text{pt}, 'c::\text{bn}) \text{Tree}_p \text{ set}$ where

alpha-Tree-termination $(t1, t2) = \{\text{abs-Tree}_p t1, \text{abs-Tree}_p t2\}$

Here it comes ...

function (*sequential*)

alpha-Tree :: $('idx, 'pred::\text{pt}, 'act::\text{bn}) \text{Tree} \Rightarrow ('idx, 'pred, 'act) \text{Tree} \Rightarrow \text{bool}$ (**infix**
 $=_\alpha$ 50) where

— $(=_\alpha)$

alpha-tConj: $tConj \text{ tset1} =_\alpha tConj \text{ tset2} \iff \text{rel-bset } \text{alpha-Tree } \text{tset1 } \text{tset2}$

| *alpha-tNot*: $tNot t1 =_\alpha tNot t2 \iff t1 =_\alpha t2$

| *alpha-tPred*: $tPred \varphi1 =_\alpha tPred \varphi2 \iff \varphi1 = \varphi2$

— the action may have binding names

| *alpha-tAct*: $tAct \alpha1 t1 =_\alpha tAct \alpha2 t2 \iff$

$(\exists p. (\text{bn } \alpha1, t1) \approx_{\text{set}} \text{alpha-Tree (supp-rel alpha-Tree) } p (\text{bn } \alpha2, t2) \wedge (\text{bn } \alpha1, \alpha1) \approx_{\text{set}} ((=) \text{supp } p (\text{bn } \alpha2, \alpha2)))$

| *alpha-other*: $- =_\alpha - \iff \text{False}$

— 254 subgoals (!)

by *pat-completeness auto*

termination

proof

let $?R = \text{inv-image } (\text{max-ext Tree-wf}_p) \text{ alpha-Tree-termination}$

show $\text{wf } ?R$

by (*metis max-ext-wf wf-Tree-wf_p wf-inv-image*)

qed (*auto simp add: max-ext.simps Tree-wf.simps simp del: permute-Tree-tConj*)

We provide more descriptive case names for the automatically generated induction principle.

lemmas *alpha-Tree-induct'* = *alpha-Tree.induct*[*case-names alpha-tConj alpha-tNot alpha-tPred alpha-tAct alpha-other(1) alpha-other(2) alpha-other(3) alpha-other(4) alpha-other(5) alpha-other(6) alpha-other(7) alpha-other(8) alpha-other(9) alpha-other(10) alpha-other(11) alpha-other(12) alpha-other(13) alpha-other(14) alpha-other(15) alpha-other(16) alpha-other(17) alpha-other(18)*]

lemma *alpha-Tree-induct*[*case-names tConj tNot tPred tAct, consumes 1*]:

assumes $t1 =_\alpha t2$

and $\bigwedge tset1 tset2. (\bigwedge a b. a \in \text{set-bset } tset1 \implies b \in \text{set-bset } tset2 \implies a =_\alpha b \implies P a b) \implies$

$\text{rel-bset } (=_\alpha) tset1 tset2 \implies P (tConj tset1) (tConj tset2)$

and $\bigwedge t1 t2. t1 =_\alpha t2 \implies P t1 t2 \implies P (tNot t1) (tNot t2)$

and $\bigwedge \varphi. P (tPred \varphi) (tPred \varphi)$

and $\bigwedge \alpha1 t1 \alpha2 t2. (\bigwedge p. p \cdot t1 =_\alpha t2 \implies P (p \cdot t1) t2) \implies$

$(\bigwedge a b. ((a \rightleftharpoons b) \cdot t1) =_\alpha t1 \implies P ((a \rightleftharpoons b) \cdot t1) t1) \implies (\bigwedge a b. ((a \rightleftharpoons b) \cdot t2) =_\alpha t2 \implies P ((a \rightleftharpoons b) \cdot t2) t2) \implies$

$(\exists p. (bn \alpha1, t1) \approx_{\text{set}} (=_\alpha) (\text{supp-rel } (=_\alpha)) p (bn \alpha2, t2) \wedge (bn \alpha1, \alpha1) \approx_{\text{set}} (=) \text{supp } p (bn \alpha2, \alpha2)) \implies$

$P (tAct \alpha1 t1) (tAct \alpha2 t2)$

shows $P t1 t2$

using *assms by (induction t1 t2 rule: alpha-Tree.induct) simp-all*

α -equivalence is equivariant.

lemma *alpha-Tree-eqt-aux*:

assumes $\bigwedge a b. (a \rightleftharpoons b) \cdot t =_\alpha t \iff p \cdot (a \rightleftharpoons b) \cdot t =_\alpha p \cdot t$

shows $p \cdot \text{supp-rel } (=_\alpha) t = \text{supp-rel } (=_\alpha) (p \cdot t)$

proof –

{

fix a

let $?B = \{b. \neg ((a \rightleftharpoons b) \cdot t) =_\alpha t\}$

let $?pB = \{b. \neg ((p \cdot a \rightleftharpoons b) \cdot p \cdot t) =_\alpha (p \cdot t)\}$

{

assume *finite ?B*

moreover have *inj-on (unpermute p) ?pB*

by (*simp add: inj-on-def unpermute-def*)

moreover have *unpermute p ' ?pB \subseteq ?B*

using *assms by auto (metis (erased, lifting) eqt-bound permute-eqt swap-eqt)*

```

ultimately have finite ?pB
  by (metis inj-on-finite)
}
moreover
{
  assume finite ?pB
  moreover have inj-on (permute p) ?B
    by (simp add: inj-on-def)
  moreover have permute p ` ?B  $\subseteq$  ?pB
    using assms by auto (metis (erased, lifting) permute-eqvt swap-eqvt)
  ultimately have finite ?B
    by (metis inj-on-finite)
}
ultimately have infinite ?B  $\longleftrightarrow$  infinite ?pB
  by auto
}
then show ?thesis
  by (auto simp add: supp-rel-def permute-set-def) (metis eqvt-bound)
qed

lemma alpha-Tree-eqvt': t1 = $\alpha$  t2  $\longleftrightarrow$  p  $\cdot$  t1 = $\alpha$  p  $\cdot$  t2
proof (induction t1 t2 rule: alpha-Tree-induct')
  case (alpha-tConj tset1 tset2) show ?case
  proof
    assume *: tConj tset1 = $\alpha$  tConj tset2
    {
      fix x
      assume x  $\in$  set-bset (p  $\cdot$  tset1)
      then obtain x' where 1: x'  $\in$  set-bset tset1 and 2: x = p  $\cdot$  x'
        by (metis imageE permute-bset.rep-eq permute-set-eq-image)
      from 1 obtain y' where 3: y'  $\in$  set-bset tset2 and 4: x' = $\alpha$  y'
        using * by (metis (mono-tags, lifting) Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
      from 3 have p  $\cdot$  y'  $\in$  set-bset (p  $\cdot$  tset2)
        by (metis mem-permute-iff set-bset-eqvt)
      moreover from 1 and 2 and 3 and 4 have x = $\alpha$  p  $\cdot$  y'
        using alpha-tConj.IH by blast
      ultimately have  $\exists y \in \text{set-bset } (p \cdot \text{tset2}). x =_{\alpha} y ..$ 
    }
  moreover
  {
    fix y
    assume y  $\in$  set-bset (p  $\cdot$  tset2)
    then obtain y' where 1: y'  $\in$  set-bset tset2 and 2: p  $\cdot$  y' = y
      by (metis imageE permute-bset.rep-eq permute-set-eq-image)
    from 1 obtain x' where 3: x'  $\in$  set-bset tset1 and 4: x' = $\alpha$  y'
      using * by (metis (mono-tags, lifting) Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
    from 3 have p  $\cdot$  x'  $\in$  set-bset (p  $\cdot$  tset1)
  }
}

```

```

    by (metis mem-permute-iff set-bset-eqvt)
  moreover from 1 and 2 and 3 and 4 have  $p \cdot x' =_{\alpha} y$ 
    using alpha-tConj.IH by blast
  ultimately have  $\exists x \in \text{set-bset } (p \cdot \text{tset1}). x =_{\alpha} y ..$ 
}
ultimately show  $p \cdot \text{tConj tset1} =_{\alpha} p \cdot \text{tConj tset2}$ 
  by (simp add: rel-bset-def rel-set-def)
next
assume *:  $p \cdot \text{tConj tset1} =_{\alpha} p \cdot \text{tConj tset2}$ 
{
  fix x
  assume 1:  $x \in \text{set-bset tset1}$ 
  then have  $p \cdot x \in \text{set-bset } (p \cdot \text{tset1})$ 
    by (metis mem-permute-iff set-bset-eqvt)
  then obtain  $y'$  where 2:  $y' \in \text{set-bset } (p \cdot \text{tset2})$  and 3:  $p \cdot x =_{\alpha} y'$ 
    using * by (metis Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq
rel-set-def)
  from 2 obtain  $y$  where 4:  $y \in \text{set-bset tset2}$  and 5:  $y' = p \cdot y$ 
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  from 1 and 3 and 4 and 5 have  $x =_{\alpha} y$ 
    using alpha-tConj.IH by blast
  with 4 have  $\exists y \in \text{set-bset tset2}. x =_{\alpha} y ..$ 
}
moreover
{
  fix y
  assume 1:  $y \in \text{set-bset tset2}$ 
  then have  $p \cdot y \in \text{set-bset } (p \cdot \text{tset2})$ 
    by (metis mem-permute-iff set-bset-eqvt)
  then obtain  $x'$  where 2:  $x' \in \text{set-bset } (p \cdot \text{tset1})$  and 3:  $x' =_{\alpha} p \cdot y$ 
    using * by (metis Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq
rel-set-def)
  from 2 obtain  $x$  where 4:  $x \in \text{set-bset tset1}$  and 5:  $p \cdot x = x'$ 
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  from 1 and 3 and 4 and 5 have  $x =_{\alpha} y$ 
    using alpha-tConj.IH by blast
  with 4 have  $\exists x \in \text{set-bset tset1}. x =_{\alpha} y ..$ 
}
ultimately show  $\text{tConj tset1} =_{\alpha} \text{tConj tset2}$ 
  by (simp add: rel-bset-def rel-set-def)
qed
next
case (alpha-tAct  $\alpha 1 t1 \alpha 2 t2$ )
from alpha-tAct.IH(2) have  $t1: p \cdot \text{supp-rel } (=_{\alpha}) t1 = \text{supp-rel } (=_{\alpha}) (p \cdot t1)$ 
  by (rule alpha-Tree-eqvt-aux)
from alpha-tAct.IH(3) have  $t2: p \cdot \text{supp-rel } (=_{\alpha}) t2 = \text{supp-rel } (=_{\alpha}) (p \cdot t2)$ 
  by (rule alpha-Tree-eqvt-aux)
show ?case
proof

```

assume $tAct\ \alpha1\ t1 =_{\alpha}\ tAct\ \alpha2\ t2$
then obtain q **where** $1: (bn\ \alpha1, t1) \approx_{set}\ (=_{\alpha})\ (supp\text{-}rel\ (=_{\alpha}))\ q\ (bn\ \alpha2, t2)$
and $2: (bn\ \alpha1, \alpha1) \approx_{set}\ (=)\ supp\ q\ (bn\ \alpha2, \alpha2)$
by *auto*
from 1 **and** $t1$ **and** $t2$ **have** $supp\text{-}rel\ (=_{\alpha})\ (p \cdot t1) - bn\ (p \cdot \alpha1) = supp\text{-}rel\ (=_{\alpha})\ (p \cdot t2) - bn\ (p \cdot \alpha2)$
by *(metis Diff-eqvt alpha-set bn-eqvt)*
moreover from 1 **and** $t1$ **have** $(supp\text{-}rel\ (=_{\alpha})\ (p \cdot t1) - bn\ (p \cdot \alpha1)) \#* (p + q - p)$
by *(metis Diff-eqvt alpha-set bn-eqvt fresh-star-permute-iff permute-perm-def)*
moreover from 1 **and** *alpha-tAct.IH(1)* **have** $p \cdot q \cdot t1 =_{\alpha}\ p \cdot t2$
by *(simp add: alpha-set)*
moreover from 2 **have** $p \cdot q \cdot -p \cdot bn\ (p \cdot \alpha1) = bn\ (p \cdot \alpha2)$
by *(simp add: alpha-set bn-eqvt)*
ultimately have $(bn\ (p \cdot \alpha1), p \cdot t1) \approx_{set}\ (=_{\alpha})\ (supp\text{-}rel\ (=_{\alpha}))\ (p + q - p)\ (bn\ (p \cdot \alpha2), p \cdot t2)$
by *(simp add: alpha-set)*
moreover from 2 **have** $(bn\ (p \cdot \alpha1), p \cdot \alpha1) \approx_{set}\ (=)\ supp\ (p + q - p)\ (bn\ (p \cdot \alpha2), p \cdot \alpha2)$
by *(simp add: alpha-set) (metis (mono-tags, lifting) Diff-eqvt bn-eqvt fresh-star-permute-iff permute-minus-cancel(2) permute-perm-def supp-eqvt)*
ultimately show $p \cdot tAct\ \alpha1\ t1 =_{\alpha}\ p \cdot tAct\ \alpha2\ t2$
by *auto*
next
assume $p \cdot tAct\ \alpha1\ t1 =_{\alpha}\ p \cdot tAct\ \alpha2\ t2$
then obtain q **where** $1: (bn\ (p \cdot \alpha1), p \cdot t1) \approx_{set}\ (=_{\alpha})\ (supp\text{-}rel\ (=_{\alpha}))\ q\ (bn\ (p \cdot \alpha2), p \cdot t2)$ **and** $2: (bn\ (p \cdot \alpha1), p \cdot \alpha1) \approx_{set}\ (=)\ supp\ q\ (bn\ (p \cdot \alpha2), p \cdot \alpha2)$
by *auto*
{
from 1 **and** $t1$ **and** $t2$ **have** $supp\text{-}rel\ (=_{\alpha})\ t1 - bn\ \alpha1 = supp\text{-}rel\ (=_{\alpha})\ t2 - bn\ \alpha2$
by *(metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff)*
moreover with 1 **and** $t2$ **have** $(supp\text{-}rel\ (=_{\alpha})\ t1 - bn\ \alpha1) \#* (-p + q + p)$
by *(auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting) DiffI bn-eqvt mem-permute-iff permute-minus-cancel(2))*
moreover from 1 **have** $-p \cdot q \cdot p \cdot t1 =_{\alpha}\ t2$
using *alpha-tAct.IH(1)* **by** *(simp add: alpha-set) (metis (no-types, lifting) permute-eqvt permute-minus-cancel(2))*
moreover from 1 **have** $-p \cdot q \cdot p \cdot bn\ \alpha1 = bn\ \alpha2$
by *(metis alpha-set bn-eqvt permute-minus-cancel(2))*
ultimately have $(bn\ \alpha1, t1) \approx_{set}\ (=_{\alpha})\ (supp\text{-}rel\ (=_{\alpha}))\ (-p + q + p)\ (bn\ \alpha2, t2)$
by *(simp add: alpha-set)*
}
moreover
{
from 2 **have** $supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2$


```

      by (metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff
supp-eqvt)
    moreover with 2 have (supp  $\alpha 1 - \text{bn } \alpha 1$ )  $\#^*$   $(-p + q + p)$ 
      by (auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting)
DiffI bn-eqvt mem-permute-iff permute-minus-cancel(1) supp-eqvt)
    moreover from 2 have  $-p \cdot q \cdot p \cdot \alpha 1 = \alpha 2$ 
      by (simp add: alpha-set)
    moreover have  $-p \cdot q \cdot p \cdot \text{bn } \alpha 1 = \text{bn } \alpha 2$ 
      by (simp add: bn-eqvt calculation(3))
    ultimately have  $(\text{bn } \alpha 1, \alpha 1) \approx_{\text{set}} (=) \text{supp } (-p + q + p) (\text{bn } \alpha 2, \alpha 2)$ 
      by (simp add: alpha-set)
  }
  ultimately show  $t\text{Act } \alpha 1 t1 =_{\alpha} t\text{Act } \alpha 2 t2$ 
    by auto
qed
qed simp-all

```

lemma *alpha-Tree-eqvt* [eqvt]: $t1 =_{\alpha} t2 \implies p \cdot t1 =_{\alpha} p \cdot t2$
by (metis *alpha-Tree-eqvt'*)

$(=_{\alpha})$ is an equivalence relation.

lemma *alpha-Tree-reflp*: *reflp alpha-Tree*

proof (rule *reflpI*)

fix $t :: ('a, 'b, 'c) \text{Tree}$

show $t =_{\alpha} t$

proof (induction t)

case *tConj* **then show** ?case **by** (metis *alpha-tConj rel-bset.rep-eq rel-setI*)

next

case *tNot* **then show** ?case **by** (metis *alpha-tNot*)

next

case *tPred* **show** ?case **by** (metis *alpha-tPred*)

next

case *tAct* **then show** ?case **by** (metis (mono-tags) *alpha-tAct alpha-refl(1)*)

qed

qed

lemma *alpha-Tree-symp*: *symp alpha-Tree*

proof (rule *sympI*)

fix $x y :: ('a, 'b, 'c) \text{Tree}$

assume $x =_{\alpha} y$ **then show** $y =_{\alpha} x$

proof (induction $x y$ rule: *alpha-Tree-induct*)

case *tConj* **then show** ?case

by (simp add: *rel-bset-def rel-set-def*) metis

next

case (*tAct* $\alpha 1 t1 \alpha 2 t2$)

then obtain p **where** $(\text{bn } \alpha 1, t1) \approx_{\text{set}} (=_{\alpha}) (\text{supp-rel } (=_{\alpha})) p (\text{bn } \alpha 2, t2)$
 $\wedge (\text{bn } \alpha 1, \alpha 1) \approx_{\text{set}} (=) \text{supp } p (\text{bn } \alpha 2, \alpha 2)$

by auto

then have $(\text{bn } \alpha 2, t2) \approx_{\text{set}} (=_{\alpha}) (\text{supp-rel } (=_{\alpha})) (-p) (\text{bn } \alpha 1, t1) \wedge (\text{bn } \alpha 2,$

```

 $\alpha 2) \approx_{\text{set}} (=) \text{supp } (-p) (\text{bn } \alpha 1, \alpha 1)$ 
  using tAct.IH by (metis (mono-tags, lifting) alpha-Tree-eqvt alpha-sym(1))
  permute-minus-cancel(2))
  then show ?case
    by auto
  qed simp-all
qed

```

lemma *alpha-Tree-transp: transp alpha-Tree*

proof (*rule transpI*)

fix $x y z :: ('a, 'b, 'c) \text{Tree}$

assume $x =_{\alpha} y$ and $y =_{\alpha} z$

then show $x =_{\alpha} z$

proof (*induction x y arbitrary: z rule: alpha-Tree-induct*)

case (*tConj tset-x tset-y*) show ?case

proof (*cases z*)

fix *tset-z*

assume $z: z = \text{tConj } \textit{tset-z}$

have *rel-bset* ($=_{\alpha}$) *tset-x tset-z*

unfolding *rel-bset.rep-eq rel-set-def Ball-def Bex-def*

proof

show $\forall x'. x' \in \textit{set-bset } \textit{tset-x} \longrightarrow (\exists z'. z' \in \textit{set-bset } \textit{tset-z} \wedge x' =_{\alpha} z')$

proof (*rule allI, rule impI*)

fix x' assume 1: $x' \in \textit{set-bset } \textit{tset-x}$

then obtain y' where 2: $y' \in \textit{set-bset } \textit{tset-y}$ and 3: $x' =_{\alpha} y'$

by (*metis rel-bset.rep-eq rel-set-def tConj.hyps*)

from 2 obtain z' where 4: $z' \in \textit{set-bset } \textit{tset-z}$ and 5: $y' =_{\alpha} z'$

by (*metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1 z*)

from 1 2 3 5 have $x' =_{\alpha} z'$

by (*rule tConj.IH*)

with 4 show $\exists z'. z' \in \textit{set-bset } \textit{tset-z} \wedge x' =_{\alpha} z'$

by auto

qed

next

show $\forall z'. z' \in \textit{set-bset } \textit{tset-z} \longrightarrow (\exists x'. x' \in \textit{set-bset } \textit{tset-x} \wedge x' =_{\alpha} z')$

proof (*rule allI, rule impI*)

fix z' assume 1: $z' \in \textit{set-bset } \textit{tset-z}$

then obtain y' where 2: $y' \in \textit{set-bset } \textit{tset-y}$ and 3: $y' =_{\alpha} z'$

by (*metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1 z*)

from 2 obtain x' where 4: $x' \in \textit{set-bset } \textit{tset-x}$ and 5: $x' =_{\alpha} y'$

by (*metis rel-bset.rep-eq rel-set-def tConj.hyps*)

from 4 2 5 3 have $x' =_{\alpha} z'$

by (*rule tConj.IH*)

with 4 show $\exists x'. x' \in \textit{set-bset } \textit{tset-x} \wedge x' =_{\alpha} z'$

by auto

qed

qed

with z show *tConj tset-x* $=_{\alpha} z$

by *simp*

```

    qed (insert tConj.prem, auto)
next
  case tNot then show ?case
  by (cases z) simp-all
next
  case tPred then show ?case
  by simp
next
  case (tAct  $\alpha 1 t1 \alpha 2 t2$ ) show ?case
  proof (cases z)
    fix  $\alpha t$ 
    assume  $z: z = tAct \alpha t$ 
    obtain  $p$  where  $1: (bn \alpha 1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) p (bn \alpha 2, t2) \wedge$ 
       $(bn \alpha 1, \alpha 1) \approx_{set} (=) supp p (bn \alpha 2, \alpha 2)$ 
    using tAct.hyps by auto
    obtain  $q$  where  $2: (bn \alpha 2, t2) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) q (bn \alpha, t) \wedge (bn$ 
       $\alpha 2, \alpha 2) \approx_{set} (=) supp q (bn \alpha, \alpha)$ 
    using tAct.prem z by auto
    have  $(bn \alpha 1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (q + p) (bn \alpha, t)$ 
    proof -
      have  $supp-rel (=_{\alpha}) t1 - bn \alpha 1 = supp-rel (=_{\alpha}) t - bn \alpha$ 
      using 1 and 2 by (metis alpha-set)
      moreover have  $(supp-rel (=_{\alpha}) t1 - bn \alpha 1) \#* (q + p)$ 
      using 1 and 2 by (metis alpha-set fresh-star-plus)
      moreover have  $(q + p) \cdot t1 =_{\alpha} t$ 
      using 1 and 2 and tAct.IH by (metis (no-types, lifting) alpha-Tree-equivt
        alpha-set permute-minus-cancel(1) permute-plus)
      moreover have  $(q + p) \cdot bn \alpha 1 = bn \alpha$ 
      using 1 and 2 by (metis alpha-set permute-plus)
      ultimately show ?thesis
      by (metis alpha-set)
    qed
    moreover have  $(bn \alpha 1, \alpha 1) \approx_{set} (=) supp (q + p) (bn \alpha, \alpha)$ 
    using 1 and 2 by (metis (mono-tags) alpha-trans(1) permute-plus)
    ultimately show tAct  $\alpha 1 t1 =_{\alpha} z$ 
    using z by auto
  qed (insert tAct.prem, auto)
qed
qed

```

lemma *alpha-Tree-equiv*: *equiv alpha-Tree*

by (auto intro: *equivI alpha-Tree-refl alpha-Tree-symp alpha-Tree-transp*)

alpha-equivalent trees have the same support modulo *alpha*-equivalence.

lemma *alpha-Tree-supp-rel*:

assumes $t1 =_{\alpha} t2$

shows $supp-rel (=_{\alpha}) t1 = supp-rel (=_{\alpha}) t2$

using *assms* **proof** (*induction rule: alpha-Tree-induct*)

case (tConj tset1 tset2)

```

have sym:  $\bigwedge x y. \text{rel-bset } (=_{\alpha}) x y \longleftrightarrow \text{rel-bset } (=_{\alpha}) y x$ 
  by (meson alpha-Tree-symp bset.rel-symp sympE)
{
  fix a b
  from tConj.hyps have *:  $\text{rel-bset } (=_{\alpha}) ((a \equiv b) \cdot \text{tset1}) ((a \equiv b) \cdot \text{tset2})$ 
    by (metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj)
  have  $\text{rel-bset } (=_{\alpha}) ((a \equiv b) \cdot \text{tset1}) \text{tset1} \longleftrightarrow \text{rel-bset } (=_{\alpha}) ((a \equiv b) \cdot \text{tset2})$ 
    by (rule iffI) (metis * alpha-Tree-transp bset.rel-transp sym tConj.hyps
  tset2
  transpE)+
}
then show ?case
  by (simp add: supp-rel-def)
next
case tNot then show ?case
  by (simp add: supp-rel-def)
next
case (tAct  $\alpha$ 1 t1  $\alpha$ 2 t2)
{
  fix a b
  have  $\text{tAct } \alpha$ 1 t1  $=_{\alpha}$   $\text{tAct } \alpha$ 2 t2
    using tAct.hyps by simp
  then have  $(a \equiv b) \cdot \text{tAct } \alpha$ 1 t1  $=_{\alpha}$   $\text{tAct } \alpha$ 1 t1  $\longleftrightarrow (a \equiv b) \cdot \text{tAct } \alpha$ 2 t2  $=_{\alpha}$ 
    tAct  $\alpha$ 2 t2
    by (metis (no-types, lifting) alpha-Tree-eqvt alpha-Tree-symp alpha-Tree-transp
  sympE transpE)
}
then show ?case
  by (simp add: supp-rel-def)
qed simp-all

```

$tAct$ preserves α -equivalence.

lemma alpha-Tree-tAct:

assumes $t1 =_{\alpha} t2$

shows $\text{tAct } \alpha t1 =_{\alpha} \text{tAct } \alpha t2$

proof –

have $(\text{bn } \alpha, t1) \approx_{\text{set}} (=_{\alpha}) (\text{supp-rel } (=_{\alpha})) 0 (\text{bn } \alpha, t2)$

using *assms* **by** (simp add: alpha-Tree-supp-rel alpha-set fresh-star-zero)

moreover have $(\text{bn } \alpha, \alpha) \approx_{\text{set}} (=) \text{supp } 0 (\text{bn } \alpha, \alpha)$

by (metis (full-types) alpha-refl(1))

ultimately show ?thesis

by auto

qed

The following lemmas describe the support modulo α -equivalence.

lemma supp-rel-tNot [simp]: $\text{supp-rel } (=_{\alpha}) (\text{tNot } t) = \text{supp-rel } (=_{\alpha}) t$

unfolding supp-rel-def **by** simp

lemma supp-rel-tPred [simp]: $\text{supp-rel } (=_{\alpha}) (\text{tPred } \varphi) = \text{supp } \varphi$

unfolding *supp-rel-def supp-def* **by** *simp*

The support modulo α -equivalence of $tAct\ \alpha\ t$ is not easily described: when t has infinite support (modulo α -equivalence), the support (modulo α -equivalence) of $tAct\ \alpha\ t$ may still contain names in $bn\ \alpha$. This incongruity is avoided when t has finite support modulo α -equivalence.

lemma *infinite-mono*: $infinite\ S \implies (\bigwedge x. x \in S \implies x \in T) \implies infinite\ T$
by (*metis infinite-super subsetI*)

lemma *supp-rel-tAct* [*simp*]:

assumes *finite* (*supp-rel* $(=_{\alpha})\ t$)

shows *supp-rel* $(=_{\alpha})\ (tAct\ \alpha\ t) = supp\ \alpha \cup supp\text{-rel}\ (=_{\alpha})\ t - bn\ \alpha$

proof

show $supp\ \alpha \cup supp\text{-rel}\ (=_{\alpha})\ t - bn\ \alpha \subseteq supp\text{-rel}\ (=_{\alpha})\ (tAct\ \alpha\ t)$

proof

fix x

assume $x \in supp\ \alpha \cup supp\text{-rel}\ (=_{\alpha})\ t - bn\ \alpha$

moreover

{

assume $x1: x \in supp\ \alpha$ **and** $x2: x \notin bn\ \alpha$

from $x1$ **have** *infinite* $\{b. (x \equiv b) \cdot \alpha \neq \alpha\}$

unfolding *supp-def* **..**

then have *infinite* $(\{b. (x \equiv b) \cdot \alpha \neq \alpha\} - supp\ \alpha)$

by (*simp add: finite-supp*)

moreover

{

fix b

assume $b \in \{b. (x \equiv b) \cdot \alpha \neq \alpha\} - supp\ \alpha$

then have $b1: (x \equiv b) \cdot \alpha \neq \alpha$ **and** $b2: b \notin supp\ \alpha - bn\ \alpha$

by *simp+*

from $b1$ **have** *sort-of* $x = sort\text{-of}\ b$

using *swap-different-sorts* **by** *fastforce*

then have $(x \equiv b) \cdot (supp\ \alpha - bn\ \alpha) \neq supp\ \alpha - bn\ \alpha$

using $b2\ x1\ x2$ **by** (*simp add: swap-set-in*)

then have $b \in \{b. \neg (x \equiv b) \cdot tAct\ \alpha\ t =_{\alpha} tAct\ \alpha\ t\}$

by (*auto simp add: alpha-set Diff-eqvt bn-eqvt*)

}

ultimately have *infinite* $\{b. \neg (x \equiv b) \cdot tAct\ \alpha\ t =_{\alpha} tAct\ \alpha\ t\}$

by (*rule infinite-mono*)

then have $x \in supp\text{-rel}\ (=_{\alpha})\ (tAct\ \alpha\ t)$

unfolding *supp-rel-def* **..**

}

moreover

{

assume $x1: x \in supp\text{-rel}\ (=_{\alpha})\ t$ **and** $x2: x \notin bn\ \alpha$

from $x1$ **have** *infinite* $\{b. \neg (x \equiv b) \cdot t =_{\alpha} t\}$

unfolding *supp-rel-def* **..**

then have *infinite* $(\{b. \neg (x \equiv b) \cdot t =_{\alpha} t\} - supp\text{-rel}\ (=_{\alpha})\ t)$

using *assms* **by** *simp*

}

```

moreover
{
  fix  $b$ 
  assume  $b \in \{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\} - \text{supp-rel } (=_{\alpha}) t$ 
  then have  $b1: \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t$  and  $b2: b \notin \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha$ 
  by simp+
  from  $b1$  have  $(x \rightleftharpoons b) \cdot t \neq t$ 
  by (metis alpha-Tree-reflp reflpE)
  then have sort-of  $x = \text{sort-of } b$ 
  using swap-different-sorts by fastforce
  then have  $(x \rightleftharpoons b) \cdot (\text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha) \neq \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha$ 
  using  $b2$   $x1$   $x2$  by (simp add: swap-set-in)
  then have  $\text{supp-rel } (=_{\alpha}) ((x \rightleftharpoons b) \cdot t) - \text{bn } ((x \rightleftharpoons b) \cdot \alpha) \neq \text{supp-rel } (=_{\alpha})$ 
 $t - \text{bn } \alpha$ 
  by (simp add: Diff-egvt bn-egvt)
  then have  $b \in \{b. \neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}$ 
  by (simp add: alpha-set)
}
ultimately have infinite  $\{b. \neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}$ 
by (rule infinite-mono)
then have  $x \in \text{supp-rel } (=_{\alpha}) (tAct \alpha t)$ 
unfolding supp-rel-def ..
}
ultimately show  $x \in \text{supp-rel } (=_{\alpha}) (tAct \alpha t)$ 
by auto
qed
next
show  $\text{supp-rel } (=_{\alpha}) (tAct \alpha t) \subseteq \text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha$ 
proof
fix  $x$ 
assume  $x \in \text{supp-rel } (=_{\alpha}) (tAct \alpha t)$ 
then have  $*$ : infinite  $\{b. \neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}$ 
unfolding supp-rel-def ..
moreover
{
  fix  $b$ 
  assume  $\neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t$ 
  then have  $(x \rightleftharpoons b) \cdot \alpha \neq \alpha \vee \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t$ 
  using alpha-Tree-tAct by force
}
ultimately have infinite  $\{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha \vee \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
by (metis (mono-tags, lifting) infinite-mono mem-Collect-eq)
then have infinite  $\{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha\} \vee \text{infinite } \{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
by (metis (mono-tags) finite-Collect-disjI)
then have  $x \in \text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t$ 
by (simp add: supp-def supp-rel-def)
moreover
{
  assume  $*$ :  $x \in \text{bn } \alpha$ 

```

```

from * obtain b where b1:  $\neg (x \equiv b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t$  and b2:  $b \notin supp \alpha$  and b3:  $b \notin supp-rel (=_{\alpha}) t$ 
  using assms by (metis (no-types, lifting) UnCI finite-UnI finite-supp
infinite-mono mem-Collect-eq)
  let ?p =  $(x \equiv b)$ 
  have  $supp-rel (=_{\alpha}) ((x \equiv b) \cdot t) - bn ((x \equiv b) \cdot \alpha) = supp-rel (=_{\alpha}) t - bn \alpha$ 
  using ** and b3 by (metis (no-types, lifting) Diff-eqvt Diff-iff alpha-Tree-eqvt'
alpha-Tree-eqvt-aux bn-eqvt swap-set-not-in)
  moreover then have  $(supp-rel (=_{\alpha}) ((x \equiv b) \cdot t) - bn ((x \equiv b) \cdot \alpha)) \#* ?p$ 
  using ** and b3 by (metis Diff-iff fresh-perm fresh-star-def swap-atom-simps(3))
  moreover have  $?p \cdot (x \equiv b) \cdot t =_{\alpha} t$ 
    using alpha-Tree-reflp reflpE by force
  moreover have  $?p \cdot bn ((x \equiv b) \cdot \alpha) = bn \alpha$ 
    by (simp add: bn-eqvt)
  moreover have  $supp ((x \equiv b) \cdot \alpha) - bn ((x \equiv b) \cdot \alpha) = supp \alpha - bn \alpha$ 
    using ** and b2 by (metis (mono-tags, hide-lams) Diff-eqvt Diff-iff bn-eqvt
supp-eqvt swap-set-not-in)
  moreover then have  $(supp ((x \equiv b) \cdot \alpha) - bn ((x \equiv b) \cdot \alpha)) \#* ?p$ 
    using ** and b2 by (simp add: fresh-star-def fresh-def supp-perm) (metis
Diff-iff swap-atom-simps(3))
  moreover have  $?p \cdot (x \equiv b) \cdot \alpha = \alpha$ 
    by simp
  ultimately have  $(x \equiv b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t$ 
    by (auto simp add: alpha-set)
  with b1 have False ..
}
ultimately show  $x \in supp \alpha \cup supp-rel (=_{\alpha}) t - bn \alpha$ 
by blast
qed
qed

```

We define the type of (infinitely branching) trees quotiented by α -equivalence.

quotient-type

```

('idx,'pred,'act) Tree $\alpha$  = ('idx,'pred::pt,'act::bn) Tree / alpha-Tree
by (fact alpha-Tree-equivp)

```

lemma *Tree* _{α} -*abs-rep* [*simp*]: $abs-Tree_{\alpha} (rep-Tree_{\alpha} t_{\alpha}) = t_{\alpha}$
by (*metis* *Quotient-Tree* _{α} *Quotient-abs-rep*)

lemma *Tree* _{α} -*rep-abs* [*simp*]: $rep-Tree_{\alpha} (abs-Tree_{\alpha} t) =_{\alpha} t$
by (*metis* *Tree* _{α} .*abs-eq-iff* *Tree* _{α} -*abs-rep*)

The permutation operation is lifted from trees.

instantiation *Tree* _{α} :: (*type*, *pt*, *bn*) *pt*
begin

```

lift-definition permute-Tree $\alpha$  :: perm  $\Rightarrow$  ('a,'b,'c) Tree $\alpha$   $\Rightarrow$  ('a,'b,'c) Tree $\alpha$ 

```

is permute
by (*fact alpha-Tree-eqvt*)

instance

proof

fix $t_\alpha :: (-,-,-) \text{Tree}_\alpha$

show $0 \cdot t_\alpha = t_\alpha$

by *transfer (metis alpha-Tree-equivp equivp-reflp permute-zero)*

next

fix $p \ q :: \text{perm}$ **and** $t_\alpha :: (-,-,-) \text{Tree}_\alpha$

show $(p + q) \cdot t_\alpha = p \cdot q \cdot t_\alpha$

by *transfer (metis alpha-Tree-equivp equivp-reflp permute-plus)*

qed

end

The abstraction function from trees to trees modulo α -equivalence is equivariant. The representation function is equivariant modulo α -equivalence.

lemmas *permute-Tree $_\alpha$.abs-eq [eqvt, simp]*

lemma *alpha-Tree-permute-rep-commute [simp]: $p \cdot \text{rep-Tree}_\alpha \ t_\alpha =_\alpha \text{rep-Tree}_\alpha \ (p \cdot t_\alpha)$*

by (*metis Tree $_\alpha$.abs-eq-iff Tree $_\alpha$ -abs-rep permute-Tree $_\alpha$.abs-eq*)

5.3 Constructors for trees modulo α -equivalence

The constructors are lifted from trees.

lift-definition $\text{Conj}_\alpha :: ('idx, 'pred, 'act) \text{Tree}_\alpha \ \text{set}['idx] \Rightarrow ('idx, 'pred::pt, 'act::bn)$

Tree_α **is**

tConj

by *simp*

lemma *map-bset-abs-rep-Tree $_\alpha$: $\text{map-bset} \ \text{abs-Tree}_\alpha \ (\text{map-bset} \ \text{rep-Tree}_\alpha \ \text{tset}_\alpha) = \text{tset}_\alpha$*

by (*metis (full-types) Quotient-Tree $_\alpha$ Quotient-abs-rep bset-lifting.bset-quot-map*)

lemma *Conj $_\alpha$ -def': $\text{Conj}_\alpha \ \text{tset}_\alpha = \text{abs-Tree}_\alpha \ (\text{tConj} \ (\text{map-bset} \ \text{rep-Tree}_\alpha \ \text{tset}_\alpha))$*

by (*metis Conj $_\alpha$.abs-eq map-bset-abs-rep-Tree $_\alpha$*)

lift-definition $\text{Not}_\alpha :: ('idx, 'pred, 'act) \text{Tree}_\alpha \Rightarrow ('idx, 'pred::pt, 'act::bn) \text{Tree}_\alpha$ **is**

tNot

by *simp*

lift-definition $\text{Pred}_\alpha :: 'pred \Rightarrow ('idx, 'pred::pt, 'act::bn) \text{Tree}_\alpha$ **is**

tPred

.

lift-definition $\text{Act}_\alpha :: 'act \Rightarrow ('idx, 'pred, 'act) \text{Tree}_\alpha \Rightarrow ('idx, 'pred::pt, 'act::bn)$

Tree_α **is**

tAct
by (*fact alpha-Tree-tAct*)

The lifted constructors are equivariant.

lemma *Conj_α-eqvt* [*eqvt*, *simp*]: $p \cdot \text{Conj}_\alpha \text{ tset}_\alpha = \text{Conj}_\alpha (p \cdot \text{tset}_\alpha)$

proof –

```

{
  fix x
  assume x ∈ set-bset (p · map-bset rep-Treeα tsetα)
  then obtain y where y ∈ set-bset (map-bset rep-Treeα tsetα) and x = p · y
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  then obtain tα where 1: tα ∈ set-bset tsetα and 2: x = p · rep-Treeα tα
    by (metis imageE map-bset.rep-eq)
  let ?x' = rep-Treeα (p · tα)
  from 1 have p · tα ∈ set-bset (p · tsetα)
    by (metis mem-permute-iff permute-bset.rep-eq)
  then have ?x' ∈ set-bset (map-bset rep-Treeα (p · tsetα))
    by (simp add: bset.set-map)
  moreover from 2 have x =α ?x'
    by (metis alpha-Tree-permute-rep-commute)
  ultimately have ∃ x' ∈ set-bset (map-bset rep-Treeα (p · tsetα)). x =α x'
    ..
}
moreover
{
  fix y
  assume y ∈ set-bset (map-bset rep-Treeα (p · tsetα))
  then obtain x where x ∈ set-bset (p · tsetα) and rep-Treeα x = y
    by (metis imageE map-bset.rep-eq)
  then obtain tα where 1: tα ∈ set-bset tsetα and 2: rep-Treeα (p · tα) = y
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  let ?y' = p · rep-Treeα tα
  from 1 have rep-Treeα tα ∈ set-bset (map-bset rep-Treeα tsetα)
    by (simp add: bset.set-map)
  then have ?y' ∈ set-bset (p · map-bset rep-Treeα tsetα)
    by (metis mem-permute-iff permute-bset.rep-eq)
  moreover from 2 have ?y' =α y
    by (metis alpha-Tree-permute-rep-commute)
  ultimately have ∃ y' ∈ set-bset (p · map-bset rep-Treeα tsetα). y' =α y
    ..
}
ultimately show ?thesis
  by (simp add: Conjα-def' map-bset-eqvt rel-bset-def rel-set-def Treeα.abs-eq-iff)
qed

```

lemma *Not_α-eqvt* [*eqvt*, *simp*]: $p \cdot \text{Not}_\alpha t_\alpha = \text{Not}_\alpha (p \cdot t_\alpha)$

by (*induct t_α*) (*simp add: Not_α.abs-eq*)

lemma *Pred_α-eqvt* [*eqvt*, *simp*]: $p \cdot \text{Pred}_\alpha \varphi = \text{Pred}_\alpha (p \cdot \varphi)$

by (*simp add: Pred $_{\alpha}$.abs-eq*)

lemma *Act $_{\alpha}$ -eqvt* [*eqvt, simp*]: $p \cdot \text{Act}_{\alpha} \alpha t_{\alpha} = \text{Act}_{\alpha} (p \cdot \alpha) (p \cdot t_{\alpha})$
 by (*induct t $_{\alpha}$*) (*simp add: Act $_{\alpha}$.abs-eq*)

The lifted constructors are injective (except for *Act $_{\alpha}$*).

lemma *Conj $_{\alpha}$ -eq-iff* [*simp*]: $\text{Conj}_{\alpha} \text{tset1}_{\alpha} = \text{Conj}_{\alpha} \text{tset2}_{\alpha} \longleftrightarrow \text{tset1}_{\alpha} = \text{tset2}_{\alpha}$
proof
 assume *Conj $_{\alpha}$ tset1 $_{\alpha}$ = Conj $_{\alpha}$ tset2 $_{\alpha}$*
 then have *tConj (map-bset rep-Tree $_{\alpha}$ tset1 $_{\alpha}$) = $_{\alpha}$ tConj (map-bset rep-Tree $_{\alpha}$ tset2 $_{\alpha}$)*
 by (*metis Conj $_{\alpha}$ -def' Tree $_{\alpha}$.abs-eq-iff*)
 then have *rel-bset (= $_{\alpha}$) (map-bset rep-Tree $_{\alpha}$ tset1 $_{\alpha}$) (map-bset rep-Tree $_{\alpha}$ tset2 $_{\alpha}$)*
 by (*auto elim: alpha-Tree.cases*)
 then show *tset1 $_{\alpha}$ = tset2 $_{\alpha}$*
 using *Quotient-Tree $_{\alpha}$ Quotient-rel-abs2 bset-lifting.bset-quot-map map-bset-abs-rep-Tree $_{\alpha}$*
 by *fastforce*
qed (*fact arg-cong*)

lemma *Not $_{\alpha}$ -eq-iff* [*simp*]: $\text{Not}_{\alpha} t1_{\alpha} = \text{Not}_{\alpha} t2_{\alpha} \longleftrightarrow t1_{\alpha} = t2_{\alpha}$
proof
 assume *Not $_{\alpha}$ t1 $_{\alpha}$ = Not $_{\alpha}$ t2 $_{\alpha}$*
 then have *tNot (rep-Tree $_{\alpha}$ t1 $_{\alpha}$) = $_{\alpha}$ tNot (rep-Tree $_{\alpha}$ t2 $_{\alpha}$)*
 by (*metis Not $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$ -abs-rep*)
 then have *rep-Tree $_{\alpha}$ t1 $_{\alpha}$ = $_{\alpha}$ rep-Tree $_{\alpha}$ t2 $_{\alpha}$*
 using *alpha-Tree.cases* by *auto*
 then show *t1 $_{\alpha}$ = t2 $_{\alpha}$*
 by (*metis Tree $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$ -abs-rep*)
next
 assume *t1 $_{\alpha}$ = t2 $_{\alpha}$* then show *Not $_{\alpha}$ t1 $_{\alpha}$ = Not $_{\alpha}$ t2 $_{\alpha}$*
 by *simp*
qed

lemma *Pred $_{\alpha}$ -eq-iff* [*simp*]: $\text{Pred}_{\alpha} \varphi1 = \text{Pred}_{\alpha} \varphi2 \longleftrightarrow \varphi1 = \varphi2$
proof
 assume *Pred $_{\alpha}$ $\varphi1$ = Pred $_{\alpha}$ $\varphi2$*
 then have *(tPred $\varphi1$:: ('d, 'b, 'e) Tree) = $_{\alpha}$ tPred $\varphi2$* — note the unrelated type
 by (*metis Pred $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-eq-iff*)
 then show *$\varphi1$ = $\varphi2$*
 using *alpha-Tree.cases* by *auto*
next
 assume *$\varphi1$ = $\varphi2$* then show *Pred $_{\alpha}$ $\varphi1$ = Pred $_{\alpha}$ $\varphi2$*
 by *simp*
qed

lemma *Act $_{\alpha}$ -eq-iff*: $\text{Act}_{\alpha} \alpha1 t1 = \text{Act}_{\alpha} \alpha2 t2 \longleftrightarrow \text{tAct } \alpha1 (\text{rep-Tree}_{\alpha} t1) =_{\alpha} \text{tAct } \alpha2 (\text{rep-Tree}_{\alpha} t2)$
 by (*metis Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$ -abs-rep*)

The lifted constructors are free (except for Act_α).

lemma *Tree $_\alpha$ -free* [simp]:
shows $Conj_\alpha \text{ tset}_\alpha \neq Not_\alpha t_\alpha$
and $Conj_\alpha \text{ tset}_\alpha \neq Pred_\alpha \varphi$
and $Conj_\alpha \text{ tset}_\alpha \neq Act_\alpha \alpha t_\alpha$
and $Not_\alpha t_\alpha \neq Pred_\alpha \varphi$
and $Not_\alpha t1_\alpha \neq Act_\alpha \alpha t2_\alpha$
and $Pred_\alpha \varphi \neq Act_\alpha \alpha t_\alpha$
by (simp add: Conj $_\alpha$ -def' Not $_\alpha$ -def Pred $_\alpha$ -def Act $_\alpha$ -def Tree $_\alpha$.abs-eq-iff)+

The following lemmas describe the support of constructed trees modulo α -equivalence.

lemma *supp-alpha-supp-rel*: $supp \ t_\alpha = supp\text{-rel} \ (=_\alpha) \ (rep\text{-Tree}_\alpha \ t_\alpha)$
unfolding *supp-def supp-rel-def* **by** (metis (mono-tags, lifting) Collect-cong Tree $_\alpha$.abs-eq-iff Tree $_\alpha$ -abs-rep alpha-Tree-permute-rep-commute)

lemma *supp-Conj $_\alpha$* [simp]: $supp \ (Conj_\alpha \ \text{tset}_\alpha) = supp \ \text{tset}_\alpha$
unfolding *supp-def* **by** *simp*

lemma *supp-Not $_\alpha$* [simp]: $supp \ (Not_\alpha \ t_\alpha) = supp \ t_\alpha$
unfolding *supp-def* **by** *simp*

lemma *supp-Pred $_\alpha$* [simp]: $supp \ (Pred_\alpha \ \varphi) = supp \ \varphi$
unfolding *supp-def* **by** *simp*

lemma *supp-Act $_\alpha$* [simp]:
assumes *finite* (supp t_α)
shows $supp \ (Act_\alpha \ \alpha \ t_\alpha) = supp \ \alpha \cup supp \ t_\alpha - bn \ \alpha$
using *assms* **by** (metis Act $_\alpha$.abs-eq Tree $_\alpha$ -abs-rep Tree $_\alpha$ -rep-abs alpha-Tree-supp-rel supp-alpha-supp-rel supp-rel-tAct)

5.4 Induction over trees modulo α -equivalence

lemma *Tree $_\alpha$ -induct* [case-names Conj $_\alpha$ Not $_\alpha$ Pred $_\alpha$ Act $_\alpha$ Env $_\alpha$, induct type: Tree $_\alpha$]:

fixes t_α
assumes $\bigwedge \text{tset}_\alpha. (\bigwedge x. x \in \text{set-bset} \ \text{tset}_\alpha \implies P \ x) \implies P \ (Conj_\alpha \ \text{tset}_\alpha)$
and $\bigwedge t_\alpha. P \ t_\alpha \implies P \ (Not_\alpha \ t_\alpha)$
and $\bigwedge \text{pred}. P \ (Pred_\alpha \ \text{pred})$
and $\bigwedge \text{act} \ t_\alpha. P \ t_\alpha \implies P \ (Act_\alpha \ \text{act} \ t_\alpha)$
shows $P \ t_\alpha$
proof (rule Tree $_\alpha$.abs-induct)
fix t **show** $P \ (abs\text{-Tree}_\alpha \ t)$
proof (induction t)
case (tConj tset)
let $?tset_\alpha = \text{map-bset} \ abs\text{-Tree}_\alpha \ \text{tset}$
have $abs\text{-Tree}_\alpha \ (tConj \ \text{tset}) = Conj_\alpha \ ?tset_\alpha$
by (simp add: Conj $_\alpha$.abs-eq)
then show ?case

```

      using assms(1) tConj.IH by (metis imageE map-bset.rep-eq)
next
  case tNot then show ?case
    using assms(2) by (metis Notα.abs-eq)
next
  case tPred show ?case
    using assms(3) by (metis Predα.abs-eq)
next
  case tAct then show ?case
    using assms(4) by (metis Actα.abs-eq)
qed
qed

```

There is no (obvious) strong induction principle for trees modulo α -equivalence: since their support may be infinite, we may not be able to rename bound variables without also renaming free variables.

5.5 Hereditarily finitely supported trees

We cannot obtain the type of infinitary formulas simply as the sub-type of all trees (modulo α -equivalence) that are finitely supported: since an infinite set of trees may be finitely supported even though its members are not (and thus, would not be formulas), the sub-type of *all* finitely supported trees does not validate the induction principle that we desire for formulas.

Instead, we define *hereditarily* finitely supported trees. We require that environments and state predicates are finitely supported.

```

inductive hereditarily-fs :: ('idx, 'pred::fs, 'act::bn) Treeα  $\Rightarrow$  bool where
  Conjα: finite (supp tsetα)  $\Longrightarrow$  ( $\bigwedge t_{\alpha}. t_{\alpha} \in \text{set-bset } tset_{\alpha} \Longrightarrow \text{hereditarily-fs } t_{\alpha}$ )
 $\Longrightarrow$  hereditarily-fs (Conjα tsetα)
| Notα: hereditarily-fs tα  $\Longrightarrow$  hereditarily-fs (Notα tα)
| Predα: hereditarily-fs (Predα  $\varphi$ )
| Actα: hereditarily-fs tα  $\Longrightarrow$  hereditarily-fs (Actα  $\alpha$  tα)

```

hereditarily-fs is equivariant.

lemma *hereditarily-fs-eqvt* [*eqvt*]:

```

  assumes hereditarily-fs tα
  shows hereditarily-fs (p  $\cdot$  tα)
using assms proof (induction rule: hereditarily-fs.induct)
  case Conjα then show ?case
    by (metis (erased, hide-lams) Conjα-eqvt hereditarily-fs.Conjα mem-permute-iff
permute-finite permute-minus-cancel(1) set-bset-eqvt supp-eqvt)
next
  case Notα then show ?case
    by (metis Notα-eqvt hereditarily-fs.Notα)
next
  case Predα then show ?case
    by (metis Predα-eqvt hereditarily-fs.Predα)

```

```

next
  case  $Act_\alpha$  then show ?case
  by (metis  $Act_\alpha$ -eqvt hereditarily-fs. $Act_\alpha$ )
qed

```

hereditarily-fs is preserved under α -renaming.

lemma *hereditarily-fs-alpha-renaming*:

```

  assumes  $Act_\alpha \alpha t_\alpha = Act_\alpha \alpha' t_\alpha'$ 
  shows  $hereditarily-fs t_\alpha \longleftrightarrow hereditarily-fs t_\alpha'$ 

```

proof

```

  assume  $hereditarily-fs t_\alpha$ 
  then show  $hereditarily-fs t_\alpha'$ 
    using assms by (auto simp add:  $Act_\alpha$ -def  $Tree_\alpha$ .abs-eq-iff alphas) (metis
 $Tree_\alpha$ .abs-eq-iff  $Tree_\alpha$ -abs-rep hereditarily-fs-eqvt permute- $Tree_\alpha$ .abs-eq)

```

next

```

  assume  $hereditarily-fs t_\alpha'$ 
  then show  $hereditarily-fs t_\alpha$ 
    using assms by (auto simp add:  $Act_\alpha$ -def  $Tree_\alpha$ .abs-eq-iff alphas) (metis
 $Tree_\alpha$ .abs-eq-iff  $Tree_\alpha$ -abs-rep hereditarily-fs-eqvt permute- $Tree_\alpha$ .abs-eq permute-minus-cancel(2))

```

qed

Hereditarily finitely supported trees have finite support.

lemma *hereditarily-fs-implies-finite-supp*:

```

  assumes  $hereditarily-fs t_\alpha$ 
  shows  $finite (supp t_\alpha)$ 

```

using *assms* **by** (*induction rule: hereditarily-fs.induct*) (*simp-all add: finite-supp*)

5.6 Infinitary formulas

Now, infinitary formulas are simply the sub-type of hereditarily finitely supported trees.

```

typedef ('idx,'pred::fs,'act::bn) formula = { $t_\alpha::('idx,'pred,'act) Tree_\alpha$ .  $hereditarily-fs t_\alpha$ }

```

by (*metis hereditarily-fs.Pred $_\alpha$ mem-Collect-eq*)

We set up Isabelle's lifting infrastructure so that we can lift definitions from the type of trees modulo α -equivalence to the sub-type of formulas.

setup-lifting *type-definition-formula*

lemma *Abs-formula-inverse* [*simp*]:

```

  assumes  $hereditarily-fs t_\alpha$ 
  shows  $Rep-formula (Abs-formula t_\alpha) = t_\alpha$ 

```

using *assms* **by** (*metis Abs-formula-inverse mem-Collect-eq*)

lemma *Rep-formula'* [*simp*]: *hereditarily-fs* (*Rep-formula* *x*)

by (*metis Rep-formula mem-Collect-eq*)

Now we lift the permutation operation.

```

instantiation formula :: (type, fs, bn) pt
begin

  lift-definition permute-formula :: perm  $\Rightarrow$  ('a,'b,'c) formula  $\Rightarrow$  ('a,'b,'c) formula
    is permute
  by (fact hereditarily-fs-eqvt)

  instance
  by standard (transfer, simp)+

end

```

The abstraction and representation functions for formulas are equivariant, and they preserve support.

```

lemma Abs-formula-eqvt [simp]:
  assumes hereditarily-fs t $_{\alpha}$ 
  shows p  $\cdot$  Abs-formula t $_{\alpha}$  = Abs-formula (p  $\cdot$  t $_{\alpha}$ )
by (metis assms eq-onp-same-args permute-formula.abs-eq)

```

```

lemma supp-Abs-formula [simp]:
  assumes hereditarily-fs t $_{\alpha}$ 
  shows supp (Abs-formula t $_{\alpha}$ ) = supp t $_{\alpha}$ 
proof –
  {
    fix p :: perm
    have p  $\cdot$  Abs-formula t $_{\alpha}$  = Abs-formula (p  $\cdot$  t $_{\alpha}$ )
      using assms by (metis Abs-formula-eqvt)
    moreover have hereditarily-fs (p  $\cdot$  t $_{\alpha}$ )
      using assms by (metis hereditarily-fs-eqvt)
    ultimately have p  $\cdot$  Abs-formula t $_{\alpha}$  = Abs-formula t $_{\alpha}$   $\longleftrightarrow$  p  $\cdot$  t $_{\alpha}$  = t $_{\alpha}$ 
      using assms by (metis Abs-formula-inverse)
  }
  then show ?thesis unfolding supp-def by simp
qed

```

```

lemmas Rep-formula-eqvt [eqvt, simp] = permute-formula.rep-eq[symmetric]

```

```

lemma supp-Rep-formula [simp]: supp (Rep-formula x) = supp x
by (metis Rep-formula' Rep-formula-inverse supp-Abs-formula)

```

```

lemma supp-map-bset-Rep-formula [simp]: supp (map-bset Rep-formula xset) =
  supp xset
proof
  have eqvt (map-bset Rep-formula)
    unfolding eqvt-def by (simp add: ext)
  then show supp (map-bset Rep-formula xset)  $\subseteq$  supp xset
    by (fact supp-fun-app-eqvt)
next
  {

```

```

fix a :: atom
have inj (map-bset Rep-formula)
  by (metis bset.inj-map Rep-formula-inject injI)
then have  $\bigwedge x y. x \neq y \implies \text{map-bset Rep-formula } x \neq \text{map-bset Rep-formula } y$ 
  by (metis inj-eq)
then have  $\{b. (a = b) \cdot xset \neq xset\} \subseteq \{b. (a = b) \cdot \text{map-bset Rep-formula } xset \neq \text{map-bset Rep-formula } xset\}$  (is ?S  $\subseteq$  ?T)
  by auto
then have infinite ?S  $\implies$  infinite ?T
  by (metis infinite-super)
}
then show  $\text{supp } xset \subseteq \text{supp } (\text{map-bset Rep-formula } xset)$ 
  unfolding supp-def by auto
qed

```

Formulas are in fact finitely supported.

```

instance formula :: (type, fs, bn) fs
by standard (metis Rep-formula' hereditarily-fs-implies-finite-supp supp-Rep-formula)

```

5.7 Constructors for infinitary formulas

We lift the constructors for trees (modulo α -equivalence) to infinitary formulas. Since Conj_α does not necessarily yield a (hereditarily) finitely supported tree when applied to a (potentially infinite) set of (hereditarily) finitely supported trees, we cannot use Isabelle's **lift_definition** to define Conj . Instead, theorems about terms of the form $\text{Conj } xset$ will usually carry an assumption that $xset$ is finitely supported.

definition $\text{Conj} :: ('idx, 'pred, 'act) \text{ formula set}['idx] \Rightarrow ('idx, 'pred::fs, 'act::bn) \text{ formula where}$

$\text{Conj } xset = \text{Abs-formula } (\text{Conj}_\alpha (\text{map-bset Rep-formula } xset))$

lemma *finite-supp-implies-hereditarily-fs-Conj $_\alpha$ [simp]*:

assumes finite (supp xset)

shows hereditarily-fs (Conj $_\alpha$ (map-bset Rep-formula xset))

proof (rule hereditarily-fs.Conj $_\alpha$)

show finite (supp (map-bset Rep-formula xset))

using assms **by** (metis supp-map-bset-Rep-formula)

next

fix t_α **assume** $t_\alpha \in \text{set-bset } (\text{map-bset Rep-formula } xset)$

then show hereditarily-fs t_α

by (auto simp add: bset.set-map)

qed

lemma *Conj-rep-eq*:

assumes finite (supp xset)

shows $\text{Rep-formula } (\text{Conj } xset) = \text{Conj}_\alpha (\text{map-bset Rep-formula } xset)$

using assms **unfolding** Conj-def **by** simp

lift-definition $Not :: ('idx, 'pred, 'act) formula \Rightarrow ('idx, 'pred::fs, 'act::bn) formula$
is

Not_α
by (fact hereditarily-fs.Not $_\alpha$)

lift-definition $Pred :: 'pred \Rightarrow ('idx, 'pred::fs, 'act::bn) formula$ **is**

$Pred_\alpha$
by (fact hereditarily-fs.Pred $_\alpha$)

lift-definition $Act :: 'act \Rightarrow ('idx, 'pred, 'act) formula \Rightarrow ('idx, 'pred::fs, 'act::bn) formula$ **is**

Act_α
by (fact hereditarily-fs.Act $_\alpha$)

The lifted constructors are equivariant (in the case of *Conj*, on finitely supported arguments).

lemma *Conj-eqvt* [*simp*]:

assumes *finite* (*supp xset*)
shows $p \cdot Conj\ xset = Conj\ (p \cdot xset)$
using *assms unfolding Conj-def* **by** *simp*

lemma *Not-eqvt* [*eqvt, simp*]: $p \cdot Not\ x = Not\ (p \cdot x)$

by *transfer simp*

lemma *Pred-eqvt* [*eqvt, simp*]: $p \cdot Pred\ \varphi = Pred\ (p \cdot \varphi)$

by *transfer simp*

lemma *Act-eqvt* [*eqvt, simp*]: $p \cdot Act\ \alpha\ x = Act\ (p \cdot \alpha)\ (p \cdot x)$

by *transfer simp*

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:

assumes *finite* (*supp xset*)
shows $supp\ (Conj\ xset) = supp\ xset$
using *assms unfolding Conj-def* **by** *simp*

lemma *supp-Not* [*simp*]: $supp\ (Not\ x) = supp\ x$

by (*metis Not.rep-eq supp-Not $_\alpha$ supp-Rep-formula*)

lemma *supp-Pred* [*simp*]: $supp\ (Pred\ \varphi) = supp\ \varphi$

by (*metis Pred.rep-eq supp-Pred $_\alpha$ supp-Rep-formula*)

lemma *supp-Act* [*simp*]: $supp\ (Act\ \alpha\ x) = supp\ \alpha \cup supp\ x - bn\ \alpha$

by (*metis Act.rep-eq finite-supp supp-Act $_\alpha$ supp-Rep-formula*)

lemma *bn-fresh-Act* [*simp*]: $bn\ \alpha \#* Act\ \alpha\ x$

by (*simp add: fresh-def fresh-star-def*)

The lifted constructors are injective (except for *Act*).

lemma *Conj-eq-iff* [*simp*]:

assumes *finite (supp xset1)* **and** *finite (supp xset2)*

shows $\text{Conj } xset1 = \text{Conj } xset2 \longleftrightarrow xset1 = xset2$

using *assms*

by (*metis (erased, hide-lams) Conj_α-eq-iff Conj-rep-eq Rep-formula-inverse injI inj-eq bset.inj-map*)

lemma *Not-eq-iff* [*simp*]: $\text{Not } x1 = \text{Not } x2 \longleftrightarrow x1 = x2$

by (*metis Not.rep-eq Not_α-eq-iff Rep-formula-inverse*)

lemma *Pred-eq-iff* [*simp*]: $\text{Pred } \varphi1 = \text{Pred } \varphi2 \longleftrightarrow \varphi1 = \varphi2$

by (*metis Pred.rep-eq Pred_α-eq-iff*)

lemma *Act-eq-iff*: $\text{Act } \alpha1 \ x1 = \text{Act } \alpha2 \ x2 \longleftrightarrow \text{Act}_\alpha \ \alpha1 \ (\text{Rep-formula } x1) = \text{Act}_\alpha \ \alpha2 \ (\text{Rep-formula } x2)$

by (*metis Act.rep-eq Rep-formula-inverse*)

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: $\text{Act } \alpha1 \ x1 = \text{Act } \alpha2 \ x2 \longleftrightarrow$

$(\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2)$

(**is** *?l* \longleftrightarrow *?r*)

proof

assume *?l*

then obtain *p* **where** *alpha*: $(\text{bn } \alpha1, \text{rep-Tree}_\alpha (\text{Rep-formula } x1)) \approx_{\text{set}} (=_\alpha) (\text{supp-rel } (=_\alpha)) \ p \ (\text{bn } \alpha2, \text{rep-Tree}_\alpha (\text{Rep-formula } x2))$ **and** *eq*: $(\text{bn } \alpha1, \alpha1) \approx_{\text{set}} (=) \ \text{supp } p \ (\text{bn } \alpha2, \alpha2)$

by (*metis Act-eq-iff Act_α-eq-iff alpha-tAct*)

from *alpha* **have** $\text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2$

by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)

moreover from *alpha* **have** $(\text{supp } x1 - \text{bn } \alpha1) \#* p$

by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)

moreover from *alpha* **have** $p \cdot x1 = x2$

by (*metis Rep-formula-eqv Rep-formula-inject Tree_α.abs-eq-iff Tree_α-abs-rep alpha-Tree-permute-rep-commute alpha-set.simps*)

moreover from *eq* **have** $\text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2$

by (*metis alpha-set.simps*)

moreover from *eq* **have** $(\text{supp } \alpha1 - \text{bn } \alpha1) \#* p$

by (*metis alpha-set.simps*)

moreover from *eq* **have** $p \cdot \alpha1 = \alpha2$

by (*simp add: alpha-set.simps*)

ultimately show *?r*

by *metis*

next

assume *?r*

then obtain *p* **where** *1*: $\text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2$ **and** *2*: $(\text{supp } x1 - \text{bn } \alpha1) \#* p$ **and** *3*: $p \cdot x1 = x2$

and 4: $\text{supp } \alpha 1 - \text{bn } \alpha 1 = \text{supp } \alpha 2 - \text{bn } \alpha 2$ **and 5:** $(\text{supp } \alpha 1 - \text{bn } \alpha 1) \#^* p$
and 6: $p \cdot \alpha 1 = \alpha 2$
by *metis*
from 1 2 3 6 **have** $(\text{bn } \alpha 1, \text{rep-Tree}_\alpha (\text{Rep-formula } x1)) \approx_{\text{set}} (=_\alpha) (\text{supp-rel } (=_\alpha)) p (\text{bn } \alpha 2, \text{rep-Tree}_\alpha (\text{Rep-formula } x2))$
by *(metis (no-types, lifting) Rep-formula-eqvt alpha-Tree-permute-rep-commute alpha-set.simps bn-eqvt supp-Rep-formula supp-alpha-suppl-rel)*
moreover from 4 5 6 **have** $(\text{bn } \alpha 1, \alpha 1) \approx_{\text{set}} (=) \text{supp } p (\text{bn } \alpha 2, \alpha 2)$
by *(simp add: alpha-set.simps bn-eqvt)*
ultimately show $\text{Act } \alpha 1 x1 = \text{Act } \alpha 2 x2$
by *(metis Act-eq-iff Act_\alpha-eq-iff alpha-tAct)*
qed

lemma *Act-eq-iff-perm-renaming:* $\text{Act } \alpha 1 x1 = \text{Act } \alpha 2 x2 \longleftrightarrow$
 $(\exists p. \text{supp } x1 - \text{bn } \alpha 1 = \text{supp } x2 - \text{bn } \alpha 2 \wedge (\text{supp } x1 - \text{bn } \alpha 1) \#^* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha 1 - \text{bn } \alpha 1 = \text{supp } \alpha 2 - \text{bn } \alpha 2 \wedge (\text{supp } \alpha 1 - \text{bn } \alpha 1) \#^* p \wedge p \cdot \alpha 1 = \alpha 2 \wedge \text{supp } p \subseteq \text{bn } \alpha 1 \cup p \cdot \text{bn } \alpha 1)$
(is ?l \longleftrightarrow ?r)

proof

assume *?l*

then obtain *p* **where** $p: \text{supp } x1 - \text{bn } \alpha 1 = \text{supp } x2 - \text{bn } \alpha 2 \wedge (\text{supp } x1 - \text{bn } \alpha 1) \#^* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha 1 - \text{bn } \alpha 1 = \text{supp } \alpha 2 - \text{bn } \alpha 2 \wedge (\text{supp } \alpha 1 - \text{bn } \alpha 1) \#^* p \wedge p \cdot \alpha 1 = \alpha 2$
by *(metis Act-eq-iff-perm)*

moreover obtain *q* **where** $q-p: \forall b \in \text{bn } \alpha 1. q \cdot b = p \cdot b$ **and** $\text{supp } q: \text{supp } q \subseteq \text{bn } \alpha 1 \cup p \cdot \text{bn } \alpha 1$

by *(metis set-renaming-perm2)*

have $\text{supp } q \subseteq \text{supp } p$

proof

fix *a* **assume** $*$: $a \in \text{supp } q$ **then show** $a \in \text{supp } p$

proof *(cases a ∈ bn α1)*

case *True* **then show** *?thesis*

using $* q-p$ **by** *(metis mem-Collect-eq supp-perm)*

next

case *False* **then have** $a \in p \cdot \text{bn } \alpha 1$

using $* \text{supp } q$ **using** *UnE subsetCE* **by** *blast*

with *False* **have** $p \cdot a \neq a$

by *(metis mem-permute-iff)*

then show *?thesis*

using *fresh-def fresh-perm* **by** *blast*

qed

qed

with *p* **have** $(\text{supp } x1 - \text{bn } \alpha 1) \#^* q$ **and** $(\text{supp } \alpha 1 - \text{bn } \alpha 1) \#^* q$

by *(meson fresh-def fresh-star-def subset-iff)+*

moreover with *p* **and** $q-p$ **have** $\bigwedge a. a \in \text{supp } \alpha 1 \implies q \cdot a = p \cdot a$ **and** $\bigwedge a. a \in \text{supp } x1 \implies q \cdot a = p \cdot a$

by *(metis Diff-iff fresh-perm fresh-star-def)+*

then have $q \cdot \alpha 1 = p \cdot \alpha 1$ **and** $q \cdot x1 = p \cdot x1$

by *(metis supp-perm-perm-eq)+*

```

ultimately show ?r
  using supp-q by (metis bn-eqvt)
next
  assume ?r then show ?l
    by (meson Act-eq-iff-perm)
qed

```

The lifted constructors are free (except for *Act*).

```

lemma Tree-free [simp]:
  shows finite (supp xset)  $\implies$  Conj xset  $\neq$  Not x
  and finite (supp xset)  $\implies$  Conj xset  $\neq$  Pred  $\varphi$ 
  and finite (supp xset)  $\implies$  Conj xset  $\neq$  Act  $\alpha$  x
  and Not x  $\neq$  Pred  $\varphi$ 
  and Not x1  $\neq$  Act  $\alpha$  x2
  and Pred  $\varphi$   $\neq$  Act  $\alpha$  x
proof -
  show finite (supp xset)  $\implies$  Conj xset  $\neq$  Not x
    by (metis Conj-rep-eq Not.rep-eq Tree $_{\alpha}$ -free(1))
  next
  show finite (supp xset)  $\implies$  Conj xset  $\neq$  Pred  $\varphi$ 
    by (metis Conj-rep-eq Pred.rep-eq Tree $_{\alpha}$ -free(2))
  next
  show finite (supp xset)  $\implies$  Conj xset  $\neq$  Act  $\alpha$  x
    by (metis Conj-rep-eq Act.rep-eq Tree $_{\alpha}$ -free(3))
  next
  show Not x  $\neq$  Pred  $\varphi$ 
    by (metis Not.rep-eq Pred.rep-eq Tree $_{\alpha}$ -free(4))
  next
  show Not x1  $\neq$  Act  $\alpha$  x2
    by (metis Not.rep-eq Act.rep-eq Tree $_{\alpha}$ -free(5))
  next
  show Pred  $\varphi$   $\neq$  Act  $\alpha$  x
    by (metis Pred.rep-eq Act.rep-eq Tree $_{\alpha}$ -free(6))
qed

```

5.8 Induction over infinitary formulas

```

lemma formula-induct [case-names Conj Not Pred Act, induct type: formula]:
  fixes x
  assumes  $\bigwedge$ xset. finite (supp xset)  $\implies$  ( $\bigwedge$ x. x  $\in$  set-bset xset  $\implies$  P x)  $\implies$  P
  (Conj xset)
  and  $\bigwedge$ formula. P formula  $\implies$  P (Not formula)
  and  $\bigwedge$ pred. P (Pred pred)
  and  $\bigwedge$ act formula. P formula  $\implies$  P (Act act formula)
  shows P x
proof (induction x)
  fix t $_{\alpha}$  :: ('a,'b,'c) Tree $_{\alpha}$ 
  assume t $_{\alpha}$   $\in$  {t $_{\alpha}$ . hereditarily-fs t $_{\alpha}$ }
  then have hereditarily-fs t $_{\alpha}$ 

```

```

    by simp
  then show P (Abs-formula tα)
  proof (induction tα)
    case (Conjα tsetα) show ?case
    proof -
      let ?tset = map-bset Abs-formula tsetα
      have  $\bigwedge t_{\alpha}'. t_{\alpha}' \in \text{set-bset } tset_{\alpha} \implies t_{\alpha}' = (\text{Rep-formula} \circ \text{Abs-formula})$ 
      by (simp add: Conjα.hyps)
      then have tsetα = map-bset (Rep-formula ∘ Abs-formula) tsetα
      by (metis bset.map-cong0 bset.map-id id-apply)
      then have *: tsetα = map-bset Rep-formula ?tset
      by (metis bset.map-comp)
      then have Abs-formula (Conjα tsetα) = Conj ?tset
      by (metis Conj-def)
      moreover from * have finite (supp ?tset)
      using Conjα.hyps(1) by (metis supp-map-bset-Rep-formula)
      moreover have ( $\bigwedge t. t \in \text{set-bset } ?tset \implies P t$ )
      using Conjα.IH by (metis imageE map-bset.rep-eq)
      ultimately show ?thesis
      using assms(1) by metis
    qed
  next
  case Notα then show ?case
  using assms(2) by (metis Formula.Abs-formula-inverse Not.rep-eq Rep-formula-inverse)
  next
  case Predα show ?case
  using assms(3) by (metis Pred.abs-eq)
  next
  case Actα then show ?case
  using assms(4) by (metis Formula.Abs-formula-inverse Act.rep-eq Rep-formula-inverse)
  qed
qed

```

5.9 Strong induction over infinitary formulas

lemma *formula-strong-induct-aux*:

```

  fixes x
  assumes  $\bigwedge xset c. \text{finite } (\text{supp } xset) \implies (\bigwedge x. x \in \text{set-bset } xset \implies (\bigwedge c. P c x))$ 
   $\implies P c (\text{Conj } xset)$ 
  and  $\bigwedge \text{formula } c. (\bigwedge c. P c \text{ formula}) \implies P c (\text{Not formula})$ 
  and  $\bigwedge \text{pred } c. P c (\text{Pred pred})$ 
  and  $\bigwedge \text{act formula } c. \text{bn act } \#* c \implies (\bigwedge c. P c \text{ formula}) \implies P c (\text{Act act formula})$ 
  shows  $\bigwedge (c :: 'd::fs) p. P c (p \cdot x)$ 
  proof (induction x)
    case (Conj xset)
    moreover then have finite (supp (p · xset))
    by (metis permute-finite supp-eqvt)

```

```

moreover have ( $\bigwedge x c. x \in \text{set-bset } (p \cdot \text{xset}) \implies P c x$ )
using Conj.IH by (metis (full-types) eqvt-bound mem-permute-iff set-bset-eqvt)
ultimately show ?case
  using assms(1) by simp
next
case Not then show ?case
  using assms(2) by simp
next
case Pred show ?case
  using assms(3) by simp
next
case (Act  $\alpha x$ ) show ?case
proof –
  — rename bn ( $p \cdot \alpha$ ) to avoid c, without touching Act ( $p \cdot \alpha$ ) ( $p \cdot x$ )
obtain q where 1: ( $q \cdot \text{bn } (p \cdot \alpha)$ )  $\#* c$  and 2: supp (Act ( $p \cdot \alpha$ ) ( $p \cdot x$ ))  $\#* q$ 
proof (rule at-set-avoiding2[of bn (p · α) c Act (p · α) (p · x), THEN exE])
  show finite (bn ( $p \cdot \alpha$ )) by (fact bn-finite)
  next
  show finite (supp c) by (fact finite-supp)
  next
  show finite (supp (Act ( $p \cdot \alpha$ ) ( $p \cdot x$ ))) by (simp add: finite-supp)
  next
  show bn ( $p \cdot \alpha$ )  $\#* \text{Act } (p \cdot \alpha) (p \cdot x)$  by (simp add: fresh-def fresh-star-def)
qed metis
from 1 have bn ( $q \cdot p \cdot \alpha$ )  $\#* c$ 
  by (simp add: bn-eqvt)
moreover from Act.IH have  $\bigwedge c. P c (q \cdot p \cdot x)$ 
  by (metis permute-plus)
ultimately have  $P c (\text{Act } (q \cdot p \cdot \alpha) (q \cdot p \cdot x))$ 
  using assms(4) by simp
moreover from 2 have  $\text{Act } (q \cdot p \cdot \alpha) (q \cdot p \cdot x) = \text{Act } (p \cdot \alpha) (p \cdot x)$ 
  using supp-perm-eq by fastforce
ultimately show ?thesis
  by simp
qed
qed

lemmas formula-strong-induct = formula-strong-induct-aux[where  $p=0$ , simplified]
declare formula-strong-induct [case-names Conj Not Pred Act]

end
theory Validity
imports
  Transition-System
  Formula
begin

```

6 Validity

The following is needed to prove termination of *validTree*.

definition *alpha-Tree-rel* **where**
 $\text{alpha-Tree-rel} \equiv \{(x,y). x =_{\alpha} y\}$

lemma *alpha-Tree-relI* [*simp*]:
assumes $x =_{\alpha} y$ **shows** $(x,y) \in \text{alpha-Tree-rel}$
using *assms unfolding alpha-Tree-rel-def by simp*

lemma *alpha-Tree-relE*:
assumes $(x,y) \in \text{alpha-Tree-rel}$ **and** $x =_{\alpha} y \implies P$
shows P
using *assms unfolding alpha-Tree-rel-def by simp*

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
 $wf (\text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf})$
proof (*rule wf-relcomp-compatible*)
show $wf (\text{hull-rel } O \text{ Tree-wf})$
by (*metis Tree-wf-eqt' wf-Tree-wf wf-hull-rel-relcomp*)
next
show $(\text{hull-rel } O \text{ Tree-wf}) O \text{ alpha-Tree-rel} \subseteq \text{alpha-Tree-rel } O (\text{hull-rel } O \text{ Tree-wf})$
proof
fix $x :: ('d, 'e, 'f) \text{ Tree} \times ('d, 'e, 'f) \text{ Tree}$
assume $x \in (\text{hull-rel } O \text{ Tree-wf}) O \text{ alpha-Tree-rel}$
then obtain $x1\ x2\ x3\ x4$ **where** $x: x = (x1, x4)$ **and** $1: (x1, x2) \in \text{hull-rel}$ **and**
 $2: (x2, x3) \in \text{Tree-wf}$ **and** $3: (x3, x4) \in \text{alpha-Tree-rel}$
by *auto*
from 2 **have** $(x1, x4) \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$
using 1 **and** 3 **proof** (*induct rule: Tree-wf.induct*)
— *tConj*
fix t **and** $tset :: ('d, 'e, 'f) \text{ Tree set}['d]$
assume $*$: $t \in \text{set-bset } tset$ **and** $**$: $(x1, t) \in \text{hull-rel}$ **and** $***$: $(tConj\ tset,$
 $x4) \in \text{alpha-Tree-rel}$
from $**$ **obtain** p **where** $x1: x1 = p \cdot t$
using *hull-rel.cases by blast*
from $***$ **have** $tConj\ tset =_{\alpha} x4$
by (*rule alpha-Tree-relE*)
then obtain $tset'$ **where** $x4: x4 = tConj\ tset'$ **and** $\text{rel-bset} (=_{\alpha})\ tset\ tset'$
by (*cases x4*) *simp-all*
with $*$ **obtain** t' **where** $t': t' \in \text{set-bset } tset'$ **and** $t =_{\alpha} t'$
by (*metis rel-bset.rep-eq rel-set-def*)
with $x1$ **have** $(x1, p \cdot t') \in \text{alpha-Tree-rel}$
by (*metis Tree_{\alpha}.abs-eq-iff alpha-Tree-relI permute-Tree_{\alpha}.abs-eq*)
moreover **have** $(p \cdot t', t') \in \text{hull-rel}$
by (*rule hull-rel.intros*)
moreover **from** $x4$ **and** t' **have** $(t', x4) \in \text{Tree-wf}$
by (*simp add: Tree-wf.intros(1)*)
ultimately show $(x1, x4) \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$

```

    by auto
next
  — tNot
  fix t
  assume *: (x1, t) ∈ hull-rel and **: (tNot t, x4) ∈ alpha-Tree-rel
  from * obtain p where x1: x1 = p · t
    using hull-rel.cases by blast
  from ** have tNot t =α x4
    by (rule alpha-Tree-relE)
  then obtain t' where x4: x4 = tNot t' and t =α t'
    by (cases x4) simp-all
  with x1 have (x1, p · t') ∈ alpha-Tree-rel
    by (metis Tree_α.abs-eq-iff alpha-Tree-relI permute-Tree_α.abs-eq x1)
  moreover have (p · t', t') ∈ hull-rel
    by (rule hull-rel.intros)
  moreover from x4 have (t', x4) ∈ Tree-wf
    using Tree-wf.intros(2) by blast
  ultimately show (x1, x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
    by auto
next
  — tAct
  fix α t
  assume *: (x1, t) ∈ hull-rel and **: (tAct α t, x4) ∈ alpha-Tree-rel
  from * obtain p where x1: x1 = p · t
    using hull-rel.cases by blast
  from ** have tAct α t =α x4
    by (rule alpha-Tree-relE)
  then obtain q t' where x4: x4 = tAct (q · α) t' and q · t =α t'
    by (cases x4) (auto simp add: alpha-set)
  with x1 have (x1, p · −q · t') ∈ alpha-Tree-rel
  by (metis Tree_α.abs-eq-iff alpha-Tree-relI permute-Tree_α.abs-eq permute-minus-cancel(1))
  moreover have (p · −q · t', t') ∈ hull-rel
    by (metis hull-rel.simps permute-plus)
  moreover from x4 have (t', x4) ∈ Tree-wf
    by (simp add: Tree-wf.intros(3))
  ultimately show (x1, x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
    by auto
qed
with x show x ∈ alpha-Tree-rel O hull-rel O Tree-wf
  by simp
qed
qed

lemma alpha-Tree-rel-relcomp-trivialI [simp]:
  assumes (x, y) ∈ R
  shows (x, y) ∈ alpha-Tree-rel O R
using assms unfolding alpha-Tree-rel-def
by (metis Tree_α.abs-eq-iff case-prodI mem-Collect-eq relcomp.relcompI)

```

lemma *alpha-Tree-rel-relcompI* [*simp*]:
assumes $x =_\alpha x'$ **and** $(x', y) \in R$
shows $(x, y) \in \text{alpha-Tree-rel } O R$
using *assms unfolding alpha-Tree-rel-def*
by (*metis case-prodI mem-Collect-eq relcomp.relcompI*)

6.1 Validity for infinitely branching trees

context *nominal-ts*
begin

Since we defined formulas via a manual quotient construction, we also need to define validity via lifting from the underlying type of infinitely branching trees. We cannot use **nominal function** because that generates proof obligations where, for formulas of the form *Conj xset*, the assumption that *xset* has finite support is missing.

declare *conj-cong* [*fundef-cong*]

function *valid-Tree* :: *'state* \Rightarrow (*'idx, 'pred, 'act*) *Tree* \Rightarrow *bool* **where**
valid-Tree *P* (*tConj tset*) \longleftrightarrow ($\forall t \in \text{set-bset } tset. \text{valid-Tree } P t$)
| *valid-Tree* *P* (*tNot t*) \longleftrightarrow $\neg \text{valid-Tree } P t$
| *valid-Tree* *P* (*tPred* φ) \longleftrightarrow $P \vdash \varphi$
| *valid-Tree* *P* (*tAct* αt) \longleftrightarrow ($\exists \alpha' t' P'. tAct \alpha t =_\alpha tAct \alpha' t' \wedge P \rightarrow \langle \alpha', P' \rangle$
 $\wedge \text{valid-Tree } P' t'$)

by *pat-completeness auto*

termination proof

let $?R = \text{inv-image } (\text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}) \text{ snd}$
{
show *wf ?R*
by (*metis wf-alpha-Tree-rel-hull-rel-Tree-wf wf-inv-image*)
next
fix *P* :: *'state* **and** *tset* :: (*'idx, 'pred, 'act*) *Tree* *set* [*'idx*] **and** *t*
assume $t \in \text{set-bset } tset$ **then show** $((P, t), (P, tConj tset)) \in ?R$
by (*simp add: Tree-wf.intros(1)*)
next
fix *P* :: *'state* **and** *t* :: (*'idx, 'pred, 'act*) *Tree*
show $((P, t), (P, tNot t)) \in ?R$
by (*simp add: Tree-wf.intros(2)*)
next
fix *P1 P2* :: *'state* **and** $\alpha 1 \alpha 2$:: *'act* **and** *t1 t2* :: (*'idx, 'pred, 'act*) *Tree*
assume $tAct \alpha 1 t1 =_\alpha tAct \alpha 2 t2$
then obtain *p* **where** $t2 =_\alpha p \cdot t1$
by (*auto simp add: alphas*) (*metis alpha-Tree-symp sympE*)
then show $((P2, t2), (P1, tAct \alpha 1 t1)) \in ?R$
by (*simp add: Tree-wf.intros(3)*)
}
qed

valid-Tree is equivariant.


```

lemma valid-Tree-eqt': valid-Tree P t  $\longleftrightarrow$  valid-Tree (p · P) (p · t)
proof (induction P t rule: valid-Tree.induct)
  case (1 P tset) show ?case
    proof
      assume *: valid-Tree P (tConj tset)
      {
        fix t
        assume t ∈ p · set-bset tset
        with 1.IH and * have valid-Tree (p · P) t
        by (metis (no-types, lifting) imageE permute-set-eq-image valid-Tree.simps(1))
      }
      then show valid-Tree (p · P) (p · tConj tset)
        by simp
    next
      assume *: valid-Tree (p · P) (p · tConj tset)
      {
        fix t
        assume t ∈ set-bset tset
        with 1.IH and * have valid-Tree P t
        by (metis mem-permute-iff permute-Tree-tConj set-bset-eqt valid-Tree.simps(1))
      }
      then show valid-Tree P (tConj tset)
        by simp
    qed
  next
    case 2 then show ?case by simp
  next
    case 3 show ?case by simp (metis permute-minus-cancel(2) satisfies-eqt)
  next
    case (4 P α t) show ?case
      proof
        assume valid-Tree P (tAct α t)
        then obtain  $\alpha' t' P'$  where *:  $tAct \alpha t =_{\alpha} tAct \alpha' t' \wedge P \rightarrow \langle \alpha', P^{\wedge} \rangle \wedge$ 
valid-Tree P' t'
        by auto
        with 4.IH have valid-Tree (p · P') (p · t')
        by blast
        moreover from * have  $p \cdot P \rightarrow \langle p \cdot \alpha', p \cdot P^{\wedge} \rangle$ 
        by (metis transition-eqt')
        moreover from * have  $p \cdot tAct \alpha t =_{\alpha} tAct (p \cdot \alpha') (p \cdot t')$ 
        by (metis alpha-Tree-eqt permute-Tree.simps(4))
        ultimately show valid-Tree (p · P) (p · tAct α t)
        by auto
      next
        assume valid-Tree (p · P) (p · tAct α t)
        then obtain  $\alpha' t' P'$  where *:  $p \cdot tAct \alpha t =_{\alpha} tAct \alpha' t' \wedge (p \cdot P) \rightarrow$ 
 $\langle \alpha', P^{\wedge} \rangle \wedge$  valid-Tree P' t'
        by auto
        then have  $eq: tAct \alpha t =_{\alpha} tAct (-p \cdot \alpha') (-p \cdot t')$ 

```

```

    by (metis alpha-Tree-eqvt permute-Tree.simps(4) permute-minus-cancel(2))
  moreover from * have  $P \rightarrow \langle -p \cdot \alpha', -p \cdot P' \rangle$ 
    by (metis permute-minus-cancel(2) transition-eqvt')
  moreover with 4.IH have  $\text{valid-Tree } (-p \cdot P') (-p \cdot t')$ 
    using eq and * by simp
  ultimately show  $\text{valid-Tree } P (tAct \alpha t)$ 
    by auto
qed
qed

```

```

lemma valid-Tree-eqvt :
  assumes  $\text{valid-Tree } P t$  shows  $\text{valid-Tree } (p \cdot P) (p \cdot t)$ 
  using assms by (metis valid-Tree-eqvt')

```

α -equivalent trees validate the same states.

```

lemma alpha-Tree-valid-Tree:
  assumes  $t1 =_\alpha t2$ 
  shows  $\text{valid-Tree } P t1 \longleftrightarrow \text{valid-Tree } P t2$ 
  using assms proof (induction t1 t2 arbitrary: P rule: alpha-Tree-induct)
  case tConj then show ?case
    by auto (metis (mono-tags) rel-bset.rep-eq rel-set-def)+
  next
  case (tAct  $\alpha1 t1 \alpha2 t2$ ) show ?case
  proof
    assume  $\text{valid-Tree } P (tAct \alpha1 t1)$ 
    then obtain  $\alpha' t' P'$  where  $tAct \alpha1 t1 =_\alpha tAct \alpha' t' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge$ 
 $\text{valid-Tree } P' t'$ 
      by auto
    moreover from tAct.hyps have  $tAct \alpha1 t1 =_\alpha tAct \alpha2 t2$ 
      using alpha-tAct by blast
    ultimately show  $\text{valid-Tree } P (tAct \alpha2 t2)$ 
      by (metis Tree $_\alpha$ .abs-eq-iff valid-Tree.simps(4))
  next
  assume  $\text{valid-Tree } P (tAct \alpha2 t2)$ 
  then obtain  $\alpha' t' P'$  where  $tAct \alpha2 t2 =_\alpha tAct \alpha' t' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge$ 
 $\text{valid-Tree } P' t'$ 
    by auto
  moreover from tAct.hyps have  $tAct \alpha1 t1 =_\alpha tAct \alpha2 t2$ 
    using alpha-tAct by blast
  ultimately show  $\text{valid-Tree } P (tAct \alpha1 t1)$ 
    by (metis Tree $_\alpha$ .abs-eq-iff valid-Tree.simps(4))
  qed
qed simp-all

```

6.2 Validity for trees modulo α -equivalence

```

lift-definition valid-Tree $_\alpha$  :: 'state  $\Rightarrow$  ('idx,'pred,'act) Tree $_\alpha \Rightarrow$  bool is
  valid-Tree
  by (fact alpha-Tree-valid-Tree)

```

lemma *valid-Tree_α-eqvt* :
assumes *valid-Tree_α P t* **shows** *valid-Tree_α (p · P) (p · t)*
using *assms* **by** *transfer (fact valid-Tree-eqvt)*

lemma *valid-Tree_α-Conj_α [simp]*: *valid-Tree_α P (Conj_α tset_α)* \longleftrightarrow $(\forall t_\alpha \in \text{set-bset } tset_\alpha. \text{valid-Tree}_\alpha P t_\alpha)$

proof –
have *valid-Tree P (rep-Tree_α (abs-Tree_α (tConj (map-bset rep-Tree_α tset_α))))*
 \longleftrightarrow *valid-Tree P (tConj (map-bset rep-Tree_α tset_α))*
by (*metis Tree_α-rep-abs alpha-Tree-valid-Tree*)
then show *?thesis*
by (*simp add: valid-Tree_α-def Conj_α-def map-bset.rep-eq*)
qed

lemma *valid-Tree_α-Not_α [simp]*: *valid-Tree_α P (Not_α t_α)* \longleftrightarrow $\neg \text{valid-Tree}_\alpha P t_\alpha$
by *transfer simp*

lemma *valid-Tree_α-Pred_α [simp]*: *valid-Tree_α P (Pred_α φ)* \longleftrightarrow $P \vdash \varphi$
by *transfer simp*

lemma *valid-Tree_α-Act_α [simp]*: *valid-Tree_α P (Act_α α t_α)* \longleftrightarrow $(\exists \alpha' t_\alpha' P'. \text{Act}_\alpha \alpha t_\alpha = \text{Act}_\alpha \alpha' t_\alpha' \wedge P \rightarrow \langle \alpha', P \rangle \wedge \text{valid-Tree}_\alpha P' t_\alpha')$

proof
assume *valid-Tree_α P (Act_α α t_α)*
moreover have *Act_α α t_α = abs-Tree_α (tAct α (rep-Tree_α t_α))*
by (*metis Act_α.abs-eq Tree_α-abs-rep*)
ultimately show $\exists \alpha' t_\alpha' P'. \text{Act}_\alpha \alpha t_\alpha = \text{Act}_\alpha \alpha' t_\alpha' \wedge P \rightarrow \langle \alpha', P \rangle \wedge \text{valid-Tree}_\alpha P' t_\alpha'$
by (*metis Act_α.abs-eq Tree_α.abs-eq-iff valid-Tree.simps(4) valid-Tree_α.abs-eq*)
next
assume $\exists \alpha' t_\alpha' P'. \text{Act}_\alpha \alpha t_\alpha = \text{Act}_\alpha \alpha' t_\alpha' \wedge P \rightarrow \langle \alpha', P \rangle \wedge \text{valid-Tree}_\alpha P' t_\alpha'$
moreover have $\bigwedge \alpha' t_\alpha'. \text{Act}_\alpha \alpha' t_\alpha' = \text{abs-Tree}_\alpha (tAct \alpha' (rep-Tree_\alpha t_\alpha'))$
by (*metis Act_α.abs-eq Tree_α-abs-rep*)
ultimately show *valid-Tree_α P (Act_α α t_α)*
by (*metis Tree_α.abs-eq-iff valid-Tree.simps(4) valid-Tree_α.abs-eq valid-Tree_α.rep-eq*)
qed

6.3 Validity for infinitary formulas

lift-definition *valid* :: *'state* \Rightarrow (*'idx, 'pred, 'act*) *formula* \Rightarrow *bool* (**infix** \models 70) is
valid-Tree_α

.

lemma *valid-eqvt* :
assumes $P \models x$ **shows** $(p \cdot P) \models (p \cdot x)$
using *assms* **by** *transfer (metis valid-Tree_α-eqvt)*

lemma *valid-Conj* [*simp*]:
assumes *finite* (*supp* *xset*)
shows $P \models \text{Conj } xset \longleftrightarrow (\forall x \in \text{set-bset } xset. P \models x)$
using *assms* **by** (*simp* *add: valid-def Conj-def map-bset.rep-eq*)

lemma *valid-Not* [*simp*]: $P \models \text{Not } x \longleftrightarrow \neg P \models x$
by *transfer simp*

lemma *valid-Pred* [*simp*]: $P \models \text{Pred } \varphi \longleftrightarrow P \vdash \varphi$
by *transfer simp*

lemma *valid-Act*: $P \models \text{Act } \alpha \ x \longleftrightarrow (\exists \alpha' \ x' \ P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x')$

proof

assume $P \models \text{Act } \alpha \ x$

moreover **have** $\text{Rep-formula } (\text{Abs-formula } (\text{Act}_\alpha \ \alpha \ (\text{Rep-formula } x))) = \text{Act}_\alpha \ \alpha \ (\text{Rep-formula } x)$

by (*metis Act.rep-eq Rep-formula-inverse*)

ultimately **show** $\exists \alpha' \ x' \ P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$

by (*auto simp add: valid-def Act-def*) (*metis Abs-formula-inverse Rep-formula' hereditarily-fs-alpha-renaming*)

next

assume $\exists \alpha' \ x' \ P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$

then **show** $P \models \text{Act } \alpha \ x$

by (*metis Act.rep-eq valid.rep-eq valid-Tree $_\alpha$ -Act $_\alpha$*)

qed

The binding names in the alpha-variant that witnesses validity may be chosen fresh for any finitely supported context.

lemma *valid-Act-strong*:

assumes *finite* (*supp* *X*)

shows $P \models \text{Act } \alpha \ x \longleftrightarrow (\exists \alpha' \ x' \ P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \ \#^* X)$

proof

assume $P \models \text{Act } \alpha \ x$

then **obtain** $\alpha' \ x' \ P'$ **where** *eq*: $\text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P$ **and** *trans*: $P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$

by (*metis valid-Act*)

have *finite* (*bn* α')

by (*fact bn-finite*)

moreover **note** (*finite* (*supp* *X*))

moreover **have** *finite* (*supp* ($\text{Act } \alpha' \ x', \langle \alpha', P' \rangle$))

by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)

moreover **have** $\text{bn } \alpha' \ \#^* (\text{Act } \alpha' \ x', \langle \alpha', P' \rangle)$

by (*auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)

ultimately **obtain** *p* **where** *fresh-X*: $(p \cdot \text{bn } \alpha') \ \#^* X$ **and** *supp* ($\text{Act } \alpha' \ x', \langle \alpha', P' \rangle$) $\#^* p$

by (*metis at-set-avoiding2*)

then have $\text{supp } (Act \alpha' x') \#* p$ **and** $\text{supp } \langle \alpha', P^\wedge \rangle \#* p$
by (*metis fresh-star-Un supp-Pair*)
then have $Act (p \cdot \alpha') (p \cdot x') = Act \alpha' x'$ **and** $\langle p \cdot \alpha', p \cdot P^\wedge \rangle = \langle \alpha', P^\wedge \rangle$
by (*metis Act-eqvt supp-perm-eq, metis abs-residual-pair-eqvt supp-perm-eq*)
then show $\exists \alpha' x' P'. Act \alpha x = Act \alpha' x' \wedge P \rightarrow \langle \alpha', P^\wedge \rangle \wedge P' \models x' \wedge \text{bn } \alpha'$
 $\#* X$
using eq and trans and valid and fresh-X by (*metis bn-eqvt valid-eqvt*)
next
assume $\exists \alpha' x' P'. Act \alpha x = Act \alpha' x' \wedge P \rightarrow \langle \alpha', P^\wedge \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \#*$
 X
then show $P \models Act \alpha x$
by (*metis valid-Act*)
qed

lemma *valid-Act-fresh*:
assumes $\text{bn } \alpha \#* P$
shows $P \models Act \alpha x \longleftrightarrow (\exists P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge P' \models x)$
proof
assume $P \models Act \alpha x$

moreover have *finite* ($\text{supp } P$)
by (*fact finite-supp*)
ultimately obtain $\alpha' x' P'$ **where**
 $\text{eq: } Act \alpha x = Act \alpha' x'$ **and** $\text{trans: } P \rightarrow \langle \alpha', P^\wedge \rangle$ **and** $\text{valid: } P' \models x'$ **and**
fresh: bn $\alpha' \#* P'$
by (*metis valid-Act-strong*)

from eq obtain p **where** $p\text{-}\alpha: \alpha' = p \cdot \alpha$ **and** $p\text{-}x: x' = p \cdot x$ **and** $\text{supp-}p:$
 $\text{supp } p \subseteq \text{bn } \alpha \cup p \cdot \text{bn } \alpha$
by (*metis Act-eq-iff-perm-renaming*)

from assms and fresh have $(\text{bn } \alpha \cup p \cdot \text{bn } \alpha) \#* P$
using $p\text{-}\alpha$ **by** (*metis bn-eqvt fresh-star-Un*)
then have $\text{supp } p \#* P$
using $\text{supp-}p$ **by** (*metis fresh-star-def subset-eq*)
then have $p\text{-}P: -p \cdot P = P$
by (*metis perm-supp-eq supp-minus-perm*)

from trans have $P \rightarrow \langle \alpha, -p \cdot P^\wedge \rangle$
using $p\text{-}P$ $p\text{-}\alpha$ **by** (*metis permute-minus-cancel(1) transition-eqvt'*)
moreover from valid have $-p \cdot P' \models x$
using $p\text{-}x$ **by** (*metis permute-minus-cancel(1) valid-eqvt*)
ultimately show $\exists P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge P' \models x$
by *meson*

next
assume $\exists P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge P' \models x$ **then show** $P \models Act \alpha x$
by (*metis valid-Act*)
qed

end

end

theory *Logical-Equivalence*

imports

Validity

begin

7 (Strong) Logical Equivalence

The definition of formulas is parametric in the index type, but from now on we want to work with a fixed (sufficiently large) index type.

locale *indexed-nominal-ts = nominal-ts satisfies transition*

for *satisfies* :: 'state::fs \Rightarrow 'pred::fs \Rightarrow bool (**infix** \vdash 70)

and *transition* :: 'state \Rightarrow ('act::bn, 'state) residual \Rightarrow bool (**infix** \rightarrow 70) +

assumes *card-idx-perm*: |UNIV::perm set| < o |UNIV::'idx set|

and *card-idx-state*: |UNIV::'state set| < o |UNIV::'idx set|

begin

definition *logically-equivalent* :: 'state \Rightarrow 'state \Rightarrow bool **where**

logically-equivalent P Q \equiv ($\forall x::('idx, 'pred, 'act)$ formula. P \models x \longleftrightarrow Q \models x)

notation *logically-equivalent* (**infix** = 50)

lemma *logically-equivalent-eqvt*:

assumes P = Q **shows** p \cdot P = p \cdot Q

using *assms unfolding logically-equivalent-def*

by (*metis (mono-tags) permute-minus-cancel(1) valid-eqvt*)

end

end

theory *Bisimilarity-Implies-Equivalence*

imports

Logical-Equivalence

begin

8 Bisimilarity Implies Logical Equivalence

context *indexed-nominal-ts*

begin

lemma *bisimilarity-implies-equivalence-Act*:

assumes $\bigwedge P Q. P \sim Q \implies P \models x \longleftrightarrow Q \models x$

and P \sim Q

and P \models Act α x

shows Q \models Act α x

proof –
have $\text{finite } (\text{supp } Q)$
by (*fact finite-supp*)
with $\langle P \models \text{Act } \alpha x \rangle$ **obtain** $\alpha' x' P'$ **where** $\text{eq}: \text{Act } \alpha x = \text{Act } \alpha' x'$ **and**
 $\text{trans}: P \rightarrow \langle \alpha', P' \rangle$ **and** $\text{valid}: P' \models x'$ **and** $\text{fresh}: \text{bn } \alpha' \#* Q$
by (*metis valid-Act-strong*)

from $\langle P \sim Q \rangle$ **and** fresh **and** trans **obtain** Q' **where** $\text{trans}': Q \rightarrow \langle \alpha', Q' \rangle$
and $\text{bisim}': P' \sim Q'$
by (*metis bisimilar-simulation-step*)

from eq **obtain** p **where** $\text{px}: x' = p \cdot x$
by (*metis Act-eq-iff-perm*)

with valid **have** $-p \cdot P' \models x$
by (*metis permute-minus-cancel(1) valid-eqv*)
moreover from bisim' **have** $(-p \cdot P') \sim (-p \cdot Q')$
by (*metis bisimilar-eqv*)
ultimately have $-p \cdot Q' \models x$
using $\langle \wedge P Q. P \sim Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **by** *metis*
with px **have** $Q' \models x'$
by (*metis permute-minus-cancel(1) valid-eqv*)

with eq **and** trans' **show** $Q \models \text{Act } \alpha x$
unfolding *valid-Act* **by** *metis*
qed

theorem *bisimilarity-implies-equivalence*: **assumes** $P \sim Q$ **shows** $P = Q$
unfolding *logically-equivalent-def* **proof**
fix $x :: ('idx, 'pred, 'act) \text{ formula}$
from *assms* **show** $P \models x \longleftrightarrow Q \models x$
proof (*induction x arbitrary: P Q*)
case (*Conj xset*) **then show** *?case*
by *simp*
next
case *Not* **then show** *?case*
by *simp*
next
case *Pred* **then show** *?case*
by (*metis bisimilar-is-bisimulation is-bisimulation-def symp-def valid-Pred*)
next
case (*Act αx*) **then show** *?case*
by (*metis bisimilar-symp bisimilarity-implies-equivalence-Act sympE*)
qed
qed

end

end

```

theory Equivalence-Implies-Bisimilarity
imports
  Logical-Equivalence
begin

```

9 Logical Equivalence Implies Bisimilarity

```

context indexed-nominal-ts
begin

```

```

definition distinguishing-formula :: ('idx, 'pred, 'act) formula  $\Rightarrow$  'state  $\Rightarrow$  'state
 $\Rightarrow$  bool
  (- distinguishes - from - [100,100,100] 100)
where
  x distinguishes P from Q  $\equiv$   $P \models x \wedge \neg Q \models x$ 

```

```

lemma distinguishing-formula-eqvt [eqvt]:
  assumes x distinguishes P from Q shows  $(p \cdot x)$  distinguishes  $(p \cdot P)$  from
 $(p \cdot Q)$ 
  using assms unfolding distinguishing-formula-def
  by (metis permute-minus-cancel(2) valid-eqvt)

```

```

lemma equivalent-iff-not-distinguished:  $(P \equiv Q) \iff \neg(\exists x. x \text{ distinguishes } P$ 
from  $Q)$ 
  by (metis (full-types) distinguishing-formula-def logically-equivalent-def valid-Not)

```

There exists a distinguishing formula for P and Q whose support is contained in $\text{supp } P$.

```

lemma distinguished-bounded-support:
  assumes x distinguishes P from Q
  obtains y where  $\text{supp } y \subseteq \text{supp } P$  and y distinguishes P from Q
proof -
  let  $?B = \{p \cdot x \mid p. \text{supp } P \#* p\}$ 
  have  $\text{supp } P \text{ supports } ?B$ 
  unfolding supports-def proof (clarify)
    fix a b
    assume a:  $a \notin \text{supp } P$  and b:  $b \notin \text{supp } P$ 
    have  $(a \equiv b) \cdot ?B \subseteq ?B$ 
    proof
      fix x'
      assume  $x' \in (a \equiv b) \cdot ?B$ 
      then obtain p where 1:  $x' = (a \equiv b) \cdot p \cdot x$  and 2:  $\text{supp } P \#* p$ 
        by (auto simp add: permute-set-def)
      let  $?q = (a \equiv b) + p$ 
      from 1 have  $x' = ?q \cdot x$ 
        by simp
      moreover from a and b and 2 have  $\text{supp } P \#* ?q$ 
        by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))
      ultimately show  $x' \in ?B$  by blast

```



```

qed
moreover have ?B ⊆ (a ⇒ b) · ?B
proof
  fix x'
  assume x' ∈ ?B
  then obtain p where 1: x' = p · x and 2: supp P #* p
    by auto
  let ?q = (a ⇒ b) + p
  from 1 have x' = (a ⇒ b) · ?q · x
    by simp
  moreover from a and b and 2 have supp P #* ?q
    by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))
  ultimately show x' ∈ (a ⇒ b) · ?B
    using mem-permute-iff by blast
qed
ultimately show (a ⇒ b) · ?B = ?B ..
qed
then have supp-B-subset-supp-P: supp ?B ⊆ supp P
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have finite-supp-B: finite (supp ?B)
  using finite-supp rev-finite-subset by blast
have ?B ⊆ (λp. p · x) ‘ UNIV
  by auto
then have |?B| ≤o |UNIV :: perm set|
  by (rule surj-imp-ordLeq)
also have |UNIV :: perm set| <o |UNIV :: 'idx set|
  by (metis card-idx-perm)
also have |UNIV :: 'idx set| ≤o natLeq + c |UNIV :: 'idx set|
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-B: |?B| <o natLeq + c |UNIV :: 'idx set| .
let ?y = Conj (Abs-bset ?B) :: ('idx, 'pred, 'act) formula
  from finite-supp-B and card-B and supp-B-subset-supp-P have supp ?y ⊆
supp P
  by simp
moreover have ?y distinguishes P from Q
  unfolding distinguishing-formula-def proof
    from assms show P ⊨ ?y
      by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
supp-perm-eq valid-eqvt)
    next
      from assms show ¬ Q ⊨ ?y
        by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
  qed
ultimately show ?thesis ..
qed

lemma equivalence-is-bisimulation: is-bisimulation logically-equivalent
proof –

```

```

have symp logically-equivalent
  by (metis logically-equivalent-def sympI)
moreover
{
  fix  $P Q \varphi$  assume  $P =\cdot Q$  then have  $P \vdash \varphi \longrightarrow Q \vdash \varphi$ 
  by (metis logically-equivalent-def valid-Pred)
}
moreover
{
  fix  $P Q \alpha P'$  assume  $P =\cdot Q$  and  $bn \alpha \#* Q$  and  $P \rightarrow \langle \alpha, P \rangle$ 
then have  $\exists Q'. Q \rightarrow \langle \alpha, Q \rangle \wedge P' =\cdot Q'$ 
  proof –
  {
    let  $?Q' = \{Q'. Q \rightarrow \langle \alpha, Q \rangle\}$ 
    assume  $\forall Q' \in ?Q'. \neg P' =\cdot Q'$ 
    then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } x \text{ distinguishes}$ 
     $P' \text{ from } Q'$ 
    by (metis equivalent-iff-not-distinguished)
    then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } \text{supp } x \subseteq \text{supp}$ 
     $P' \wedge x \text{ distinguishes } P' \text{ from } Q'$ 
    by (metis distinguished-bounded-support)
    then obtain  $f :: 'state \Rightarrow ('idx, 'pred, 'act) \text{ formula}$  where
     $*: \forall Q' \in ?Q'. \text{supp } (f Q') \subseteq \text{supp } P' \wedge (f Q') \text{ distinguishes } P' \text{ from } Q'$ 
    by metis
    have  $\text{supp } P' \text{ supports } (f \text{ ' } ?Q')$ 
    unfolding supports-def proof (clarify)
    fix  $a b$ 
    assume  $a: a \notin \text{supp } P'$  and  $b: b \notin \text{supp } P'$ 
    have  $(a \Rightarrow b) \cdot (f \text{ ' } ?Q') \subseteq f \text{ ' } ?Q'$ 
    proof
    fix  $x'$ 
    assume  $x' \in (a \Rightarrow b) \cdot (f \text{ ' } ?Q')$ 
    then obtain  $Q'$  where  $1: x' = (a \Rightarrow b) \cdot f Q'$  and  $2: Q \rightarrow \langle \alpha, Q \rangle$ 
    by auto (metis permute-swap-cancel transition-eqt')
    with  $*$  and  $a$  and  $b$  have  $a \notin \text{supp } (f Q')$  and  $b \notin \text{supp } (f Q')$ 
    by auto
    with  $1$  have  $x' = f Q'$ 
    by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
    with  $2$  show  $x' \in f \text{ ' } ?Q'$ 
    by simp
  }
  qed
  moreover have  $f \text{ ' } ?Q' \subseteq (a \Rightarrow b) \cdot (f \text{ ' } ?Q')$ 
  proof
  fix  $x'$ 
  assume  $x' \in f \text{ ' } ?Q'$ 
  then obtain  $Q'$  where  $1: x' = f Q'$  and  $2: Q \rightarrow \langle \alpha, Q \rangle$ 
  by auto
  with  $*$  and  $a$  and  $b$  have  $a \notin \text{supp } (f Q')$  and  $b \notin \text{supp } (f Q')$ 
  by auto

```

```

    with 1 have  $x' = (a \equiv b) \cdot f Q'$ 
      by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
    with 2 show  $x' \in (a \equiv b) \cdot (f ' ?Q')$ 
      using mem-permute-iff by blast
  qed
  ultimately show  $(a \equiv b) \cdot (f ' ?Q') = f ' ?Q' ..$ 
  qed
  then have supp-image-subset-supp-P':  $\text{supp } (f ' ?Q') \subseteq \text{supp } P'$ 
    by (metis (erased, lifting) finite-supp supp-is-subset)
  then have finite-supp-image:  $\text{finite } (\text{supp } (f ' ?Q'))$ 
    using finite-supp rev-finite-subset by blast
  have  $|f ' ?Q'| \leq o \mid UNIV :: 'state \text{ set}$ 
    by (metis card-of-UNIV card-of-image ordLeq-transitive)
  also have  $\mid UNIV :: 'state \text{ set} \mid < o \mid UNIV :: 'idx \text{ set}$ 
    by (metis card-idx-state)
  also have  $\mid UNIV :: 'idx \text{ set} \mid \leq o \text{ natLeq } + c \mid UNIV :: 'idx \text{ set}$ 
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally have card-image:  $|f ' ?Q'| < o \text{ natLeq } + c \mid UNIV :: 'idx \text{ set} \mid .$ 
  let  $?y = \text{Conj } (\text{Abs-bset } (f ' ?Q')) :: ('idx, 'pred, 'act) \text{ formula}$ 
  have  $P \models \text{Act } \alpha ?y$ 
    unfolding valid-Act proof (standard+)
      show  $P \rightarrow \langle \alpha, P \rangle$  by fact
    next
      {
        fix  $Q'$ 
        assume  $Q \rightarrow \langle \alpha, Q \rangle$ 
        with * have  $P' \models f Q'$ 
          by (metis distinguishing-formula-def mem-Collect-eq)
        }
      then show  $P' \models ?y$ 
        by (simp add: finite-supp-image card-image)
    qed
  moreover have  $\neg Q \models \text{Act } \alpha ?y$ 
  proof
    assume  $Q \models \text{Act } \alpha ?y$ 
    then obtain  $Q'$  where 1:  $Q \rightarrow \langle \alpha, Q \rangle$  and 2:  $Q' \models ?y$ 
      using  $\langle bn \ \alpha \ \#* \ Q \rangle$  by (metis valid-Act-fresh)
    from 2 have  $\bigwedge Q''. Q \rightarrow \langle \alpha, Q'' \rangle \longrightarrow Q' \models f Q''$ 
      by (simp add: finite-supp-image card-image)
    with 1 and * show False
      using distinguishing-formula-def by blast
  qed
  ultimately have False
    by (metis  $\langle P = \cdot Q \rangle$  logically-equivalent-def)
  }
  then show  $?thesis$  by auto
  qed
}
ultimately show  $?thesis$ 

```

```

    unfolding is-bisimulation-def by metis
  qed

  theorem equivalence-implies-bisimilarity: assumes  $P = Q$  shows  $P \sim Q$ 
  using assms by (metis bisimilar-def equivalence-is-bisimulation)

end

end

theory FS-Set
imports
  Nominal2.Nominal2
begin

```

10 Finitely Supported Sets

We define the type of finitely supported sets (over some permutation type $'a$). Note that we cannot more generally define the (sub-)type of finitely supported elements for arbitrary permutation types $'a$: there is no guarantee that this type is non-empty.

```

typedef 'a fs-set = {x::'a::pt set. finite (supp x)}
  by (simp add: exI[where x={}] supp-set-empty)

```

```

setup-lifting type-definition-fs-set

```

Type $'a$ *fs-set* is a finitely supported permutation type.

```

instantiation fs-set :: (pt) pt
begin

```

```

  lift-definition permute-fs-set :: perm  $\Rightarrow$  'a fs-set  $\Rightarrow$  'a fs-set is permute
  by (metis permute-finite supp-eqvt)

```

```

instance
  apply (intro-classes)
  apply (metis (mono-tags) permute-fs-set.rep-eq Rep-fs-set-inverse permute-zero)
  apply (metis (mono-tags) permute-fs-set.rep-eq Rep-fs-set-inverse permute-plus)
  done

```

```

end

```

```

instantiation fs-set :: (pt) fs
begin

```

```

  instance
  proof (intro-classes)
    fix x :: 'a fs-set
    from Rep-fs-set have finite (supp (Rep-fs-set x)) by simp

```

hence *finite* $\{a. \text{infinite } \{b. (a \equiv b) \cdot \text{Rep-fs-set } x \neq \text{Rep-fs-set } x\}\}$ **by** (*unfold supp-def*)
hence *finite* $\{a. \text{infinite } \{b. ((a \equiv b) \cdot x) \neq x\}\}$ **by** *transfer*
thus *finite* (*supp x*) **by** (*fold supp-def*)
qed

end

Set membership.

lift-definition *member-fs-set* :: 'a::pt \Rightarrow 'a fs-set \Rightarrow bool **is** (\in) .

notation

member-fs-set (\in_{fs}) **and**
member-fs-set (\in_{fs} -) [51, 51] 50)

lemma *member-fs-set-permute-iff* [*simp*]: $p \cdot x \in_{fs} p \cdot X \longleftrightarrow x \in_{fs} X$
by *transfer* (*simp add: mem-permute-iff*)

lemma *member-fs-set-eqt* [*eqvt*]: $x \in_{fs} X \Longrightarrow p \cdot x \in_{fs} p \cdot X$
by *simp*

end

theory *FL-Transition-System*

imports

Transition-System FS-Set

begin

11 Nominal Transition Systems with Effects and F/L -Bisimilarity

11.1 Nominal transition systems with effects

The paper defines an effect as a finitely supported function from states to states. It then fixes an equivariant set \mathcal{F} of effects. In our formalization, we avoid working with such a (carrier) set, and instead introduce a type of (finitely supported) effects together with an (equivariant) application operator for effects and states.

Equivariance (of the type of effects) is implicitly guaranteed (by the type of *permute*).

First represents the (finitely supported) set of effects that must be observed before following a transition.

type-synonym 'eff first = 'eff fs-set

Later is a function that represents how the set F (for *first*) changes depending on the action of a transition and the chosen effect.

type-synonym ('a,'eff) later = 'a \times 'eff first \times 'eff \Rightarrow 'eff first

locale *effect-nominal-ts* = *nominal-ts* *satisfies transition*
for *satisfies* :: 'state::fs ⇒ 'pred::fs ⇒ bool (**infix** ⊢ 70)
and *transition* :: 'state ⇒ ('act::bn, 'state) *residual* ⇒ bool (**infix** → 70) +
fixes *effect-apply* :: 'effect::fs ⇒ 'state ⇒ 'state ((-) - [0,101] 100)
and *L* :: ('act, 'effect) *later*
assumes *effect-apply-eqv*: *eqvt effect-apply*
and *L-eqv*: *eqvt L* — *L* is assumed to be equivariant.

begin

lemma *effect-apply-eqv-aux* [*simp*]: $p \cdot \text{effect-apply} = \text{effect-apply}$
by (*metis effect-apply-eqv eqvt-def*)

lemma *effect-apply-eqv'* [*eqvt*]: $p \cdot \langle f \rangle P = \langle p \cdot f \rangle (p \cdot P)$
by *simp*

lemma *L-eqv-aux* [*simp*]: $p \cdot L = L$
by (*metis L-eqv eqvt-def*)

lemma *L-eqv'* [*eqvt*]: $p \cdot L (\alpha, P, f) = L (p \cdot \alpha, p \cdot P, p \cdot f)$
by *simp*

end

11.2 *L*-bisimulations and *F/L*-bisimilarity

context *effect-nominal-ts*

begin

definition *is-L-bisimulation*:: ('effect first ⇒ 'state ⇒ 'state ⇒ bool) ⇒ bool
where

$$\begin{aligned} \textit{is-L-bisimulation } R \equiv & \\ & \forall F. \textit{symp } (R F) \wedge \\ & (\forall P Q. R F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))) \wedge \\ & (\forall P Q. R F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \textit{bn } \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow \\ & \quad \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge R (L (\alpha, F, f)) P' Q')))) \end{aligned}$$

definition *FL-bisimilar* :: 'effect first ⇒ 'state ⇒ 'state ⇒ bool **where**
FL-bisimilar *F P Q* ≡ ∃ *R*. *is-L-bisimulation* *R* ∧ (*R F*) *P Q*

abbreviation *FL-bisimilar'* (- ~[-] - [51,0,51] 50) **where**
 $P \sim.[F] Q \equiv \textit{FL-bisimilar } F P Q$

FL-bisimilar is an equivariant relation, and (for every *F*) an equivalence.

lemma *is-L-bisimulation-eqv* [*eqvt*]:
assumes *is-L-bisimulation* *R* **shows** *is-L-bisimulation* ($p \cdot R$)
unfolding *is-L-bisimulation-def*
proof (*clarify*)

```

fix F
have symp ((p · R) F) (is ?S)
using assms unfolding is-L-bisimulation-def by (metis eqvt-lambda symp-eqvt)
moreover have  $\forall P Q. (p \cdot R) F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))$  (is ?T)
proof (clarify)
  fix P Q f  $\varphi$ 
  assume pR: (p · R) F P Q and effect:  $f \in_{fs} F$  and satisfies:  $\langle f \rangle P \vdash \varphi$ 
  from pR have R (-p · F) (-p · P) (-p · Q)
    by (simp add: eqvt-lambda permute-bool-def unpermute-def)
  moreover have (-p · f)  $\in_{fs}$  (-p · F)
    using effect by simp
  moreover have  $\langle -p \cdot f \rangle (-p \cdot P) \vdash -p \cdot \varphi$ 
    using satisfies by (metis effect-apply-eqvt' satisfies-eqvt)
  ultimately have  $\langle -p \cdot f \rangle (-p \cdot Q) \vdash -p \cdot \varphi$ 
    using assms unfolding is-L-bisimulation-def by auto
  then show  $\langle f \rangle Q \vdash \varphi$ 
  by (metis (full-types) effect-apply-eqvt' permute-minus-cancel(1) satisfies-eqvt)
qed
moreover have  $\forall P Q. (p \cdot R) F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge (p \cdot R) (L (\alpha, F, f)) P' Q')))$  (is ?U)
proof (clarify)
  fix P Q f  $\alpha P'$ 
  assume pR: (p · R) F P Q and effect:  $f \in_{fs} F$  and fresh:  $\text{bn } \alpha \#* (\langle f \rangle Q, F, f)$  and trans:  $\langle f \rangle P \rightarrow \langle \alpha, P' \rangle$ 
  from pR have R (-p · F) (-p · P) (-p · Q)
    by (simp add: eqvt-lambda permute-bool-def unpermute-def)
  moreover have (-p · f)  $\in_{fs}$  (-p · F)
    using effect by simp
  moreover have  $\text{bn } (-p \cdot \alpha) \#* (\langle -p \cdot f \rangle (-p \cdot Q), -p \cdot F, -p \cdot f)$ 
    using fresh by (metis (full-types) effect-apply-eqvt' bn-eqvt fresh-star-Pair fresh-star-permute-iff)
  moreover have  $\langle -p \cdot f \rangle (-p \cdot P) \rightarrow \langle -p \cdot \alpha, -p \cdot P' \rangle$ 
    using trans by (metis effect-apply-eqvt' transition-eqvt')
  ultimately obtain Q' where T:  $\langle -p \cdot f \rangle (-p \cdot Q) \rightarrow \langle -p \cdot \alpha, Q' \rangle$  and R:  $R (L (-p \cdot \alpha, -p \cdot F, -p \cdot f)) (-p \cdot P') Q'$ 
    using assms unfolding is-L-bisimulation-def by meson
  from T have  $\langle f \rangle Q \rightarrow \langle \alpha, p \cdot Q' \rangle$ 
    by (metis (no-types, lifting) effect-apply-eqvt' abs-residual-pair-eqvt permute-minus-cancel(1) transition-eqvt)
  moreover from R have (p · R) (p · L (-p ·  $\alpha$ , -p · F, -p · f)) (p · -p · P') (p · Q')
    by (metis permute-boolI permute-fun-def permute-minus-cancel(2))
  then have (p · R) (L ( $\alpha, F, f$ )) P' (p · Q')
    by (simp add: permute-self)
  ultimately show  $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge (p \cdot R) (L (\alpha, F, f)) P' Q'$ 
    by metis

```

qed
ultimately show $?S \wedge ?T \wedge ?U$ by *simp*
qed

lemma *FL-bisimilar-eqt*:
assumes $P \sim.[F] Q$ shows $(p \cdot P) \sim.[p \cdot F] (p \cdot Q)$
using *assms*
by (*metis eqt-apply permute-boolI is-L-bisimulation-eqt FL-bisimilar-def*)

lemma *FL-bisimilar-reflp*: *reflp* (*FL-bisimilar* F)
proof (*rule reflpI*)
fix x
have *is-L-bisimulation* $(\lambda \cdot. (=))$
unfolding *is-L-bisimulation-def* by (*simp add: symp-def*)
then show $x \sim.[F] x$
unfolding *FL-bisimilar-def* by *auto*
qed

lemma *FL-bisimilar-symp*: *symp* (*FL-bisimilar* F)
proof (*rule sympI*)
fix $P Q$
assume $P \sim.[F] Q$
then obtain R where $*$: *is-L-bisimulation* $R \wedge R F P Q$
unfolding *FL-bisimilar-def* ..
then have $R F Q P$
unfolding *is-L-bisimulation-def* by (*simp add: symp-def*)
with $*$ show $Q \sim.[F] P$
unfolding *FL-bisimilar-def* by *auto*
qed

lemma *FL-bisimilar-is-L-bisimulation*: *is-L-bisimulation* *FL-bisimilar*
unfolding *is-L-bisimulation-def* **proof**
fix F
have *symp* (*FL-bisimilar* F) (**is** $?R$)
by (*fact FL-bisimilar-symp*)
moreover have $\forall P Q. P \sim.[F] Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))$ (**is** $?S$)
by (*auto simp add: is-L-bisimulation-def FL-bisimilar-def*)
moreover have $\forall P Q. P \sim.[F] Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge P' \sim.[L(\alpha, F, f)] Q'))$ (**is** $?T$)
by (*auto simp add: is-L-bisimulation-def FL-bisimilar-def*) *blast*
ultimately show $?R \wedge ?S \wedge ?T$
by *metis*
qed

lemma *FL-bisimilar-simulation-step*:
assumes $P \sim.[F] Q$ and $f \in_{fs} F$ and $\text{bn } \alpha \#* (\langle f \rangle Q, F, f)$ and $\langle f \rangle P \rightarrow$

$\langle \alpha, P \rangle$
obtains Q' where $\langle f \rangle Q \rightarrow \langle \alpha, Q \rangle$ and $P' \sim [L(\alpha, F, f)] Q'$
using *assms* by (metis (poly-guards-query) FL-bisimilar-is-L-bisimulation is-L-bisimulation-def)

lemma *FL-bisimilar-transp: transp (FL-bisimilar F)*
proof (rule transpI)
fix $P Q R$
assume $PQ: P \sim [F] Q$ and $QR: Q \sim [F] R$
let $?FL\text{-bisim} = \lambda F. (FL\text{-bisimilar } F) OO (FL\text{-bisimilar } F)$
have $\bigwedge F. \text{symp } (?FL\text{-bisim } F)$
proof (rule sympI)
fix $F P R$
assume $?FL\text{-bisim } F P R$
then obtain Q where $P \sim [F] Q$ and $Q \sim [F] R$
by *blast*
then have $R \sim [F] Q$ and $Q \sim [F] P$
by (metis FL-bisimilar-symp sympE)+
then show $?FL\text{-bisim } F R P$
by *blast*
qed

moreover have $\bigwedge F. \forall P Q. ?FL\text{-bisim } F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))$
using *FL-bisimilar-is-L-bisimulation is-L-bisimulation-def* **by** *auto*
moreover have $\bigwedge F. \forall P Q. ?FL\text{-bisim } F P Q \longrightarrow$
 $(\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow$
 $\langle f \rangle P \rightarrow \langle \alpha, P \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q \rangle \wedge ?FL\text{-bisim } (L(\alpha, F, f))$
 $P' Q'))$
proof (clarify)
fix $F P R Q f \alpha P'$
assume $PR: P \sim [F] R$ and $RQ: R \sim [F] Q$ and *effect: $f \in_{fs} F$* and
fresh: $\text{bn } \alpha \# (\langle f \rangle Q, F, f)$* and *trans: $\langle f \rangle P \rightarrow \langle \alpha, P \rangle$*
— rename $\langle \alpha, P \rangle$ to avoid $(\langle f \rangle R, F)$, without touching $(\langle f \rangle Q, F, f)$
obtain p where 1: $(p \cdot \text{bn } \alpha) \#* (\langle f \rangle R, F, f)$ and 2: *supp* $(\langle \alpha, P \rangle, (\langle f \rangle Q, F, f)) \#* p$
proof (rule at-set-avoiding2[of $\text{bn } \alpha (\langle f \rangle R, F, f) (\langle \alpha, P \rangle, (\langle f \rangle Q, F, f))$, THEN *exE*])
show *finite* $(\text{bn } \alpha)$ **by** (fact *bn-finite*)
next
show *finite* $(\text{supp } (\langle f \rangle R, F, f))$ **by** (fact *finite-supp*)
next
show *finite* $(\text{supp } (\langle \alpha, P \rangle, (\langle f \rangle Q, F, f)))$ **by** (simp add: *finite-supp supp-Pair*)
next
show $\text{bn } \alpha \#* (\langle \alpha, P \rangle, (\langle f \rangle Q, F, f))$
using *bn-abs-residual-fresh fresh fresh-star-Pair* **by** *blast*
qed *metis*
from 2 **have** 3: *supp* $\langle \alpha, P \rangle \#* p$ and 4: *supp* $(\langle f \rangle Q, F, f) \#* p$
by (simp add: *fresh-star-Un supp-Pair*)+
from 3 **have** $\langle p \cdot \alpha, p \cdot P \rangle = \langle \alpha, P \rangle$

using *supp-perm-eq* **by** *fastforce*
then obtain pR' **where** 5: $\langle f \rangle R \rightarrow \langle p \cdot \alpha, pR' \rangle$ **and** 6: $(p \cdot P') \sim [L (p \cdot \alpha, F, f)] pR'$
using *PR effect trans 1* **by** (*metis FL-bisimilar-simulation-step bn-eqvt*)
from *fresh* **and** 4 **have** $bn (p \cdot \alpha) \#* (\langle f \rangle Q, F, f)$
by (*metis bn-eqvt fresh-star-permute-iff supp-perm-eq*)
then obtain pQ' **where** 7: $\langle f \rangle Q \rightarrow \langle p \cdot \alpha, pQ' \rangle$ **and** 8: $pR' \sim [L (p \cdot \alpha, F, f)] pQ'$
using *RQ effect 5* **by** (*metis FL-bisimilar-simulation-step*)
from 4 **have** $supp (\langle f \rangle Q) \#* p$
by (*simp add: fresh-star-Un supp-Pair*)
with 7 **have** $\langle f \rangle Q \rightarrow \langle \alpha, -p \cdot pQ' \rangle$
by (*metis permute-minus-cancel(2) supp-perm-eq transition-eqvt'*)
moreover from 6 **and** 8 **have** $?FL-bisim (L (p \cdot \alpha, F, f)) (p \cdot P') pQ'$
by (*metis relcompp.relcompI*)
then have $?FL-bisim (-p \cdot L (p \cdot \alpha, F, f)) (-p \cdot p \cdot P') (-p \cdot pQ')$
using *FL-bisimilar-eqvt* **by** *blast*
then have $?FL-bisim (L (\alpha, -p \cdot F, -p \cdot f)) P' (-p \cdot pQ')$
by (*simp add: L-eqvt'*)
then have $?FL-bisim (L (\alpha, F, f)) P' (-p \cdot pQ')$
using 4 **by** (*metis fresh-star-Un permute-minus-cancel(2) supp-Pair supp-perm-eq*)
ultimately show $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge ?FL-bisim (L (\alpha, F, f)) P' Q'$
by *metis*
qed
ultimately have *is-L-bisimulation* $?FL-bisim$
unfolding *is-L-bisimulation-def* **by** *metis*
moreover have $?FL-bisim F P R$
using *PQ QR* **by** *blast*
ultimately show $P \sim [F] R$
unfolding *FL-bisimilar-def* **by** *meson*
qed

lemma *FL-bisimilar-equivp: equivp (FL-bisimilar F)*
by (*metis FL-bisimilar-reflp FL-bisimilar-symp FL-bisimilar-transp equivp-reflp-symp-transp*)

end

end

theory *FL-Formula*

imports

Nominal-Bounded-Set

Nominal-Wellfounded

Residual

FL-Transition-System

begin

12 Infinitary Formulas With Effects

12.1 Infinitely branching trees

First, we define a type of trees, with a constructor $tConj$ that maps (potentially infinite) sets of trees into trees. To avoid paradoxes (note that there is no injection from the powerset of trees into the set of trees), the cardinality of the argument set must be bounded.

The effect consequence operator $\langle f \rangle$ is always and only used as a prefix to a predicate or an action formula. So to simplify the representation of formula trees with effects, the effect operator is merged into the predicate or action it precedes.

```
datatype ('idx,'pred,'act,'eff) Tree =
  tConj ('idx,'pred,'act,'eff) Tree set['idx] — potentially infinite sets of trees
| tNot ('idx,'pred,'act,'eff) Tree
| tPred 'eff 'pred
| tAct 'eff 'act ('idx,'pred,'act,'eff) Tree
```

The (automatically generated) induction principle for trees allows us to prove that the following relation over trees is well-founded. This will be useful for termination proofs when we define functions by recursion over trees.

```
inductive-set Tree-wf :: ('idx,'pred,'act,'eff) Tree rel where
  t ∈ set-bset tset ⇒ (t, tConj tset) ∈ Tree-wf
| (t, tNot t) ∈ Tree-wf
| (t, tAct f α t) ∈ Tree-wf
```

lemma wf-Tree-wf: wf Tree-wf

unfolding wf-def

proof (rule allI, rule impI, rule allI)

fix P :: ('idx,'pred,'act,'eff) Tree ⇒ bool **and** t

assume $\forall x. (\forall y. (y, x) \in \text{Tree-wf} \longrightarrow P y) \longrightarrow P x$

then show P t

proof (induction t)

case tConj **then show** ?case

by (metis Tree.distinct(2) Tree.distinct(5) Tree.inject(1) Tree-wf.cases)

next

case tNot **then show** ?case

by (metis Tree.distinct(1) Tree.distinct(9) Tree.inject(2) Tree-wf.cases)

next

case tPred **then show** ?case

by (metis Tree.distinct(11) Tree.distinct(3) Tree.distinct(7) Tree-wf.cases)

next

case tAct **then show** ?case

by (metis Tree.distinct(10) Tree.distinct(6) Tree.inject(4) Tree-wf.cases)

qed

qed

We define a permutation operation on the type of trees.

```

instantiation Tree :: (type, pt, pt, pt) pt
begin

  primrec permute-Tree :: perm  $\Rightarrow$  (-,-,-,-) Tree  $\Rightarrow$  (-,-,-,-) Tree where
    p  $\cdot$  (tConj tset) = tConj (map-bset (permute p) tset) — neat trick to get
around the fact that tset is not of permutation type yet
  | p  $\cdot$  (tNot t) = tNot (p  $\cdot$  t)
  | p  $\cdot$  (tPred f  $\varphi$ ) = tPred (p  $\cdot$  f) (p  $\cdot$   $\varphi$ )
  | p  $\cdot$  (tAct f  $\alpha$  t) = tAct (p  $\cdot$  f) (p  $\cdot$   $\alpha$ ) (p  $\cdot$  t)

  instance
  proof
    fix t :: (-,-,-,-) Tree
    show 0  $\cdot$  t = t
    proof (induction t)
      case tConj then show ?case
        by (simp, transfer) (auto simp: image-def)
    qed simp-all
  next
    fix p q :: perm and t :: (-,-,-,-) Tree
    show (p + q)  $\cdot$  t = p  $\cdot$  q  $\cdot$  t
    proof (induction t)
      case tConj then show ?case
        by (simp, transfer) (auto simp: image-def)
    qed simp-all
  qed

```

Now that the type of trees—and hence the type of (bounded) sets of trees—is a permutation type, we can massage the definition of $p \cdot tConj\ tset$ into its more usual form.

```

lemma permute-Tree-tConj [simp]:  $p \cdot tConj\ tset = tConj\ (p \cdot tset)$ 
by (simp add: map-bset-permute)

```

```

declare permute-Tree.simps(1) [simp del]

```

The relation *Tree-wf* is equivariant.

```

lemma Tree-wf-eqvt-aux:
  assumes (t1, t2)  $\in$  Tree-wf shows (p  $\cdot$  t1, p  $\cdot$  t2)  $\in$  Tree-wf
using assms proof (induction rule: Tree-wf.induct)
  fix t :: ('a,'b,'c,'d) Tree and tset :: ('a,'b,'c,'d) Tree set['a]
  assume t  $\in$  set-bset tset then show (p  $\cdot$  t, p  $\cdot$  tConj tset)  $\in$  Tree-wf
    by (metis Tree-wf.intros(1) mem-permute-iff permute-Tree-tConj set-bset-eqvt)
next
  fix t :: ('a,'b,'c,'d) Tree
  show (p  $\cdot$  t, p  $\cdot$  tNot t)  $\in$  Tree-wf

```

```

  by (metis Tree-wf.intros(2) permute-Tree.simps(2))
next
fix t :: ('a,'b,'c,'d) Tree and f and  $\alpha$ 
show (p · t, p · tAct f  $\alpha$  t) ∈ Tree-wf
  by (metis Tree-wf.intros(3) permute-Tree.simps(4))
qed

```

```

lemma Tree-wf-eqt [eqvt, simp]: p · Tree-wf = Tree-wf
proof
  show p · Tree-wf ⊆ Tree-wf
    by (auto simp add: permute-set-def) (rule Tree-wf-eqt-aux)
next
  show Tree-wf ⊆ p · Tree-wf
    by (auto simp add: permute-set-def) (metis Tree-wf-eqt-aux permute-minus-cancel(1))
qed

```

```

lemma Tree-wf-eqt': eqvt Tree-wf
by (metis Tree-wf-eqt eqvtI)

```

The definition of *permute* for trees gives rise to the usual notion of support. The following lemmas, one for each constructor, describe the support of trees.

```

lemma supp-tConj [simp]: supp (tConj tset) = supp tset
unfolding supp-def by simp

```

```

lemma supp-tNot [simp]: supp (tNot t) = supp t
unfolding supp-def by simp

```

```

lemma supp-tPred [simp]: supp (tPred f  $\varphi$ ) = supp f ∪ supp  $\varphi$ 
unfolding supp-def by (simp add: Collect-imp-eq Collect-neg-eq)

```

```

lemma supp-tAct [simp]: supp (tAct f  $\alpha$  t) = supp f ∪ supp  $\alpha$  ∪ supp t
unfolding supp-def by (auto simp add: Collect-imp-eq Collect-neg-eq)

```

12.2 Trees modulo α -equivalence

We generalize the notion of support, which considers whether a permuted element is *equal* to itself, to arbitrary endorelations. This is available as *supp-rel* in Nominal Isabelle.

```

lemma supp-rel-eqt [eqvt]:
  p · supp-rel R x = supp-rel (p · R) (p · x)
by (simp add: supp-rel-def)

```

Usually, the definition of α -equivalence presupposes a notion of free variables. However, the variables that are “free” in an infinitary conjunction are not necessarily those that are free in one of the conjuncts. For instance, consider a conjunction over *all* names. Applying any permutation will yield the same conjunction, i.e., this conjunction has *no* free variables.

To obtain the correct notion of free variables for infinitary conjunctions, we initially defined α -equivalence and free variables via mutual recursion. In particular, we defined the free variables of a conjunction as term $fv\text{-Tree}(tConj\ tset) = supp\text{-rel}\ \alpha\text{-Tree}(tConj\ tset)$.

We then realized that it is not necessary to define the concept of “free variables” at all, but the definition of α -equivalence becomes much simpler (in particular, it is no longer mutually recursive) if we directly use the support modulo α -equivalence.

The following lemmas and constructions are used to prove termination of our definition.

lemma *supp-rel-cong* [*fundef-cong*]:

$\llbracket x=x'; \bigwedge a\ b.\ R\ ((a \equiv b) \cdot x')\ x' \longleftrightarrow R'\ ((a \equiv b) \cdot x')\ x' \rrbracket \implies supp\text{-rel}\ R\ x = supp\text{-rel}\ R'\ x'$

by (*simp add: supp-rel-def*)

lemma *rel-bset-cong* [*fundef-cong*]:

$\llbracket x=x'; y=y'; \bigwedge a\ b.\ a \in set\text{-bset}\ x' \implies b \in set\text{-bset}\ y' \implies R\ a\ b \longleftrightarrow R'\ a\ b \rrbracket \implies rel\text{-bset}\ R\ x\ y \longleftrightarrow rel\text{-bset}\ R'\ x'\ y'$

by (*simp add: rel-bset-def rel-set-def*)

lemma *alpha-set-cong* [*fundef-cong*]:

$\llbracket bs=bs'; x=x'; R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y'; f\ x' = f'\ x'; f\ y' = f'\ y'; p=p'; cs=cs'; y=y' \rrbracket \implies$

$\alpha\text{-set}\ (bs, x)\ R\ f\ p\ (cs, y) \longleftrightarrow \alpha\text{-set}\ (bs', x')\ R'\ f'\ p'\ (cs', y')$

by (*simp add: alpha-set*)

quotient-type

$(\text{'id}\ x, \text{'pred}, \text{'act}, \text{'eff})\ Tree_p = (\text{'id}\ x, \text{'pred}::pt, \text{'act}::bn, \text{'eff}::fs)\ Tree / hull\text{-rel}\ p$

by (*fact hull-relp-equivp*)

lemma *abs-Tree_p-eq* [*simp*]: $abs\text{-Tree}_p\ (p \cdot t) = abs\text{-Tree}_p\ t$

by (*metis hull-relp.simps Tree_p.abs-eq-iff*)

lemma *permute-rep-abs-Tree_p*:

obtains p **where** $rep\text{-Tree}_p\ (abs\text{-Tree}_p\ t) = p \cdot t$

by (*metis Quotient3-Tree_p Tree_p.abs-eq-iff rep-abs-rsp hull-relp.simps*)

lift-definition $Tree\text{-wf}_p :: (\text{'id}\ x, \text{'pred}::pt, \text{'act}::bn, \text{'eff}::fs)\ Tree_p\ rel\ is$

$Tree\text{-wf} .$

lemma *Tree-wf_pI* [*simp*]:

assumes $(a, b) \in Tree\text{-wf}$

shows $(abs\text{-Tree}_p\ (p \cdot a), abs\text{-Tree}_p\ b) \in Tree\text{-wf}_p$

using *assms* **by** (*metis (erased, lifting) Tree_p.abs-eq-iff Tree-wf_p.abs-eq hull-relp.intros map-prod-simp rev-image-eqI*)

lemma *Tree-wf_p-trivialI* [*simp*]:

assumes $(a, b) \in \text{Tree-wf}$
shows $(\text{abs-Tree}_p a, \text{abs-Tree}_p b) \in \text{Tree-wf}_p$
using *assms* **by** $(\text{metis } \text{Tree-wf}_p I \text{ permute-zero})$

lemma *Tree-wf_pE*:
assumes $(a_p, b_p) \in \text{Tree-wf}_p$
obtains $a b$ **where** $a_p = \text{abs-Tree}_p a$ **and** $b_p = \text{abs-Tree}_p b$ **and** $(a, b) \in \text{Tree-wf}$
using *assms* **by** $(\text{metis } (\text{erased}, \text{lifting}) \text{Pair-inject } \text{Tree-wf}_p.\text{abs-eq } \text{prod-fun-imageE})$

lemma *wf-Tree-wf_p*: $\text{wf } \text{Tree-wf}_p$
apply $(\text{rule } \text{wf-subset}[\text{of } \text{inv-image } (\text{hull-rel } O \text{Tree-wf}) \text{rep-Tree}_p])$
apply $(\text{metis } \text{Tree-wf-eqvt}' \text{wf-Tree-wf } \text{wf-hull-rel-relcomp } \text{wf-inv-image})$
apply $(\text{auto } \text{elim}!: \text{Tree-wf}_p E)$
apply $(\text{rename-tac } t1 \ t2)$
apply $(\text{rule-tac } t=t1 \ \text{in } \text{permute-rep-abs-Tree}_p)$
apply $(\text{rule-tac } t=t2 \ \text{in } \text{permute-rep-abs-Tree}_p)$
apply $(\text{rename-tac } p1 \ p2)$
apply $(\text{subgoal-tac } (p2 \cdot t1, p2 \cdot t2) \in \text{Tree-wf})$
apply $(\text{subgoal-tac } (p1 \cdot t1, p2 \cdot t1) \in \text{hull-rel})$
apply $(\text{metis } \text{relcomp.relcompI})$
apply $(\text{metis } \text{hull-rel.simps } \text{permute-minus-cancel}(2) \ \text{permute-plus})$
apply $(\text{metis } \text{Tree-wf-eqvt-aux})$
done

fun *alpha-Tree-termination* :: $('a, 'b, 'c, 'd) \text{Tree} \times ('a, 'b, 'c, 'd) \text{Tree} \Rightarrow ('a, 'b::pt, 'c::bn, 'd::fs) \text{Tree}_p \text{ set}$ **where**
 $\text{alpha-Tree-termination } (t1, t2) = \{\text{abs-Tree}_p t1, \text{abs-Tree}_p t2\}$

Here it comes ...

function *(sequential)*
 $\text{alpha-Tree} :: ('idx, 'pred::pt, 'act::bn, 'eff::fs) \text{Tree} \Rightarrow ('idx, 'pred, 'act, 'eff) \text{Tree} \Rightarrow \text{bool}$ **(infix** $=_\alpha$ **50)** **where**
 $- (=_\alpha)$
 $\text{alpha-tConj}: \text{tConj } tset1 =_\alpha \text{tConj } tset2 \iff \text{rel-bset } \text{alpha-Tree } tset1 \ tset2$
 $| \text{alpha-tNot}: \text{tNot } t1 =_\alpha \text{tNot } t2 \iff t1 =_\alpha t2$
 $| \text{alpha-tPred}: \text{tPred } f1 \ \varphi1 =_\alpha \text{tPred } f2 \ \varphi2 \iff f1 = f2 \wedge \varphi1 = \varphi2$
 $-$ the action may have binding names
 $| \text{alpha-tAct}: \text{tAct } f1 \ \alpha1 \ t1 =_\alpha \text{tAct } f2 \ \alpha2 \ t2 \iff$
 $f1 = f2 \wedge (\exists p. (bn \ \alpha1, t1) \approx_{\text{set}} \text{alpha-Tree } (\text{supp-rel } \text{alpha-Tree}) \ p \ (bn \ \alpha2, t2) \wedge (bn \ \alpha1, \alpha1) \approx_{\text{set}} ((=)) \ \text{supp } p \ (bn \ \alpha2, \alpha2))$
 $| \text{alpha-other}: - =_\alpha - \iff \text{False}$
 $-$ 254 subgoals (!)
by *pat-completeness auto*
termination
proof
let $?R = \text{inv-image } (\text{max-ext } \text{Tree-wf}_p) \ \text{alpha-Tree-termination}$
show $\text{wf } ?R$
by $(\text{metis } \text{max-ext-wf } \text{wf-Tree-wf}_p \ \text{wf-inv-image})$

qed (*auto simp add: max-ext.simps Tree-wf.simps simp del: permute-Tree-tConj*)

We provide more descriptive case names for the automatically generated induction principle, and specialize it to an induction rule for α -equivalence.

lemmas *alpha-Tree-induct'* = *alpha-Tree.induct*[*case-names alpha-tConj alpha-tNot alpha-tPred alpha-tAct alpha-other(1) alpha-other(2) alpha-other(3) alpha-other(4) alpha-other(5) alpha-other(6) alpha-other(7) alpha-other(8) alpha-other(9) alpha-other(10) alpha-other(11) alpha-other(12) alpha-other(13) alpha-other(14) alpha-other(15) alpha-other(16) alpha-other(17) alpha-other(18)*]

lemma *alpha-Tree-induct*[*case-names tConj tNot tPred tAct, consumes 1*]:

assumes $t1 =_{\alpha} t2$
and $\bigwedge tset1\ tset2. (\bigwedge a\ b. a \in \text{set-bset } tset1 \implies b \in \text{set-bset } tset2 \implies a =_{\alpha} b \implies P\ a\ b) \implies$
 $\text{rel-bset } (=_{\alpha})\ tset1\ tset2 \implies P\ (tConj\ tset1)\ (tConj\ tset2)$
and $\bigwedge t1\ t2. t1 =_{\alpha} t2 \implies P\ t1\ t2 \implies P\ (tNot\ t1)\ (tNot\ t2)$
and $\bigwedge f\ \varphi. P\ (tPred\ f\ \varphi)\ (tPred\ f\ \varphi)$
and $\bigwedge f1\ \alpha1\ t1\ f2\ \alpha2\ t2. (\bigwedge p. p \cdot t1 =_{\alpha} t2 \implies P\ (p \cdot t1)\ t2) \implies f1 = f2 \implies$
 $(\exists p. (bn\ \alpha1, t1) \approx_{\text{set}} (=_{\alpha})\ (\text{supp-rel } (=_{\alpha}))\ p\ (bn\ \alpha2, t2) \wedge (bn\ \alpha1, \alpha1) \approx_{\text{set}} (=)\ \text{supp } p\ (bn\ \alpha2, \alpha2)) \implies$
 $P\ (tAct\ f1\ \alpha1\ t1)\ (tAct\ f2\ \alpha2\ t2)$
shows $P\ t1\ t2$
using *assms by (induction t1 t2 rule: alpha-Tree.induct) simp-all*

α -equivalence is equivariant.

lemma *alpha-Tree-eqt-aux*:

assumes $\bigwedge a\ b. (a \equiv b) \cdot t =_{\alpha} t \iff p \cdot (a \equiv b) \cdot t =_{\alpha} p \cdot t$
shows $p \cdot \text{supp-rel } (=_{\alpha})\ t = \text{supp-rel } (=_{\alpha})\ (p \cdot t)$

proof –

{
 fix a
 let $?B = \{b. \neg ((a \equiv b) \cdot t) =_{\alpha} t\}$
 let $?pB = \{b. \neg ((p \cdot a \equiv b) \cdot p \cdot t) =_{\alpha} (p \cdot t)\}$
 {
 assume *finite ?B*
 moreover have *inj-on (unpermute p) ?pB*
 by (*simp add: inj-on-def unpermute-def*)
 moreover have *unpermute p ' ?pB \subseteq ?B*
 using *assms by auto (metis (erased, lifting) eqt-bound permute-eqt swap-eqt)*
 ultimately have *finite ?pB*
 by (*metis inj-on-finite*)
 }
 moreover
 {
 assume *finite ?pB*
 moreover have *inj-on (permute p) ?B*
 by (*simp add: inj-on-def*)
 moreover have *permute p ' ?B \subseteq ?pB*
 }
}


```

    using assms by auto (metis (erased, lifting) permute-eqvt swap-eqvt)
  ultimately have finite ?B
    by (metis inj-on-finite)
}
ultimately have infinite ?B  $\longleftrightarrow$  infinite ?pB
  by auto
}
then show ?thesis
  by (auto simp add: supp-rel-def permute-set-def) (metis eqvt-bound)
qed

lemma alpha-Tree-eqvt':  $t1 =_{\alpha} t2 \longleftrightarrow p \cdot t1 =_{\alpha} p \cdot t2$ 
proof (induction t1 t2 rule: alpha-Tree-induct')
  case (alpha-tConj tset1 tset2) show ?case
  proof
    assume *:  $tConj\ tset1 =_{\alpha} tConj\ tset2$ 
    {
      fix x
      assume  $x \in set-bset\ (p \cdot tset1)$ 
      then obtain x' where 1:  $x' \in set-bset\ tset1$  and 2:  $x = p \cdot x'$ 
        by (metis imageE permute-bset.rep-eq permute-set-eq-image)
      from 1 obtain y' where 3:  $y' \in set-bset\ tset2$  and 4:  $x' =_{\alpha} y'$ 
        using * by (metis (mono-tags, lifting) FL-Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
      from 3 have  $p \cdot y' \in set-bset\ (p \cdot tset2)$ 
        by (metis mem-permute-iff set-bset-eqvt)
      moreover from 1 and 2 and 3 and 4 have  $x =_{\alpha} p \cdot y'$ 
        using alpha-tConj.IH by blast
      ultimately have  $\exists y \in set-bset\ (p \cdot tset2). x =_{\alpha} y ..$ 
    }
    moreover
    {
      fix y
      assume  $y \in set-bset\ (p \cdot tset2)$ 
      then obtain y' where 1:  $y' \in set-bset\ tset2$  and 2:  $p \cdot y' = y$ 
        by (metis imageE permute-bset.rep-eq permute-set-eq-image)
      from 1 obtain x' where 3:  $x' \in set-bset\ tset1$  and 4:  $x' =_{\alpha} y'$ 
        using * by (metis (mono-tags, lifting) FL-Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
      from 3 have  $p \cdot x' \in set-bset\ (p \cdot tset1)$ 
        by (metis mem-permute-iff set-bset-eqvt)
      moreover from 1 and 2 and 3 and 4 have  $p \cdot x' =_{\alpha} y$ 
        using alpha-tConj.IH by blast
      ultimately have  $\exists x \in set-bset\ (p \cdot tset1). x =_{\alpha} y ..$ 
    }
  }
  ultimately show  $p \cdot tConj\ tset1 =_{\alpha} p \cdot tConj\ tset2$ 
    by (simp add: rel-bset-def rel-set-def)
next
  assume *:  $p \cdot tConj\ tset1 =_{\alpha} p \cdot tConj\ tset2$ 

```

```

{
  fix x
  assume 1: x ∈ set-bset tset1
  then have p · x ∈ set-bset (p · tset1)
    by (metis mem-permute-iff set-bset-eqvt)
  then obtain y' where 2: y' ∈ set-bset (p · tset2) and 3: p · x =α y'
  using * by (metis FL-Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq
rel-set-def)
  from 2 obtain y where 4: y ∈ set-bset tset2 and 5: y' = p · y
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  from 1 and 3 and 4 and 5 have x =α y
    using alpha-tConj.IH by blast
  with 4 have ∃ y ∈ set-bset tset2. x =α y ..
}
moreover
{
  fix y
  assume 1: y ∈ set-bset tset2
  then have p · y ∈ set-bset (p · tset2)
    by (metis mem-permute-iff set-bset-eqvt)
  then obtain x' where 2: x' ∈ set-bset (p · tset1) and 3: x' =α p · y
  using * by (metis FL-Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq
rel-set-def)
  from 2 obtain x where 4: x ∈ set-bset tset1 and 5: p · x = x'
    by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  from 1 and 3 and 4 and 5 have x =α y
    using alpha-tConj.IH by blast
  with 4 have ∃ x ∈ set-bset tset1. x =α y ..
}
ultimately show tConj tset1 =α tConj tset2
  by (simp add: rel-bset-def rel-set-def)
qed
next
case (alpha-tAct f1 α1 t1 f2 α2 t2)
from alpha-tAct.IH(2) have t1: p · supp-rel (=α) t1 = supp-rel (=α) (p · t1)
  by (rule alpha-Tree-eqvt-aux)
from alpha-tAct.IH(3) have t2: p · supp-rel (=α) t2 = supp-rel (=α) (p · t2)
  by (rule alpha-Tree-eqvt-aux)
show ?case
proof
  assume tAct f1 α1 t1 =α tAct f2 α2 t2
  then obtain q where 0: f1 = f2 and 1: (bn α1, t1) ≈set (=α) (supp-rel
(=α)) q (bn α2, t2) and 2: (bn α1, α1) ≈set (=) supp q (bn α2, α2)
  by auto
  from 1 and t1 and t2 have supp-rel (=α) (p · t1) - bn (p · α1) = supp-rel
(=α) (p · t2) - bn (p · α2)
  by (metis Diff-eqvt alpha-set bn-eqvt)
  moreover from 1 and t1 have (supp-rel (=α) (p · t1) - bn (p · α1)) #* (p
+ q - p)

```

by (*metis Diff-eqvt alpha-set bn-eqvt fresh-star-permute-iff permute-perm-def*)
moreover from 1 and alpha-tAct.IH(1) have $p \cdot q \cdot t1 =_{\alpha} p \cdot t2$
 by (*simp add: alpha-set*)
moreover from 2 have $p \cdot q \cdot -p \cdot bn (p \cdot \alpha1) = bn (p \cdot \alpha2)$
 by (*simp add: alpha-set bn-eqvt*)
ultimately have $(bn (p \cdot \alpha1), p \cdot t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (p + q - p) (bn (p \cdot \alpha2), p \cdot t2)$
 by (*simp add: alpha-set*)
moreover from 2 have $(bn (p \cdot \alpha1), p \cdot \alpha1) \approx_{set} (=) supp (p + q - p) (bn (p \cdot \alpha2), p \cdot \alpha2)$
 by (*simp add: alpha-set*) (*metis (mono-tags, lifting) Diff-eqvt bn-eqvt fresh-star-permute-iff permute-minus-cancel(2) permute-perm-def supp-eqvt*)
ultimately show $p \cdot tAct f1 \alpha1 t1 =_{\alpha} p \cdot tAct f2 \alpha2 t2$ **using 0**
 by *auto*
next
assume $p \cdot tAct f1 \alpha1 t1 =_{\alpha} p \cdot tAct f2 \alpha2 t2$
then obtain q **where** $0: f1 = f2$ **and** $1: (bn (p \cdot \alpha1), p \cdot t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) q (bn (p \cdot \alpha2), p \cdot t2)$ **and** $2: (bn (p \cdot \alpha1), p \cdot \alpha1) \approx_{set} (=) supp q (bn (p \cdot \alpha2), p \cdot \alpha2)$
 by *auto*
 {
from 1 and t1 and t2 have $supp-rel (=_{\alpha}) t1 - bn \alpha1 = supp-rel (=_{\alpha}) t2 - bn \alpha2$
 by (*metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff*)
moreover with 1 and t2 have $(supp-rel (=_{\alpha}) t1 - bn \alpha1) \#* (-p + q + p)$
 by (*auto simp add: fresh-star-def fresh-perm alphas*) (*metis (no-types, lifting) DiffI bn-eqvt mem-permute-iff permute-minus-cancel(2)*)
moreover from 1 have $-p \cdot q \cdot p \cdot t1 =_{\alpha} t2$
using *alpha-tAct.IH(1)* **by** (*simp add: alpha-set*) (*metis (no-types, lifting) permute-eqvt permute-minus-cancel(2)*)
moreover from 1 have $-p \cdot q \cdot p \cdot bn \alpha1 = bn \alpha2$
by (*metis alpha-set bn-eqvt permute-minus-cancel(2)*)
ultimately have $(bn \alpha1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (-p + q + p) (bn \alpha2, t2)$
 by (*simp add: alpha-set*)
 }
moreover
 {
from 2 have $supp \alpha1 - bn \alpha1 = supp \alpha2 - bn \alpha2$
by (*metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff supp-eqvt*)
moreover with 2 have $(supp \alpha1 - bn \alpha1) \#* (-p + q + p)$
by (*auto simp add: fresh-star-def fresh-perm alphas*) (*metis (no-types, lifting) DiffI bn-eqvt mem-permute-iff permute-minus-cancel(1) supp-eqvt*)
moreover from 2 have $-p \cdot q \cdot p \cdot \alpha1 = \alpha2$
by (*simp add: alpha-set*)
moreover have $-p \cdot q \cdot p \cdot bn \alpha1 = bn \alpha2$
by (*simp add: bn-eqvt calculation(3)*)
 }

```

      ultimately have (bn  $\alpha 1$ ,  $\alpha 1$ )  $\approx_{set}$  (=)  $supp$  ( $-p + q + p$ ) (bn  $\alpha 2$ ,  $\alpha 2$ )
      by (simp add: alpha-set)
    }
    ultimately show  $tAct$   $f1$   $\alpha 1$   $t1 =_{\alpha}$   $tAct$   $f2$   $\alpha 2$   $t2$  using 0
    by auto
  qed
qed simp-all

```

```

lemma alpha-Tree-eqvt [eqvt]:  $t1 =_{\alpha} t2 \implies p \cdot t1 =_{\alpha} p \cdot t2$ 
by (metis alpha-Tree-eqvt')

```

$(=_{\alpha})$ is an equivalence relation.

```

lemma alpha-Tree-reflp: reflp alpha-Tree

```

```

proof (rule reflpI)
  fix t :: ('a,'b,'c,'d) Tree
  show  $t =_{\alpha} t$ 
  proof (induction t)
    case tConj then show ?case by (metis alpha-tConj rel-bset.rep-eq rel-setI)
  next
    case tNot then show ?case by (metis alpha-tNot)
  next
    case tPred then show ?case by (metis alpha-tPred)
  next
    case tAct then show ?case by (metis (mono-tags) alpha-tAct alpha-refl(1))
  qed
qed

```

```

lemma alpha-Tree-symp: symp alpha-Tree

```

```

proof (rule sympI)
  fix x y :: ('a,'b,'c,'d) Tree
  assume  $x =_{\alpha} y$  then show  $y =_{\alpha} x$ 
  proof (induction x y rule: alpha-Tree-induct)
    case tConj then show ?case
    by (simp add: rel-bset-def rel-set-def) metis
  next
    case (tAct  $f1$   $\alpha 1$   $t1$   $f2$   $\alpha 2$   $t2$ )
    then obtain  $p$  where  $f1=f2 \wedge (bn \alpha 1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) p (bn \alpha 2, t2) \wedge (bn \alpha 1, \alpha 1) \approx_{set} (=) supp p (bn \alpha 2, \alpha 2)$ 
    by auto
    then have  $f1=f2 \wedge (bn \alpha 2, t2) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (-p) (bn \alpha 1, t1) \wedge (bn \alpha 2, \alpha 2) \approx_{set} (=) supp (-p) (bn \alpha 1, \alpha 1)$ 
    using tAct.IH by (metis (mono-tags, lifting) alpha-Tree-eqvt alpha-sym(1) permute-minus-cancel(2))
    then show ?case
    by auto
  qed simp-all
qed

```

```

lemma alpha-Tree-transp: transp alpha-Tree

```

```

proof (rule transpI)
  fix  $x\ y\ z:: ('a, 'b, 'c, 'd)\ Tree$ 
  assume  $x =_\alpha y$  and  $y =_\alpha z$ 
  then show  $x =_\alpha z$ 
  proof (induction  $x\ y$  arbitrary:  $z$  rule: alpha-Tree-induct)
    case (tConj tset-x tset-y) show ?case
      proof (cases  $z$ )
        fix  $tset-z$ 
        assume  $z: z = tConj\ tset-z$ 
        have  $rel-bset\ (=_\alpha)\ tset-x\ tset-z$ 
          unfolding rel-bset.rep-eq rel-set-def Ball-def Bex-def
          proof
            show  $\forall x'. x' \in set-bset\ tset-x \longrightarrow (\exists z'. z' \in set-bset\ tset-z \wedge x' =_\alpha z')$ 
            proof (rule allI, rule impI)
              fix  $x'$  assume  $1: x' \in set-bset\ tset-x$ 
              then obtain  $y'$  where  $2: y' \in set-bset\ tset-y$  and  $3: x' =_\alpha y'$ 
                by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
              from  $2$  obtain  $z'$  where  $4: z' \in set-bset\ tset-z$  and  $5: y' =_\alpha z'$ 
                by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1)
              from  $1\ 2\ 3\ 5$  have  $x' =_\alpha z'$ 
                by (rule tConj.IH)
              with  $4$  show  $\exists z'. z' \in set-bset\ tset-z \wedge x' =_\alpha z'$ 
                by auto
            qed
          next
            show  $\forall z'. z' \in set-bset\ tset-z \longrightarrow (\exists x'. x' \in set-bset\ tset-x \wedge x' =_\alpha z')$ 
            proof (rule allI, rule impI)
              fix  $z'$  assume  $1: z' \in set-bset\ tset-z$ 
              then obtain  $y'$  where  $2: y' \in set-bset\ tset-y$  and  $3: y' =_\alpha z'$ 
                by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1)
              from  $2$  obtain  $x'$  where  $4: x' \in set-bset\ tset-x$  and  $5: x' =_\alpha y'$ 
                by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
              from  $4\ 2\ 5\ 3$  have  $x' =_\alpha z'$ 
                by (rule tConj.IH)
              with  $4$  show  $\exists x'. x' \in set-bset\ tset-x \wedge x' =_\alpha z'$ 
                by auto
            qed
          with  $z$  show  $tConj\ tset-x =_\alpha z$ 
            by simp
          qed (insert tConj.prem1, auto)
        next
          case tNot then show ?case
            by (cases  $z$ ) simp-all
          next
            case tPred then show ?case
              by simp
          next
            case (tAct f1  $\alpha_1$  t1 f2  $\alpha_2$  t2) show ?case

```

```

proof (cases z)
  fix f  $\alpha$  t
  assume z: z = tAct f  $\alpha$  t
  obtain p where 1: f1=f2  $\wedge$  (bn  $\alpha$ 1, t1)  $\approx_{\text{set}} (=_{\alpha})$  (supp-rel (=  $\alpha$ )) p (bn
 $\alpha$ 2, t2)  $\wedge$  (bn  $\alpha$ 1,  $\alpha$ 1)  $\approx_{\text{set}} (=)$  supp p (bn  $\alpha$ 2,  $\alpha$ 2)
  using tAct.hyps by auto
  obtain q where 2: f2=f  $\wedge$  (bn  $\alpha$ 2, t2)  $\approx_{\text{set}} (=_{\alpha})$  (supp-rel (=  $\alpha$ )) q (bn  $\alpha$ ,
t)  $\wedge$  (bn  $\alpha$ 2,  $\alpha$ 2)  $\approx_{\text{set}} (=)$  supp q (bn  $\alpha$ ,  $\alpha$ )
  using tAct.prem1 z by auto
  have f1=f  $\wedge$  (bn  $\alpha$ 1, t1)  $\approx_{\text{set}} (=_{\alpha})$  (supp-rel (=  $\alpha$ )) (q + p) (bn  $\alpha$ , t)
  proof -
    have supp-rel (=  $\alpha$ ) t1 - bn  $\alpha$ 1 = supp-rel (=  $\alpha$ ) t - bn  $\alpha$ 
    using 1 and 2 by (metis alpha-set)
    moreover have (supp-rel (=  $\alpha$ ) t1 - bn  $\alpha$ 1)  $\sharp^*$  (q + p)
    using 1 and 2 by (metis alpha-set fresh-star-plus)
    moreover have (q + p)  $\cdot$  t1 =  $\alpha$  t
    using 1 and 2 and tAct.IH by (metis (no-types, lifting) alpha-Tree-eqvt
alpha-set permute-minus-cancel(1) permute-plus)
    moreover have (q + p)  $\cdot$  bn  $\alpha$ 1 = bn  $\alpha$ 
    using 1 and 2 by (metis alpha-set permute-plus)
    moreover have f1=f
    using 1 and 2 by simp
    ultimately show ?thesis
    by (metis alpha-set)
  qed
  moreover have (bn  $\alpha$ 1,  $\alpha$ 1)  $\approx_{\text{set}} (=)$  supp (q + p) (bn  $\alpha$ ,  $\alpha$ )
  using 1 and 2 by (metis (mono-tags) alpha-trans(1) permute-plus)
  ultimately show tAct f1  $\alpha$ 1 t1 =  $\alpha$  z
  using z by auto
qed (insert tAct.prem1, auto)
qed
qed

```

lemma alpha-Tree-equivp: equivp alpha-Tree
by (auto intro: equivpI alpha-Tree-reflp alpha-Tree-symp alpha-Tree-transp)

α -equivalent trees have the same support modulo α -equivalence.

lemma alpha-Tree-supp-rel:

```

assumes t1 =  $\alpha$  t2
shows supp-rel (=  $\alpha$ ) t1 = supp-rel (=  $\alpha$ ) t2
using assms proof (induction rule: alpha-Tree-induct)
  case (tConj tset1 tset2)
  have sym:  $\bigwedge x y. \text{rel-bset} (=_{\alpha}) x y \longleftrightarrow \text{rel-bset} (=_{\alpha}) y x$ 
  by (meson alpha-Tree-symp bset.rel-symp sympE)
  {
    fix a b
    from tConj.hyps have *: rel-bset (=  $\alpha$ ) ((a  $\equiv$  b)  $\cdot$  tset1) ((a  $\equiv$  b)  $\cdot$  tset2)
    by (metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj)
    have rel-bset (=  $\alpha$ ) ((a  $\equiv$  b)  $\cdot$  tset1) tset1  $\longleftrightarrow$  rel-bset (=  $\alpha$ ) ((a  $\equiv$  b)  $\cdot$  tset2)
  }

```

```

tset2
  by (rule iffI) (metis * alpha-Tree-transp bset.rel-transp sym tConj.hyps
transpE)+
}
then show ?case
  by (simp add: supp-rel-def)
next
case tNot then show ?case
  by (simp add: supp-rel-def)
next
case (tAct f1  $\alpha$ 1 t1 f2  $\alpha$ 2 t2)
{
  fix a b
  have tAct f1  $\alpha$ 1 t1 = $_{\alpha}$  tAct f2  $\alpha$ 2 t2
  using tAct.hyps by simp
  then have (a  $\rightleftharpoons$  b)  $\cdot$  tAct f1  $\alpha$ 1 t1 = $_{\alpha}$  tAct f1  $\alpha$ 1 t1  $\longleftrightarrow$  (a  $\rightleftharpoons$  b)  $\cdot$  tAct f2
 $\alpha$ 2 t2 = $_{\alpha}$  tAct f2  $\alpha$ 2 t2
  by (metis (no-types, lifting) alpha-Tree-eqvt alpha-Tree-symp alpha-Tree-transp
sympE transpE)
}
then show ?case
  by (simp add: supp-rel-def)
qed simp-all

```

$tAct$ preserves α -equivalence.

lemma *alpha-Tree-tAct*:

assumes $t1 =_{\alpha} t2$

shows $tAct f \alpha t1 =_{\alpha} tAct f \alpha t2$

proof –

have $(bn \alpha, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) 0 (bn \alpha, t2)$

using *assms* **by** (simp add: alpha-Tree-supp-rel alpha-set fresh-star-zero)

moreover have $(bn \alpha, \alpha) \approx_{set} (=) supp 0 (bn \alpha, \alpha)$

by (metis (full-types) alpha-refl(1))

ultimately show *?thesis*

by *auto*

qed

The following lemmas describe the support modulo α -equivalence.

lemma *supp-rel-tNot* [*simp*]: $supp-rel (=_{\alpha}) (tNot t) = supp-rel (=_{\alpha}) t$

unfolding *supp-rel-def* **by** *simp*

lemma *supp-rel-tPred* [*simp*]: $supp-rel (=_{\alpha}) (tPred f \varphi) = supp f \cup supp \varphi$

unfolding *supp-rel-def supp-def* **by** (simp add: Collect-imp-eq Collect-neg-eq)

The support modulo α -equivalence of $tAct \alpha t$ is not easily described: when t has infinite support (modulo α -equivalence), the support (modulo α -equivalence) of $tAct \alpha t$ may still contain names in $bn \alpha$. This incongruity is avoided when t has finite support modulo α -equivalence.

lemma *infinite-mono*: $infinite S \implies (\bigwedge x. x \in S \implies x \in T) \implies infinite T$

by (*metis infinite-super subsetI*)

lemma *supp-rel-tAct [simp]*:

assumes *finite (supp-rel (=α) t)*

shows $\text{supp-rel } (=_{\alpha}) (tAct f \alpha t) = \text{supp } f \cup (\text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha)$

proof

show $\text{supp } f \cup (\text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha) \subseteq \text{supp-rel } (=_{\alpha}) (tAct f \alpha t)$

proof

fix x

assume $x \in \text{supp } f \cup (\text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha)$

moreover

{

assume $x1: x \in \text{supp } f$

from $x1$ **have** *infinite* $\{b. (x \rightleftharpoons b) \cdot f \neq f\}$

unfolding *supp-def* ..

then have *infinite* $(\{b. (x \rightleftharpoons b) \cdot f \neq f\} - \text{supp } f)$

by (*simp add: finite-supp*)

moreover

{

fix b

assume $b \in \{b. (x \rightleftharpoons b) \cdot f \neq f\} - \text{supp } f$

then have $b1: (x \rightleftharpoons b) \cdot f \neq f$ **and** $b2: b \notin \text{supp } f$

by *simp+*

then have *sort-of* $x = \text{sort-of } b$

using *swap-different-sorts* **by** *fastforce*

then have $(x \rightleftharpoons b) \cdot \text{supp } f \neq \text{supp } f$

using $b2$ $x1$ **using** *swap-set-in* **by** *blast*

then have $b \in \{b. \neg (x \rightleftharpoons b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t\}$

by *auto*

}

ultimately have *infinite* $\{b. \neg (x \rightleftharpoons b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t\}$

by (*rule infinite-mono*)

then have $x \in \text{supp-rel } (=_{\alpha}) (tAct f \alpha t)$

unfolding *supp-rel-def* ..

}

moreover

{

assume $x1: x \in \text{supp } \alpha$ **and** $x2: x \notin \text{bn } \alpha$

from $x1$ **have** *infinite* $\{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha\}$

unfolding *supp-def* ..

then have *infinite* $(\{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha\} - \text{supp } \alpha)$

by (*simp add: finite-supp*)

moreover

{

fix b

assume $b \in \{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha\} - \text{supp } \alpha$

then have $b1: (x \rightleftharpoons b) \cdot \alpha \neq \alpha$ **and** $b2: b \notin \text{supp } \alpha - \text{bn } \alpha$

by *simp+*

}


```

from  $b1$  have  $sort\text{-}of\ x = sort\text{-}of\ b$ 
  using  $swap\text{-}different\text{-}sorts$  by  $fastforce$ 
then have  $(x \rightleftharpoons b) \cdot (supp\ \alpha - bn\ \alpha) \neq supp\ \alpha - bn\ \alpha$ 
  using  $b2\ x1\ x2$  by ( $simp\ add: swap\text{-}set\text{-}in$ )
then have  $b \in \{b. \neg (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_{\alpha} tAct\ f\ \alpha\ t\}$ 
  by ( $auto\ simp\ add: alpha\text{-}set\ Diff\text{-}eqvt\ bn\text{-}eqvt$ )
}
ultimately have  $infinite\ \{b. \neg (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_{\alpha} tAct\ f\ \alpha\ t\}$ 
  by ( $rule\ infinite\text{-}mono$ )
then have  $x \in supp\text{-}rel\ (=_{\alpha})\ (tAct\ f\ \alpha\ t)$ 
  unfolding  $supp\text{-}rel\text{-}def\ ..$ 
}
moreover
{
  assume  $x1: x \in supp\text{-}rel\ (=_{\alpha})\ t$  and  $x2: x \notin bn\ \alpha$ 
from  $x1$  have  $infinite\ \{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
  unfolding  $supp\text{-}rel\text{-}def\ ..$ 
then have  $infinite\ (\{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\} - supp\text{-}rel\ (=_{\alpha})\ t)$ 
  using  $assms$  by  $simp$ 
moreover
{
  fix  $b$ 
  assume  $b \in \{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\} - supp\text{-}rel\ (=_{\alpha})\ t$ 
  then have  $b1: \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t$  and  $b2: b \notin supp\text{-}rel\ (=_{\alpha})\ t - bn\ \alpha$ 
  by  $simp+$ 
from  $b1$  have  $(x \rightleftharpoons b) \cdot t \neq t$ 
  by ( $metis\ alpha\text{-}Tree\text{-}reflp\ reflpE$ )
then have  $sort\text{-}of\ x = sort\text{-}of\ b$ 
  using  $swap\text{-}different\text{-}sorts$  by  $fastforce$ 
then have  $(x \rightleftharpoons b) \cdot (supp\text{-}rel\ (=_{\alpha})\ t - bn\ \alpha) \neq supp\text{-}rel\ (=_{\alpha})\ t - bn\ \alpha$ 
  using  $b2\ x1\ x2$  by ( $simp\ add: swap\text{-}set\text{-}in$ )
then have  $supp\text{-}rel\ (=_{\alpha})\ ((x \rightleftharpoons b) \cdot t) - bn\ ((x \rightleftharpoons b) \cdot \alpha) \neq supp\text{-}rel\ (=_{\alpha})\ t - bn\ \alpha$ 
  by ( $simp\ add: Diff\text{-}eqvt\ bn\text{-}eqvt$ )
then have  $b \in \{b. \neg (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_{\alpha} tAct\ f\ \alpha\ t\}$ 
  by ( $simp\ add: alpha\text{-}set$ )
}
}
ultimately have  $infinite\ \{b. \neg (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_{\alpha} tAct\ f\ \alpha\ t\}$ 
  by ( $rule\ infinite\text{-}mono$ )
then have  $x \in supp\text{-}rel\ (=_{\alpha})\ (tAct\ f\ \alpha\ t)$ 
  unfolding  $supp\text{-}rel\text{-}def\ ..$ 
}
ultimately show  $x \in supp\text{-}rel\ (=_{\alpha})\ (tAct\ f\ \alpha\ t)$ 
  by  $auto$ 
qed
next
show  $supp\text{-}rel\ (=_{\alpha})\ (tAct\ f\ \alpha\ t) \subseteq supp\ f \cup (supp\ \alpha \cup supp\text{-}rel\ (=_{\alpha})\ t - bn\ \alpha)$ 
proof
  fix  $x$ 

```

assume $x \in \text{supp-rel } (=_{\alpha}) (tAct f \alpha t)$
then have $*$: $\text{infinite } \{b. \neg (x \equiv b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t\}$
unfolding *supp-rel-def ..*
moreover
{
fix b
assume $\neg (x \equiv b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t$
then have $(x \equiv b) \cdot f \neq f \vee (x \equiv b) \cdot \alpha \neq \alpha \vee \neg (x \equiv b) \cdot t =_{\alpha} t$
using *alpha-Tree-tAct by force*
}
ultimately have $\text{infinite } \{b. (x \equiv b) \cdot f \neq f \vee (x \equiv b) \cdot \alpha \neq \alpha \vee \neg (x \equiv b) \cdot t =_{\alpha} t\}$
using *infinite-mono mem-Collect-eq by force*
then have $\text{infinite } \{b. (x \equiv b) \cdot f \neq f\} \vee \text{infinite } \{b. (x \equiv b) \cdot \alpha \neq \alpha\} \vee$
 $\text{infinite } \{b. \neg (x \equiv b) \cdot t =_{\alpha} t\}$
by (*metis (mono-tags) finite-Collect-disjI*)
then have $x \in \text{supp } f \cup \text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t$
by (*simp add: supp-def supp-rel-def*)
moreover
{
assume $**$: $x \in \text{bn } \alpha \wedge x \notin \text{supp } f$
from $*$ **obtain** b **where** $b0$: $\neg (x \equiv b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t$ **and** $b1$:
 $b \notin \text{supp } f$ **and** $b2$: $b \notin \text{supp } \alpha$ **and** $b3$: $b \notin \text{supp-rel } (=_{\alpha}) t$
using *assms by (metis (no-types, lifting) UnCI finite-UnI finite-supp*
infinite-mono mem-Collect-eq)
let $?p = (x \equiv b)$
have $\text{supp-rel } (=_{\alpha}) ((x \equiv b) \cdot t) - \text{bn } ((x \equiv b) \cdot \alpha) = \text{supp-rel } (=_{\alpha}) t -$
 $\text{bn } \alpha$
using $**$ **and** $b3$ **by** (*metis (no-types, lifting) Diff-eqvt Diff-iff alpha-Tree-eqvt'*
alpha-Tree-eqvt-aux bn-eqvt swap-set-not-in)
moreover then have $(\text{supp-rel } (=_{\alpha}) ((x \equiv b) \cdot t) - \text{bn } ((x \equiv b) \cdot \alpha)) \#*$
 $?p$
using $**$ **and** $b3$ **by** (*metis Diff-iff fresh-perm fresh-star-def swap-atom-simps(3)*)
moreover have $?p \cdot (x \equiv b) \cdot t =_{\alpha} t$
using *alpha-Tree-reflp reflpE by force*
moreover have $?p \cdot \text{bn } ((x \equiv b) \cdot \alpha) = \text{bn } \alpha$
by (*simp add: bn-eqvt*)
moreover have $\text{supp } ((x \equiv b) \cdot \alpha) - \text{bn } ((x \equiv b) \cdot \alpha) = \text{supp } \alpha - \text{bn } \alpha$
using $**$ **and** $b2$ **by** (*metis (mono-tags, hide-lams) Diff-eqvt Diff-iff bn-eqvt*
supp-eqvt swap-set-not-in)
moreover then have $(\text{supp } ((x \equiv b) \cdot \alpha) - \text{bn } ((x \equiv b) \cdot \alpha)) \#* ?p$
using $**$ **and** $b2$ **by** (*simp add: fresh-star-def fresh-def supp-perm*) (*metis*
Diff-iff swap-atom-simps(3))
moreover have $?p \cdot (x \equiv b) \cdot \alpha = \alpha$
by *simp*
ultimately have $\exists p. (\text{bn } ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot t) \approx_{\text{set}} (=_{\alpha}) \text{supp-rel}$
 $(=_{\alpha}) p (\text{bn } \alpha, t) \wedge$
 $(\text{bn } ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot \alpha) \approx_{\text{set}} (=) \text{supp } p (\text{bn } \alpha, \alpha)$
by (*auto simp add: alpha-set.simps*)

```

    moreover have  $(x \equiv b) \cdot f = f$  using ** and b1
      by (simp add: fresh-def swap-fresh-fresh)
    ultimately have  $(x \equiv b) \cdot tAct\ f\ \alpha\ t =_{\alpha} tAct\ f\ \alpha\ t$ 
      by simp
    with b0 have False ..
  }
  ultimately show  $x \in supp\ f \cup (supp\ \alpha \cup supp\text{-rel}\ (=_{\alpha})\ t - bn\ \alpha)$ 
    by blast
qed
qed

```

We define the type of (infinitely branching) trees quotiented by α -equivalence.

quotient-type

```

('idx,'pred,'act,'eff) Tree $_{\alpha}$  = ('idx,'pred::pt,'act::bn,'eff::fs) Tree / alpha-Tree
by (fact alpha-Tree-equivp)

```

lemma *Tree $_{\alpha}$ -abs-rep* [simp]: *abs-Tree $_{\alpha}$ (rep-Tree $_{\alpha}$ t $_{\alpha}$) = t $_{\alpha}$*
by (metis *Quotient-Tree $_{\alpha}$ Quotient-abs-rep*)

lemma *Tree $_{\alpha}$ -rep-abs* [simp]: *rep-Tree $_{\alpha}$ (abs-Tree $_{\alpha}$ t) = $_{\alpha}$ t*
by (metis *Tree $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$ -abs-rep*)

The permutation operation is lifted from trees.

instantiation *Tree $_{\alpha}$:: (type, pt, bn, fs) pt*
begin

```

lift-definition permute-Tree $_{\alpha}$  :: perm  $\Rightarrow$  ('a,'b,'c,'d) Tree $_{\alpha}$   $\Rightarrow$  ('a,'b,'c,'d) Tree $_{\alpha}$ 
  is permute
by (fact alpha-Tree-eqvt)

```

instance

proof

```

  fix t $_{\alpha}$  :: (-,-,-) Tree $_{\alpha}$ 

```

```

  show  $0 \cdot t_{\alpha} = t_{\alpha}$ 

```

```

    by transfer (metis alpha-Tree-equivp equivp-reflp permute-zero)

```

next

```

  fix p q :: perm and t $_{\alpha}$  :: (-,-,-) Tree $_{\alpha}$ 

```

```

  show  $(p + q) \cdot t_{\alpha} = p \cdot q \cdot t_{\alpha}$ 

```

```

    by transfer (metis alpha-Tree-equivp equivp-reflp permute-plus)

```

qed

end

The abstraction function from trees to trees modulo α -equivalence is equivariant. The representation function is equivariant modulo α -equivalence.

lemmas *permute-Tree $_{\alpha}$.abs-eq* [eqvt, simp]

lemma *alpha-Tree-permute-rep-commute* [simp]: *p \cdot rep-Tree $_{\alpha}$ t $_{\alpha}$ = $_{\alpha}$ rep-Tree $_{\alpha}$ (p \cdot t $_{\alpha}$)*

by (*metis Tree_α.abs-eq-iff Tree_α.abs-rep permute-Tree_α.abs-eq*)

12.3 Constructors for trees modulo α -equivalence

The constructors are lifted from trees.

lift-definition *Conj_α* :: ('idx,'pred,'act,'eff) Tree_α set['idx] ⇒ ('idx,'pred::pt,'act::bn,'eff::fs)
Tree_α is
tConj
 by *simp*

lemma *map-bset-abs-rep-Tree_α*: map-bset abs-Tree_α (map-bset rep-Tree_α tset_α) =
tset_α
 by (*metis (full-types) Quotient-Tree_α Quotient-abs-rep bset-lifting.bset-quot-map*)

lemma *Conj_α-def'*: Conj_α tset_α = abs-Tree_α (tConj (map-bset rep-Tree_α tset_α))
 by (*metis Conj_α.abs-eq map-bset-abs-rep-Tree_α*)

lift-definition *Not_α* :: ('idx,'pred,'act,'eff) Tree_α ⇒ ('idx,'pred::pt,'act::bn,'eff::fs)
Tree_α is
tNot
 by *simp*

lift-definition *Pred_α* :: 'eff ⇒ 'pred ⇒ ('idx,'pred::pt,'act::bn,'eff::fs) Tree_α is
tPred

lift-definition *Act_α* :: 'eff ⇒ 'act ⇒ ('idx,'pred,'act,'eff) Tree_α ⇒ ('idx,'pred::pt,'act::bn,'eff::fs)
Tree_α is
tAct
 by (*fact alpha-Tree-tAct*)

The lifted constructors are equivariant.

lemma *Conj_α-eqvt* [eqvt, simp]: p · Conj_α tset_α = Conj_α (p · tset_α)

proof –

{
 fix x
 assume x ∈ set-bset (p · map-bset rep-Tree_α tset_α)
 then obtain y where y ∈ set-bset (map-bset rep-Tree_α tset_α) and x = p · y
 by (*metis imageE permute-bset.rep-eq permute-set-eq-image*)
 then obtain t_α where 1: t_α ∈ set-bset tset_α and 2: x = p · rep-Tree_α t_α
 by (*metis imageE map-bset.rep-eq*)
 let ?x' = rep-Tree_α (p · t_α)
 from 1 have p · t_α ∈ set-bset (p · tset_α)
 by (*metis mem-permute-iff permute-bset.rep-eq*)
 then have ?x' ∈ set-bset (map-bset rep-Tree_α (p · tset_α))
 by (*simp add: bset.set-map*)
 moreover from 2 have x =_α ?x'
 by (*metis alpha-Tree-permute-rep-commute*)
 ultimately have ∃ x' ∈ set-bset (map-bset rep-Tree_α (p · tset_α)). x =_α x'

```

    ..
  }
  moreover
  {
    fix y
    assume  $y \in \text{set-bset } (\text{map-bset rep-Tree}_\alpha (p \cdot \text{tset}_\alpha))$ 
    then obtain  $x$  where  $x \in \text{set-bset } (p \cdot \text{tset}_\alpha)$  and  $\text{rep-Tree}_\alpha x = y$ 
      by (metis imageE map-bset.rep-eq)
    then obtain  $t_\alpha$  where 1:  $t_\alpha \in \text{set-bset } \text{tset}_\alpha$  and 2:  $\text{rep-Tree}_\alpha (p \cdot t_\alpha) = y$ 
      by (metis imageE permute-bset.rep-eq permute-set-eq-image)
    let  $?y' = p \cdot \text{rep-Tree}_\alpha t_\alpha$ 
    from 1 have  $\text{rep-Tree}_\alpha t_\alpha \in \text{set-bset } (\text{map-bset rep-Tree}_\alpha \text{tset}_\alpha)$ 
      by (simp add: bset.set-map)
    then have  $?y' \in \text{set-bset } (p \cdot \text{map-bset rep-Tree}_\alpha \text{tset}_\alpha)$ 
      by (metis mem-permute-iff permute-bset.rep-eq)
    moreover from 2 have  $?y' =_\alpha y$ 
      by (metis alpha-Tree-permute-rep-commute)
    ultimately have  $\exists y' \in \text{set-bset } (p \cdot \text{map-bset rep-Tree}_\alpha \text{tset}_\alpha). y' =_\alpha y$ 
    ..
  }
  ultimately show ?thesis
    by (simp add: Conj $_\alpha$ -def' map-bset-eqv rel-bset-def rel-set-def Tree $_\alpha$ .abs-eq-iff)
qed

```

lemma $\text{Not}_\alpha\text{-eqvt}$ [eqvt, simp]: $p \cdot \text{Not}_\alpha t_\alpha = \text{Not}_\alpha (p \cdot t_\alpha)$
 by (induct t_α) (simp add: Not_α .abs-eq)

lemma $\text{Pred}_\alpha\text{-eqvt}$ [eqvt, simp]: $p \cdot \text{Pred}_\alpha f \varphi = \text{Pred}_\alpha (p \cdot f) (p \cdot \varphi)$
 by (simp add: Pred_α .abs-eq)

lemma $\text{Act}_\alpha\text{-eqvt}$ [eqvt, simp]: $p \cdot \text{Act}_\alpha f \alpha t_\alpha = \text{Act}_\alpha (p \cdot f) (p \cdot \alpha) (p \cdot t_\alpha)$
 by (induct t_α) (simp add: Act_α .abs-eq)

The lifted constructors are injective (except for Act_α).

lemma $\text{Conj}_\alpha\text{-eq-iff}$ [simp]: $\text{Conj}_\alpha \text{tset1}_\alpha = \text{Conj}_\alpha \text{tset2}_\alpha \iff \text{tset1}_\alpha = \text{tset2}_\alpha$
proof

```

  assume  $\text{Conj}_\alpha \text{tset1}_\alpha = \text{Conj}_\alpha \text{tset2}_\alpha$ 
  then have  $t\text{Conj } (\text{map-bset rep-Tree}_\alpha \text{tset1}_\alpha) =_\alpha t\text{Conj } (\text{map-bset rep-Tree}_\alpha \text{tset2}_\alpha)$ 
    by (metis Conj $_\alpha$ -def' Tree $_\alpha$ .abs-eq-iff)
  then have  $\text{rel-bset } (=_\alpha) (\text{map-bset rep-Tree}_\alpha \text{tset1}_\alpha) (\text{map-bset rep-Tree}_\alpha \text{tset2}_\alpha)$ 
    by (auto elim: alpha-Tree.cases)
  then show  $\text{tset1}_\alpha = \text{tset2}_\alpha$ 
    using Quotient-Tree $_\alpha$  Quotient-rel-abs2 bset-lifting.bset-quot-map map-bset-abs-rep-Tree $_\alpha$ 
  by fastforce
qed (fact arg-cong)

```

lemma $\text{Not}_\alpha\text{-eq-iff}$ [simp]: $\text{Not}_\alpha t1_\alpha = \text{Not}_\alpha t2_\alpha \iff t1_\alpha = t2_\alpha$
proof

assume $\text{Not}_\alpha t1_\alpha = \text{Not}_\alpha t2_\alpha$
then have $t\text{Not} (\text{rep-Tree}_\alpha t1_\alpha) =_\alpha t\text{Not} (\text{rep-Tree}_\alpha t2_\alpha)$
by (*metis* $\text{Not}_\alpha.\text{abs-eq}$ $\text{Tree}_\alpha.\text{abs-eq-iff}$ $\text{Tree}_\alpha.\text{abs-rep}$)
then have $\text{rep-Tree}_\alpha t1_\alpha =_\alpha \text{rep-Tree}_\alpha t2_\alpha$
using *alpha-Tree.cases* **by** *auto*
then show $t1_\alpha = t2_\alpha$
by (*metis* $\text{Tree}_\alpha.\text{abs-eq-iff}$ $\text{Tree}_\alpha.\text{abs-rep}$)
next
assume $t1_\alpha = t2_\alpha$ **then show** $\text{Not}_\alpha t1_\alpha = \text{Not}_\alpha t2_\alpha$
by *simp*
qed

lemma $\text{Pred}_\alpha.\text{eq-iff}$ [*simp*]: $\text{Pred}_\alpha f1 \varphi1 = \text{Pred}_\alpha f2 \varphi2 \longleftrightarrow f1 = f2 \wedge \varphi1 = \varphi2$
proof

assume $\text{Pred}_\alpha f1 \varphi1 = \text{Pred}_\alpha f2 \varphi2$
then have $(t\text{Pred} f1 \varphi1 :: ('e, 'b, 'f, 'd) \text{Tree}) =_\alpha t\text{Pred} f2 \varphi2$ — note the unrelated type
by (*metis* $\text{Pred}_\alpha.\text{abs-eq}$ $\text{Tree}_\alpha.\text{abs-eq-iff}$)
then show $f1 = f2 \wedge \varphi1 = \varphi2$
using *alpha-Tree.cases* **by** *auto*
next
assume $f1 = f2 \wedge \varphi1 = \varphi2$ **then show** $\text{Pred}_\alpha f1 \varphi1 = \text{Pred}_\alpha f2 \varphi2$
by *simp*
qed

lemma $\text{Act}_\alpha.\text{eq-iff}$: $\text{Act}_\alpha f1 \alpha1 t1 = \text{Act}_\alpha f2 \alpha2 t2 \longleftrightarrow t\text{Act} f1 \alpha1 (\text{rep-Tree}_\alpha t1) =_\alpha t\text{Act} f2 \alpha2 (\text{rep-Tree}_\alpha t2)$
by (*metis* $\text{Act}_\alpha.\text{abs-eq}$ $\text{Tree}_\alpha.\text{abs-eq-iff}$ $\text{Tree}_\alpha.\text{abs-rep}$)

The lifted constructors are free (except for Act_α).

lemma $\text{Tree}_\alpha.\text{free}$ [*simp*]:
shows $\text{Conj}_\alpha \text{tset}_\alpha \neq \text{Not}_\alpha t_\alpha$
and $\text{Conj}_\alpha \text{tset}_\alpha \neq \text{Pred}_\alpha f \varphi$
and $\text{Conj}_\alpha \text{tset}_\alpha \neq \text{Act}_\alpha f \alpha t_\alpha$
and $\text{Not}_\alpha t_\alpha \neq \text{Pred}_\alpha f \varphi$
and $\text{Not}_\alpha t1_\alpha \neq \text{Act}_\alpha f \alpha t2_\alpha$
and $\text{Pred}_\alpha f1 \varphi \neq \text{Act}_\alpha f2 \alpha t_\alpha$
by (*simp add: Conj_α-def' Not_α-def Pred_α-def Act_α-def Tree_α.abs-eq-iff*)₊

The following lemmas describe the support of constructed trees modulo α -equivalence.

lemma $\text{supp-alpha-supp-rel}$: $\text{supp} t_\alpha = \text{supp-rel} (=_\alpha) (\text{rep-Tree}_\alpha t_\alpha)$
unfolding supp-def supp-rel-def **by** (*metis* (*mono-tags*, *lifting*) *Collect-cong* $\text{Tree}_\alpha.\text{abs-eq-iff}$ $\text{Tree}_\alpha.\text{abs-rep}$ *alpha-Tree-permute-rep-commute*)

lemma supp-Conj_α [*simp*]: $\text{supp} (\text{Conj}_\alpha \text{tset}_\alpha) = \text{supp} \text{tset}_\alpha$
unfolding supp-def **by** *simp*

lemma supp-Not_α [*simp*]: $\text{supp} (\text{Not}_\alpha t_\alpha) = \text{supp} t_\alpha$

unfolding *supp-def* **by** *simp*

lemma *supp-Pred_α* [*simp*]: $\text{supp } (\text{Pred}_\alpha f \varphi) = \text{supp } f \cup \text{supp } \varphi$
unfolding *supp-def* **by** (*simp add: Collect-imp-eq Collect-neg-eq*)

lemma *supp-Act_α* [*simp*]:
assumes *finite* (*supp t_α*)
shows $\text{supp } (\text{Act}_\alpha f \alpha t_\alpha) = \text{supp } f \cup (\text{supp } \alpha \cup \text{supp } t_\alpha - \text{bn } \alpha)$
using *assms* **by** (*metis Act_α.abs-eq Tree_α-abs-rep Tree_α-rep-abs alpha-Tree-supp-rel supp-alpha-supp-rel supp-rel-tAct*)

12.4 Induction over trees modulo α -equivalence

lemma *Tree_α-induct* [*case-names Conj_α Not_α Pred_α Act_α Env_α*, *induct type: Tree_α*]:

fixes *t_α*
assumes $\bigwedge tset_\alpha. (\bigwedge x. x \in \text{set-bset } tset_\alpha \implies P x) \implies P (\text{Conj}_\alpha tset_\alpha)$
and $\bigwedge t_\alpha. P t_\alpha \implies P (\text{Not}_\alpha t_\alpha)$
and $\bigwedge f \text{ pred}. P (\text{Pred}_\alpha f \text{ pred})$
and $\bigwedge f \text{ act } t_\alpha. P t_\alpha \implies P (\text{Act}_\alpha f \text{ act } t_\alpha)$
shows $P t_\alpha$
proof (*rule Tree_α.abs-induct*)
fix *t* **show** $P (\text{abs-Tree}_\alpha t)$
proof (*induction t*)
case (*tConj tset*)
let *?tset_α* = *map-bset abs-Tree_α tset*
have $\text{abs-Tree}_\alpha (t\text{Conj } tset) = \text{Conj}_\alpha ?tset_\alpha$
by (*simp add: Conj_α.abs-eq*)
then show *?case*
using *assms(1) tConj.IH* **by** (*metis imageE map-bset.rep-eq*)
next
case *tNot* **then show** *?case*
using *assms(2)* **by** (*metis Not_α.abs-eq*)
next
case *tPred* **show** *?case*
using *assms(3)* **by** (*metis Pred_α.abs-eq*)
next
case *tAct* **then show** *?case*
using *assms(4)* **by** (*metis Act_α.abs-eq*)
qed
qed

There is no (obvious) strong induction principle for trees modulo α -equivalence: since their support may be infinite, we may not be able to rename bound variables without also renaming free variables.

12.5 Hereditarily finitely supported trees

We cannot obtain the type of infinitary formulas simply as the sub-type of all trees (modulo α -equivalence) that are finitely supported: since an infinite set of trees may be finitely supported even though its members are not (and thus, would not be formulas), the sub-type of *all* finitely supported trees does not validate the induction principle that we desire for formulas.

Instead, we define *hereditarily* finitely supported trees. We require that environments and state predicates are finitely supported.

inductive *hereditarily-fs* :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) Tree $_{\alpha}$ \Rightarrow bool **where**
Conj $_{\alpha}$: finite (supp tset $_{\alpha}$) \Longrightarrow ($\bigwedge t_{\alpha}. t_{\alpha} \in \text{set-bset } tset_{\alpha} \Longrightarrow \text{hereditarily-fs } t_{\alpha}$)
 \Longrightarrow *hereditarily-fs* (Conj $_{\alpha}$ tset $_{\alpha}$)
| *Not $_{\alpha}$* : *hereditarily-fs* t $_{\alpha}$ \Longrightarrow *hereditarily-fs* (Not $_{\alpha}$ t $_{\alpha}$)
| *Pred $_{\alpha}$* : *hereditarily-fs* (Pred $_{\alpha}$ f φ)
| *Act $_{\alpha}$* : *hereditarily-fs* t $_{\alpha}$ \Longrightarrow *hereditarily-fs* (Act $_{\alpha}$ f α t $_{\alpha}$)

hereditarily-fs is equivariant.

lemma *hereditarily-fs-eqvt* [eqvt]:

assumes *hereditarily-fs* t $_{\alpha}$

shows *hereditarily-fs* (p \cdot t $_{\alpha}$)

using *assms* **proof** (*induction rule: hereditarily-fs.induct*)

case *Conj $_{\alpha}$* **then show** ?case

by (*metis* (*erased*, *hide-lams*) *Conj $_{\alpha}$ -eqvt hereditarily-fs.Conj $_{\alpha}$ mem-permute-iff permute-finite permute-minus-cancel(1) set-bset-eqvt supp-eqvt*)

next

case *Not $_{\alpha}$* **then show** ?case

by (*metis* *Not $_{\alpha}$ -eqvt hereditarily-fs.Not $_{\alpha}$*)

next

case *Pred $_{\alpha}$* **then show** ?case

by (*metis* *Pred $_{\alpha}$ -eqvt hereditarily-fs.Pred $_{\alpha}$*)

next

case *Act $_{\alpha}$* **then show** ?case

by (*metis* *Act $_{\alpha}$ -eqvt hereditarily-fs.Act $_{\alpha}$*)

qed

hereditarily-fs is preserved under α -renaming.

lemma *hereditarily-fs-alpha-renaming*:

assumes *Act $_{\alpha}$ f α t $_{\alpha}$ = Act $_{\alpha}$ f' α' t $_{\alpha}'$*

shows *hereditarily-fs* t $_{\alpha}$ \longleftrightarrow *hereditarily-fs* t $_{\alpha}'$

proof

assume *hereditarily-fs* t $_{\alpha}$

then show *hereditarily-fs* t $_{\alpha}'$

using *assms* **by** (*auto simp add: Act $_{\alpha}$ -def Tree $_{\alpha}$.abs-eq-iff alphas*) (*metis* *Tree $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$.abs-rep hereditarily-fs-eqvt permute-Tree $_{\alpha}$.abs-eq*)

next

assume *hereditarily-fs* t $_{\alpha}'$

then show *hereditarily-fs* t $_{\alpha}$


```

using assms by (auto simp add: Act $\alpha$ -def Tree $\alpha$ .abs-eq-iff alphas) (metis
Tree $\alpha$ .abs-eq-iff Tree $\alpha$ .abs-rep hereditarily-fs-eqvt permute-Tree $\alpha$ .abs-eq permute-minus-cancel(2))
qed

```

Hereditarily finitely supported trees have finite support.

```

lemma hereditarily-fs-implies-finite-supp:
  assumes hereditarily-fs t $\alpha$ 
  shows finite (supp t $\alpha$ )
using assms by (induction rule: hereditarily-fs.induct) (simp-all add: finite-supp)

```

12.6 Infinitary formulas

Now, infinitary formulas are simply the sub-type of hereditarily finitely supported trees.

```

typedef ('idx, 'pred::fs, 'act::bn, 'eff::fs) formula = {t $\alpha$ ::('idx, 'pred, 'act, 'eff) Tree $\alpha$ .hereditarily-fs t $\alpha$ }
by (metis hereditarily-fs.Pred $\alpha$  mem-Collect-eq)

```

We set up Isabelle's lifting infrastructure so that we can lift definitions from the type of trees modulo α -equivalence to the sub-type of formulas.

```

setup-lifting type-definition-formula

```

```

lemma Abs-formula-inverse [simp]:
  assumes hereditarily-fs t $\alpha$ 
  shows Rep-formula (Abs-formula t $\alpha$ ) = t $\alpha$ 
using assms by (metis Abs-formula-inverse mem-Collect-eq)

```

```

lemma Rep-formula' [simp]: hereditarily-fs (Rep-formula x)
by (metis Rep-formula mem-Collect-eq)

```

Now we lift the permutation operation.

```

instantiation formula :: (type, fs, bn, fs) pt
begin

```

```

  lift-definition permute-formula :: perm  $\Rightarrow$  ('a, 'b, 'c, 'd) formula  $\Rightarrow$  ('a, 'b, 'c, 'd)
  formula
  is permute
  by (fact hereditarily-fs-eqvt)

```

```

instance
by standard (transfer, simp)+

```

```

end

```

The abstraction and representation functions for formulas are equivariant, and they preserve support.

```

lemma Abs-formula-eqvt [simp]:

```

assumes *hereditarily-fs* t_α
shows $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha)$
by (*metis assms eq-onp-same-args permute-formula.abs-eq*)

lemma *supp-Abs-formula* [*simp*]:

assumes *hereditarily-fs* t_α
shows $\text{supp } (\text{Abs-formula } t_\alpha) = \text{supp } t_\alpha$

proof –

{
 fix $p :: \text{perm}$
 have $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha)$
 using *assms* **by** (*metis Abs-formula-eqt*)
 moreover **have** *hereditarily-fs* $(p \cdot t_\alpha)$
 using *assms* **by** (*metis hereditarily-fs-eqt*)
 ultimately **have** $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } t_\alpha \longleftrightarrow p \cdot t_\alpha = t_\alpha$
 using *assms* **by** (*metis Abs-formula-inverse*)
}

then show *?thesis unfolding supp-def* **by** *simp*

qed

lemmas *Rep-formula-eqt* [*eqvt, simp*] = *permute-formula.rep-eq*[*symmetric*]

lemma *supp-Rep-formula* [*simp*]: $\text{supp } (\text{Rep-formula } x) = \text{supp } x$

by (*metis Rep-formula' Rep-formula-inverse supp-Abs-formula*)

lemma *supp-map-bset-Rep-formula* [*simp*]: $\text{supp } (\text{map-bset } \text{Rep-formula } xset) = \text{supp } xset$

proof

have *eqvt* (*map-bset Rep-formula*)
 unfolding *eqvt-def* **by** (*simp add: ext*)
then show $\text{supp } (\text{map-bset } \text{Rep-formula } xset) \subseteq \text{supp } xset$
 by (*fact supp-fun-app-eqt*)

next

{
 fix $a :: \text{atom}$
 have *inj* (*map-bset Rep-formula*)
 by (*metis bset.inj-map Rep-formula-inject injI*)
 then have $\bigwedge x y. x \neq y \implies \text{map-bset } \text{Rep-formula } x \neq \text{map-bset } \text{Rep-formula } y$
 by (*metis inj-eq*)
 then have $\{b. (a \rightleftharpoons b) \cdot xset \neq xset\} \subseteq \{b. (a \rightleftharpoons b) \cdot \text{map-bset } \text{Rep-formula } xset \neq \text{map-bset } \text{Rep-formula } xset\}$ (**is** $?S \subseteq ?T$)
 by *auto*
 then have *infinite* $?S \implies \text{infinite } ?T$
 by (*metis infinite-super*)
}

then show $\text{supp } xset \subseteq \text{supp } (\text{map-bset } \text{Rep-formula } xset)$

unfolding *supp-def* **by** *auto*

qed

Formulas are in fact finitely supported.

instance *formula* :: (*type*, *fs*, *bn*, *fs*) *fs*
by *standard* (*metis Rep-formula' hereditarily-fs-implies-finite-supp supp-Rep-formula*)

12.7 Constructors for infinitary formulas

We lift the constructors for trees (modulo α -equivalence) to infinitary formulas. Since $Conj_\alpha$ does not necessarily yield a (hereditarily) finitely supported tree when applied to a (potentially infinite) set of (hereditarily) finitely supported trees, we cannot use Isabelle's **lift_definition** to define $Conj$. Instead, theorems about terms of the form $Conj\ xset$ will usually carry an assumption that $xset$ is finitely supported.

definition $Conj$:: (*'idx*, *'pred*, *'act*, *'eff*) *formula set* [*'idx*] \Rightarrow (*'idx*, *'pred*::*fs*, *'act*::*bn*, *'eff*::*fs*) *formula* **where**
 $Conj\ xset = Abs\text{-formula}\ (Conj_\alpha\ (map\text{-bset}\ Rep\text{-formula}\ xset))$

lemma *finite-supp-implies-hereditarily-fs-Conj $_\alpha$* [*simp*]:
assumes *finite* (*supp* *xset*)
shows *hereditarily-fs* ($Conj_\alpha\ (map\text{-bset}\ Rep\text{-formula}\ xset)$)
proof (*rule hereditarily-fs.Conj $_\alpha$*)
show *finite* (*supp* ($map\text{-bset}\ Rep\text{-formula}\ xset$))
using *assms* **by** (*metis supp-map-bset-Rep-formula*)
next
fix t_α **assume** $t_\alpha \in set\text{-bset}\ (map\text{-bset}\ Rep\text{-formula}\ xset)$
then show *hereditarily-fs* t_α
by (*auto simp add: bset.set-map*)
qed

lemma *Conj-rep-eq*:
assumes *finite* (*supp* *xset*)
shows $Rep\text{-formula}\ (Conj\ xset) = Conj_\alpha\ (map\text{-bset}\ Rep\text{-formula}\ xset)$
using *assms* **unfolding** *Conj-def* **by** *simp*

lift-definition Not :: (*'idx*, *'pred*, *'act*, *'eff*) *formula* \Rightarrow (*'idx*, *'pred*::*fs*, *'act*::*bn*, *'eff*::*fs*) *formula* **is**
 Not_α
by (*fact hereditarily-fs.Not $_\alpha$*)

lift-definition $Pred$:: *'eff* \Rightarrow *'pred* \Rightarrow (*'idx*, *'pred*::*fs*, *'act*::*bn*, *'eff*::*fs*) *formula* **is**
 $Pred_\alpha$
by (*fact hereditarily-fs.Pred $_\alpha$*)

lift-definition Act :: *'eff* \Rightarrow *'act* \Rightarrow (*'idx*, *'pred*, *'act*, *'eff*) *formula* \Rightarrow (*'idx*, *'pred*::*fs*, *'act*::*bn*, *'eff*::*fs*) *formula* **is**
 Act_α
by (*fact hereditarily-fs.Act $_\alpha$*)

The lifted constructors are equivariant (in the case of $Conj$, on finitely sup-

ported arguments).

lemma *Conj-eqvt* [*simp*]:
assumes *finite* (*supp xset*)
shows $p \cdot \text{Conj } xset = \text{Conj } (p \cdot xset)$
using *assms unfolding Conj-def by simp*

lemma *Not-eqvt* [*eqvt, simp*]: $p \cdot \text{Not } x = \text{Not } (p \cdot x)$
by *transfer simp*

lemma *Pred-eqvt* [*eqvt, simp*]: $p \cdot \text{Pred } f \ \varphi = \text{Pred } (p \cdot f) \ (p \cdot \varphi)$
by *transfer simp*

lemma *Act-eqvt* [*eqvt, simp*]: $p \cdot \text{Act } f \ \alpha \ x = \text{Act } (p \cdot f) \ (p \cdot \alpha) \ (p \cdot x)$
by *transfer simp*

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows $\text{supp } (\text{Conj } xset) = \text{supp } xset$
using *assms unfolding Conj-def by simp*

lemma *supp-Not* [*simp*]: $\text{supp } (\text{Not } x) = \text{supp } x$
by (*metis Not.rep-eq supp-Not_α supp-Rep-formula*)

lemma *supp-Pred* [*simp*]: $\text{supp } (\text{Pred } f \ \varphi) = \text{supp } f \cup \text{supp } \varphi$
by (*metis Pred.rep-eq supp-Pred_α supp-Rep-formula*)

lemma *supp-Act* [*simp*]: $\text{supp } (\text{Act } f \ \alpha \ x) = \text{supp } f \cup (\text{supp } \alpha \cup \text{supp } x - \text{bn } \alpha)$
by (*metis Act.rep-eq finite-supp supp-Act_α supp-Rep-formula*)

The lifted constructors are injective (partially for *Act*).

lemma *Conj-eq-iff* [*simp*]:
assumes *finite* (*supp xset1*) **and** *finite* (*supp xset2*)
shows $\text{Conj } xset1 = \text{Conj } xset2 \iff xset1 = xset2$
using *assms*
by (*metis (erased, hide-lams) Conj_α-eq-iff Conj-rep-eq Rep-formula-inverse injI inj-eq bset.inj-map*)

lemma *Not-eq-iff* [*simp*]: $\text{Not } x1 = \text{Not } x2 \iff x1 = x2$
by (*metis Not.rep-eq Not_α-eq-iff Rep-formula-inverse*)

lemma *Pred-eq-iff* [*simp*]: $\text{Pred } f1 \ \varphi1 = \text{Pred } f2 \ \varphi2 \iff f1 = f2 \wedge \varphi1 = \varphi2$
by (*metis Pred.rep-eq Pred_α-eq-iff*)

lemma *Act-eq-iff*: $\text{Act } f1 \ \alpha1 \ x1 = \text{Act } f2 \ \alpha2 \ x2 \iff \text{Act}_\alpha \ f1 \ \alpha1 \ (\text{Rep-formula } x1) = \text{Act}_\alpha \ f2 \ \alpha2 \ (\text{Rep-formula } x2)$
by (*metis Act.rep-eq Rep-formula-inverse*)

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: $Act\ f1\ \alpha1\ x1 = Act\ f2\ \alpha2\ x2 \longleftrightarrow$
 $f1 = f2 \wedge (\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#^* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#^* p \wedge p \cdot \alpha1 = \alpha2)$
(is ?l \longleftrightarrow ?r)

proof

assume *: ?l
then have $f1 = f2$
 by (*metis Act-eq-iff Act_α-eq-iff alpha-tAct*)
 moreover from * **obtain** p **where** $alpha: (\text{bn } \alpha1, \text{rep-Tree}_\alpha (\text{Rep-formula } x1)) \approx_{set} (=_\alpha) (\text{supp-rel } (=_\alpha)) p (\text{bn } \alpha2, \text{rep-Tree}_\alpha (\text{Rep-formula } x2))$ **and** $eq: (\text{bn } \alpha1, \alpha1) \approx_{set} (=) \text{supp } p (\text{bn } \alpha2, \alpha2)$
 by (*metis Act-eq-iff Act_α-eq-iff alpha-tAct*)
 from $alpha$ **have** $\text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2$
 by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)
 moreover from $alpha$ **have** $(\text{supp } x1 - \text{bn } \alpha1) \#^* p$
 by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)
 moreover from $alpha$ **have** $p \cdot x1 = x2$
 by (*metis Rep-formula-eqvt Rep-formula-inject Tree_α.abs-eq-iff Tree_α-abs-rep alpha-Tree-permute-rep-commute alpha-set.simps*)
 moreover from eq **have** $\text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2$
 by (*metis alpha-set.simps*)
 moreover from eq **have** $(\text{supp } \alpha1 - \text{bn } \alpha1) \#^* p$
 by (*metis alpha-set.simps*)
 moreover from eq **have** $p \cdot \alpha1 = \alpha2$
 by (*simp add: alpha-set.simps*)
 ultimately show ?r
 by *metis*
next
 assume *: ?r
 then have $f1 = f2$
 by *metis*
 moreover from * **obtain** p **where** $1: \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2$
and $2: (\text{supp } x1 - \text{bn } \alpha1) \#^* p$ **and** $3: p \cdot x1 = x2$
 and $4: \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2$ **and** $5: (\text{supp } \alpha1 - \text{bn } \alpha1) \#^* p$
and $6: p \cdot \alpha1 = \alpha2$
 by *metis*
 from $1\ 2\ 3\ 6$ **have** $(\text{bn } \alpha1, \text{rep-Tree}_\alpha (\text{Rep-formula } x1)) \approx_{set} (=_\alpha) (\text{supp-rel } (=_\alpha)) p (\text{bn } \alpha2, \text{rep-Tree}_\alpha (\text{Rep-formula } x2))$
 by (*metis (no-types, lifting) Rep-formula-eqvt alpha-Tree-permute-rep-commute alpha-set.simps bn-eqvt supp-Rep-formula supp-alpha-supp-rel*)
 moreover from $4\ 5\ 6$ **have** $(\text{bn } \alpha1, \alpha1) \approx_{set} (=) \text{supp } p (\text{bn } \alpha2, \alpha2)$
 by (*simp add: alpha-set.simps bn-eqvt*)
 ultimately show $Act\ f1\ \alpha1\ x1 = Act\ f2\ \alpha2\ x2$
 by (*metis Act-eq-iff Act_α-eq-iff alpha-tAct*)
qed

lemma *Act-eq-iff-perm-renaming*: $Act\ f1\ \alpha1\ x1 = Act\ f2\ \alpha2\ x2 \longleftrightarrow$
 $f1 = f2 \wedge (\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#^*$

$p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1)$
 $\#* p \wedge p \cdot \alpha1 = \alpha2 \wedge \text{supp } p \subseteq \text{bn } \alpha1 \cup p \cdot \text{bn } \alpha1$
 (is ?l \longleftrightarrow ?r)

proof

assume ?l then have f1 = f2

by (metis Act-eq-iff-perm)

moreover from ?l obtain p where $p: \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2$
 $\wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2$
 $\wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2$

by (metis Act-eq-iff-perm)

moreover obtain q where $q-p: \forall b \in \text{bn } \alpha1. q \cdot b = p \cdot b$ **and** $\text{supp-}q: \text{supp } q \subseteq \text{bn } \alpha1 \cup p \cdot \text{bn } \alpha1$

by (metis set-renaming-perm2)

have $\text{supp } q \subseteq \text{supp } p$

proof

fix a assume $*$: $a \in \text{supp } q$ **then show** $a \in \text{supp } p$

proof (cases $a \in \text{bn } \alpha1$)

case True then show ?thesis

using $* q-p$ **by** (metis mem-Collect-eq supp-perm)

next

case False then have $a \in p \cdot \text{bn } \alpha1$

using $* \text{supp-}q$ **using** UnE subsetCE **by** blast

with False have $p \cdot a \neq a$

by (metis mem-permute-iff)

then show ?thesis

using fresh-def fresh-perm **by** blast

qed

qed

with p have $(\text{supp } x1 - \text{bn } \alpha1) \#* q$ **and** $(\text{supp } \alpha1 - \text{bn } \alpha1) \#* q$

by (meson fresh-def fresh-star-def subset-iff)+

moreover with p and q-p have $\bigwedge a. a \in \text{supp } \alpha1 \implies q \cdot a = p \cdot a$ **and** $\bigwedge a. a \in \text{supp } x1 \implies q \cdot a = p \cdot a$

by (metis Diff-iff fresh-perm fresh-star-def)+

then have $q \cdot \alpha1 = p \cdot \alpha1$ **and** $q \cdot x1 = p \cdot x1$

by (metis supp-perm-perm-eq)+

ultimately show ?r

using $\text{supp-}q$ **by** (metis bn-eqvt)

next

assume ?r then show ?l

by (meson Act-eq-iff-perm)

qed

The lifted constructors are free (except for Act).

lemma Tree-free [simp]:

shows $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Not } x$

and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Pred } f \ \varphi$

and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Act } f \ \alpha \ x$

and $\text{Not } x \neq \text{Pred } f \ \varphi$

and $\text{Not } x1 \neq \text{Act } f \ \alpha \ x2$

```

and  $Pred\ f1\ \varphi \neq Act\ f2\ \alpha\ x$ 
proof –
  show  $finite\ (supp\ xset) \implies Conj\ xset \neq Not\ x$ 
    by  $(metis\ Conj\ rep\ eq\ Not\ rep\ eq\ Tree_\alpha\ free(1))$ 
next
  show  $finite\ (supp\ xset) \implies Conj\ xset \neq Pred\ f\ \varphi$ 
    by  $(metis\ Conj\ rep\ eq\ Pred\ rep\ eq\ Tree_\alpha\ free(2))$ 
next
  show  $finite\ (supp\ xset) \implies Conj\ xset \neq Act\ f\ \alpha\ x$ 
    by  $(metis\ Conj\ rep\ eq\ Act\ rep\ eq\ Tree_\alpha\ free(3))$ 
next
  show  $Not\ x \neq Pred\ f\ \varphi$ 
    by  $(metis\ Not\ rep\ eq\ Pred\ rep\ eq\ Tree_\alpha\ free(4))$ 
next
  show  $Not\ x1 \neq Act\ f\ \alpha\ x2$ 
    by  $(metis\ Not\ rep\ eq\ Act\ rep\ eq\ Tree_\alpha\ free(5))$ 
next
  show  $Pred\ f1\ \varphi \neq Act\ f2\ \alpha\ x$ 
    by  $(metis\ Pred\ rep\ eq\ Act\ rep\ eq\ Tree_\alpha\ free(6))$ 
qed

```

12.8 F/L -formulas

```

context effect-nominal-ts
begin

```

The predicate *is-FL-formula* will characterise exactly those formulas in a particular set $A^{F/L}$.

```

inductive is-FL-formula :: 'effect first  $\implies$  ('idx, 'pred, 'act, 'effect) formula  $\implies$  bool
```

where

```

  Conj:  $finite\ (supp\ xset) \implies (\bigwedge x. x \in set\ bset\ xset \implies is\ FL\ formula\ F\ x) \implies is\ FL\ formula\ F\ (Conj\ xset)$ 
| Not:  $is\ FL\ formula\ F\ x \implies is\ FL\ formula\ F\ (Not\ x)$ 
| Pred:  $f \in_{fs}\ F \implies is\ FL\ formula\ F\ (Pred\ f\ \varphi)$ 
| Act:  $f \in_{fs}\ F \implies bn\ \alpha\ \sharp^*\ (F, f) \implies is\ FL\ formula\ (L\ (\alpha, F, f))\ x \implies is\ FL\ formula\ F\ (Act\ f\ \alpha\ x)$ 

```

```

abbreviation in-A :: 'idx, 'pred, 'act, 'effect) formula  $\implies$  'effect first  $\implies$  bool
```

```

   $(- \in \mathcal{A}[-] [51, 0] 50)$  where
```

```

   $x \in \mathcal{A}[F] \equiv is\ FL\ formula\ F\ x$ 
```

```

declare is-FL-formula.induct [case-names Conj Not Pred Act, induct type: formula]

```

```

lemma is-FL-formula-eqv [eqvt]:  $x \in \mathcal{A}[F] \implies p \cdot x \in \mathcal{A}[p \cdot F]$ 
```

```

proof (erule is-FL-formula.induct)
```

```

  fix xset :: 'a, 'pred, 'act, 'effect) formula set [a] and F
```

```

  assume 1:  $finite\ (supp\ xset)$  and 2:  $\bigwedge x. x \in set\ bset\ xset \implies p \cdot x \in \mathcal{A}[p \cdot F]$ 
```

```

  from 1 have  $finite\ (supp\ (p \cdot xset))$ 
```

```

    by (metis permute-finite supp-eqvt)
  moreover from 2 have  $\bigwedge x. x \in \text{set-bset } (p \cdot \text{xset}) \implies x \in \mathcal{A}[p \cdot F]$ 
    by (metis (mono-tags) imageE permute-set-eq-image set-bset-eqvt)
  ultimately show  $p \cdot \text{Conj xset} \in \mathcal{A}[p \cdot F]$ 
    using 1 by (simp add: Conj)
next
fix F and x :: ('a, 'pred, 'act, 'effect) formula
assume  $p \cdot x \in \mathcal{A}[p \cdot F]$ 
then show  $p \cdot \text{Not } x \in \mathcal{A}[p \cdot F]$ 
  by (simp add: Not)
next
fix f and F :: 'effect first and  $\varphi$ 
assume  $f \in_{fs} F$ 
then show  $p \cdot \text{Pred } f \varphi \in \mathcal{A}[p \cdot F]$ 
  by (simp add: Pred)
next
fix f F  $\alpha$  and x :: ('a, 'pred, 'act, 'effect) formula
assume  $f \in_{fs} F$  and  $bn \alpha \#* (F, f)$  and  $p \cdot x \in \mathcal{A}[p \cdot L (\alpha, F, f)]$ 
then show  $p \cdot \text{Act } f \alpha x \in \mathcal{A}[p \cdot F]$ 
  by (metis (mono-tags, lifting) Act Act-eqvt L-eqvt' Pair-eqvt bn-eqvt fresh-star-permute-iff
member-fs-set-permute-iff)
qed

end

```

12.9 Induction over infinitary formulas

12.10 Strong induction over infinitary formulas

```

end
theory FL-Validity
imports
  FL-Transition-System
  FL-Formula
begin

```

13 Validity With Effects

The following is needed to prove termination of *FL-validTree*.

definition *alpha-Tree-rel* **where**
 $\text{alpha-Tree-rel} \equiv \{(x, y). x =_{\alpha} y\}$

lemma *alpha-Tree-relI* [*simp*]:
assumes $x =_{\alpha} y$ **shows** $(x, y) \in \text{alpha-Tree-rel}$
using *assms* **unfolding** *alpha-Tree-rel-def* **by** *simp*

lemma *alpha-Tree-relE*:
assumes $(x, y) \in \text{alpha-Tree-rel}$ **and** $x =_{\alpha} y \implies P$

shows P
using *assms unfolding alpha-Tree-rel-def by simp*

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
wf (alpha-Tree-rel O hull-rel O Tree-wf)

proof (*rule wf-relcomp-compatible*)
show *wf (hull-rel O Tree-wf)*
by (*metis Tree-wf-eqt' wf-Tree-wf wf-hull-rel-relcomp*)

next
show (*hull-rel O Tree-wf*) *O alpha-Tree-rel* \subseteq *alpha-Tree-rel O (hull-rel O Tree-wf)*

proof
fix $x :: ('e, 'f, 'g, 'h) \text{Tree} \times ('e, 'f, 'g, 'h) \text{Tree}$
assume $x \in (\text{hull-rel } O \text{ Tree-wf}) \ O \ \text{alpha-Tree-rel}$
then obtain $x_1 \ x_2 \ x_3 \ x_4$ **where** $x: x = (x_1, x_4)$ **and** $1: (x_1, x_2) \in \text{hull-rel}$ **and**
 $2: (x_2, x_3) \in \text{Tree-wf}$ **and** $3: (x_3, x_4) \in \text{alpha-Tree-rel}$
by *auto*
from 2 **have** $(x_1, x_4) \in \text{alpha-Tree-rel } O \ \text{hull-rel } O \ \text{Tree-wf}$
using 1 **and** 3 **proof** (*induct rule: Tree-wf.induct*)
— *tConj*
fix t **and** $tset :: ('e, 'f, 'g, 'h) \text{Tree set}[e]$
assume $*$: $t \in \text{set-bset } tset$ **and** $**$: $(x_1, t) \in \text{hull-rel}$ **and** $***$: $(tConj \ tset,$
 $x_4) \in \text{alpha-Tree-rel}$
from $**$ **obtain** p **where** $x_1: x_1 = p \cdot t$
using *hull-rel.cases by blast*
from $***$ **have** $tConj \ tset =_{\alpha} x_4$
by (*rule alpha-Tree-relE*)
then obtain $tset'$ **where** $x_4: x_4 = tConj \ tset'$ **and** $\text{rel-bset } (=_{\alpha}) \ tset \ tset'$
by (*cases x4*) *simp-all*
with $*$ **obtain** t' **where** $t': t' \in \text{set-bset } tset'$ **and** $t =_{\alpha} t'$
by (*metis rel-bset.rep-eq rel-set-def*)
with x_1 **have** $(x_1, p \cdot t') \in \text{alpha-Tree-rel}$
by (*metis Tree_{\alpha}.abs-eq-iff alpha-Tree-relI permute-Tree_{\alpha}.abs-eq*)
moreover have $(p \cdot t', t') \in \text{hull-rel}$
by (*rule hull-rel.intros*)
moreover from x_4 **and** t' **have** $(t', x_4) \in \text{Tree-wf}$
by (*simp add: Tree-wf.intros(1)*)
ultimately show $(x_1, x_4) \in \text{alpha-Tree-rel } O \ \text{hull-rel } O \ \text{Tree-wf}$
by *auto*

next
— *tNot*
fix t
assume $*$: $(x_1, t) \in \text{hull-rel}$ **and** $**$: $(tNot \ t, x_4) \in \text{alpha-Tree-rel}$
from $*$ **obtain** p **where** $x_1: x_1 = p \cdot t$
using *hull-rel.cases by blast*
from $**$ **have** $tNot \ t =_{\alpha} x_4$
by (*rule alpha-Tree-relE*)
then obtain t' **where** $x_4: x_4 = tNot \ t'$ **and** $t =_{\alpha} t'$
by (*cases x4*) *simp-all*
with x_1 **have** $(x_1, p \cdot t') \in \text{alpha-Tree-rel}$

```

    by (metis Tree $\alpha$ .abs-eq-iff alpha-Tree-relI permute-Tree $\alpha$ .abs-eq x1)
  moreover have (p · t', t') ∈ hull-rel
    by (rule hull-rel.intros)
  moreover from x4 have (t', x4) ∈ Tree-wf
    using Tree-wf.intros(2) by blast
  ultimately show (x1,x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
    by auto
next
— tAct
fix f α t
assume *: (x1,t) ∈ hull-rel and **: (tAct f α t, x4) ∈ alpha-Tree-rel
from * obtain p where x1: x1 = p · t
  using hull-rel.cases by blast
from ** have tAct f α t = $\alpha$  x4
  by (rule alpha-Tree-relE)
then obtain q t' where x4: x4 = tAct f (q · α) t' and q · t = $\alpha$  t'
  by (cases x4) (auto simp add: alpha-set)
with x1 have (x1, p · -q · t') ∈ alpha-Tree-rel
by (metis Tree $\alpha$ .abs-eq-iff alpha-Tree-relI permute-Tree $\alpha$ .abs-eq permute-minus-cancel(1))
moreover have (p · -q · t', t') ∈ hull-rel
  by (metis hull-rel.simps permute-plus)
moreover from x4 have (t', x4) ∈ Tree-wf
  by (simp add: Tree-wf.intros(3))
ultimately show (x1,x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
  by auto
qed
with x show x ∈ alpha-Tree-rel O hull-rel O Tree-wf
  by simp
qed
qed

```

```

lemma alpha-Tree-rel-relcomp-trivialI [simp]:
  assumes (x, y) ∈ R
  shows (x, y) ∈ alpha-Tree-rel O R
using assms unfolding alpha-Tree-rel-def
by (metis Tree $\alpha$ .abs-eq-iff case-prodI mem-Collect-eq relcomp.relcompI)

```

```

lemma alpha-Tree-rel-relcompI [simp]:
  assumes x = $\alpha$  x' and (x', y) ∈ R
  shows (x, y) ∈ alpha-Tree-rel O R
using assms unfolding alpha-Tree-rel-def
by (metis case-prodI mem-Collect-eq relcomp.relcompI)

```

13.1 Validity for infinitely branching trees

```

context effect-nominal-ts
begin

```

Since we defined formulas via a manual quotient construction, we also need to define validity via lifting from the underlying type of infinitely branching

trees. We cannot use **nominal_function** because that generates proof obligations where, for formulas of the form $Conj\ xset$, the assumption that $xset$ has finite support is missing.

```

declare conj-cong [fundef-cong]

function (sequential) FL-valid-Tree :: 'state  $\Rightarrow$  ('idx,'pred,'act,'effect) Tree  $\Rightarrow$ 
bool where
  FL-valid-Tree P (tConj tset)  $\longleftrightarrow$  ( $\forall t \in set-bset\ tset.$  FL-valid-Tree P t)
| FL-valid-Tree P (tNot t)  $\longleftrightarrow$   $\neg$  FL-valid-Tree P t
| FL-valid-Tree P (tPred f  $\varphi$ )  $\longleftrightarrow$   $\langle f \rangle P \vdash \varphi$ 
| FL-valid-Tree P (tAct f  $\alpha$  t)  $\longleftrightarrow$  ( $\exists \alpha' t' P'.$  tAct f  $\alpha$  t  $=_{\alpha}$  tAct f  $\alpha'$  t'  $\wedge \langle f \rangle P$ 
 $\rightarrow \langle \alpha', P' \rangle \wedge$  FL-valid-Tree P' t')
by pat-completeness auto
termination proof
  let ?R = inv-image (alpha-Tree-rel O hull-rel O Tree-wf) snd
  {
    show wf ?R
    by (metis wf-alpha-Tree-rel-hull-rel-Tree-wf wf-inv-image)
  next
    fix P :: 'state and tset :: ('idx,'pred,'act,'effect) Tree set['idx] and t
    assume t  $\in$  set-bset tset then show ((P, t), (P, tConj tset))  $\in$  ?R
    by (simp add: Tree-wf.intros(1))
  next
    fix P :: 'state and t :: ('idx,'pred,'act,'effect) Tree
    show ((P, t), (P, tNot t))  $\in$  ?R
    by (simp add: Tree-wf.intros(2))
  next
    fix P1 P2 :: 'state and f and  $\alpha 1$   $\alpha 2$  and t1 t2 :: ('idx,'pred,'act,'effect) Tree
    assume tAct f  $\alpha 1$  t1  $=_{\alpha}$  tAct f  $\alpha 2$  t2
    then obtain p where t2  $=_{\alpha}$  p  $\cdot$  t1
    by (auto simp add: alphas) (metis alpha-Tree-symp sympE)
    then show ((P2, t2), (P1, tAct f  $\alpha 1$  t1))  $\in$  ?R
    by (simp add: Tree-wf.intros(3))
  }
qed

```

$FL\text{-valid-Tree}$ is equivariant.

```

lemma FL-valid-Tree-eqvt': FL-valid-Tree P t  $\longleftrightarrow$  FL-valid-Tree (p  $\cdot$  P) (p  $\cdot$  t)
proof (induction P t rule: FL-valid-Tree.induct)
  case (1 P tset) show ?case
  proof
    assume *: FL-valid-Tree P (tConj tset)
    {
      fix t
      assume t  $\in$  p  $\cdot$  set-bset tset
      with 1.IH and * have FL-valid-Tree (p  $\cdot$  P) t
      by (metis (no-types, lifting) imageE permute-set-eq-image FL-valid-Tree.simps(1))
    }
    then show FL-valid-Tree (p  $\cdot$  P) (p  $\cdot$  tConj tset)
  qed

```

```

    by simp
  next
  assume *: FL-valid-Tree (p · P) (p · tConj tset)
  {
    fix t
    assume t ∈ set-bset tset
    with 1.IH and * have FL-valid-Tree P t
    by (metis mem-permute-iff permute-Tree-tConj set-bset-eqvt FL-valid-Tree.simps(1))
  }
  then show FL-valid-Tree P (tConj tset)
    by simp
  qed
next
case 2 then show ?case by simp
next
case 3 show ?case by simp (metis effect-apply-eqvt' permute-minus-cancel(2)
satisfies-eqvt)
next
case (4 P f α t) show ?case
proof
  assume FL-valid-Tree P (tAct f α t)
  then obtain α' t' P' where *: tAct f α t =α tAct f α' t' ∧ ⟨f⟩P → ⟨α', P'⟩
  ∧ FL-valid-Tree P' t'
  by auto
  with 4.IH have FL-valid-Tree (p · P') (p · t')
  by blast
  moreover from * have p · ⟨f⟩P → ⟨p · α', p · P'⟩
  by (metis transition-eqvt')
  moreover from * have p · tAct f α t =α tAct (p · f) (p · α') (p · t')
  by (metis alpha-Tree-eqvt permute-Tree.simps(4))
  ultimately show FL-valid-Tree (p · P) (p · tAct f α t)
  by auto
next
assume FL-valid-Tree (p · P) (p · tAct f α t)
then obtain α' t' P' where *: p · tAct f α t =α tAct (p · f) α' t' ∧ (p ·
⟨f⟩P) → ⟨α', P'⟩ ∧ FL-valid-Tree P' t'
by auto
then have eq: tAct f α t =α tAct f (-p · α') (-p · t')
by (metis alpha-Tree-eqvt permute-Tree.simps(4) permute-minus-cancel(2))
moreover from * have ⟨f⟩P → ⟨-p · α', -p · P'⟩
by (metis permute-minus-cancel(2) transition-eqvt')
moreover with 4.IH have FL-valid-Tree (-p · P') (-p · t')
using eq and * by simp
ultimately show FL-valid-Tree P (tAct f α t)
by auto
qed
qed

```

lemma *FL-valid-Tree-eqvt* [eqvt]:

assumes $FL\text{-valid-Tree } P \ t$ **shows** $FL\text{-valid-Tree } (p \cdot P) \ (p \cdot t)$
using *assms* **by** (*metis FL-valid-Tree-eqvt*)

α -equivalent trees validate the same states.

lemma *alpha-Tree-FL-valid-Tree*:
assumes $t1 =_\alpha t2$
shows $FL\text{-valid-Tree } P \ t1 \longleftrightarrow FL\text{-valid-Tree } P \ t2$
using *assms* **proof** (*induction t1 t2 arbitrary: P rule: alpha-Tree-induct*)
case *tConj* **then show** *?case*
by *auto* (*metis (mono-tags) rel-bset.rep-eq rel-set-def*)
next
case (*tAct f1 $\alpha1$ t1 f2 $\alpha2$ t2*) **show** *?case*
proof
assume $FL\text{-valid-Tree } P \ (tAct \ f1 \ \alpha1 \ t1)$
then obtain $\alpha' \ t' \ P'$ **where** $tAct \ f1 \ \alpha1 \ t1 =_\alpha tAct \ f1 \ \alpha' \ t' \ \wedge \langle f1 \rangle P \rightarrow$
 $\langle \alpha', P' \rangle \wedge FL\text{-valid-Tree } P' \ t'$
by *auto*
moreover from *tAct.hyps* **have** $tAct \ f1 \ \alpha1 \ t1 =_\alpha tAct \ f2 \ \alpha2 \ t2$
using *alpha-tAct* **by** *blast*
ultimately show $FL\text{-valid-Tree } P \ (tAct \ f2 \ \alpha2 \ t2)$
using *tAct.hyps* **by** (*metis Tree $_\alpha$.abs-eq-iff FL-valid-Tree.simps(4)*)
next
assume $FL\text{-valid-Tree } P \ (tAct \ f2 \ \alpha2 \ t2)$
then obtain $\alpha' \ t' \ P'$ **where** $tAct \ f2 \ \alpha2 \ t2 =_\alpha tAct \ f2 \ \alpha' \ t' \ \wedge \langle f2 \rangle P \rightarrow$
 $\langle \alpha', P' \rangle \wedge FL\text{-valid-Tree } P' \ t'$
by *auto*
moreover from *tAct.hyps* **have** $tAct \ f1 \ \alpha1 \ t1 =_\alpha tAct \ f2 \ \alpha2 \ t2$
using *alpha-tAct* **by** *blast*
ultimately show $FL\text{-valid-Tree } P \ (tAct \ f1 \ \alpha1 \ t1)$
using *tAct.hyps* **by** (*metis Tree $_\alpha$.abs-eq-iff FL-valid-Tree.simps(4)*)
qed
qed *simp-all*

13.2 Validity for trees modulo α -equivalence

lift-definition $FL\text{-valid-Tree}_\alpha :: 'state \Rightarrow ('idx, 'pred, 'act, 'effect) \ Tree_\alpha \Rightarrow bool$
is

FL-valid-Tree
by (*fact alpha-Tree-FL-valid-Tree*)

lemma *FL-valid-Tree $_\alpha$ -eqvt* [*eqvt*]:
assumes $FL\text{-valid-Tree}_\alpha \ P \ t$ **shows** $FL\text{-valid-Tree}_\alpha \ (p \cdot P) \ (p \cdot t)$
using *assms* **by** *transfer* (*fact FL-valid-Tree-eqvt*)

lemma *FL-valid-Tree $_\alpha$ -Conj $_\alpha$* [*simp*]: $FL\text{-valid-Tree}_\alpha \ P \ (Conj_\alpha \ tset_\alpha) \longleftrightarrow (\forall t_\alpha \in set\text{-bset} \ tset_\alpha. FL\text{-valid-Tree}_\alpha \ P \ t_\alpha)$
proof –
have $FL\text{-valid-Tree } P \ (rep\text{-Tree}_\alpha \ (abs\text{-Tree}_\alpha \ (tConj \ (map\text{-bset} \ rep\text{-Tree}_\alpha \ tset_\alpha))))$
 $\longleftrightarrow FL\text{-valid-Tree } P \ (tConj \ (map\text{-bset} \ rep\text{-Tree}_\alpha \ tset_\alpha))$

by (*metis Tree_α-rep-abs alpha-Tree-FL-valid-Tree*)
then show *?thesis*
by (*simp add: FL-valid-Tree_α-def Conj_α-def map-bset.rep-eq*)
qed

lemma *FL-valid-Tree_α-Not_α* [*simp*]: *FL-valid-Tree_α P (Not_α t_α) ↔ ¬ FL-valid-Tree_α P t_α*
by *transfer simp*

lemma *FL-valid-Tree_α-Pred_α* [*simp*]: *FL-valid-Tree_α P (Pred_α f φ) ↔ ⟨f⟩P ⊢ φ*
by *transfer simp*

lemma *FL-valid-Tree_α-Act_α* [*simp*]: *FL-valid-Tree_α P (Act_α f α t_α) ↔ (∃ α' t_{α'} P'. Act_α f α t_α = Act_α f α' t_{α'} ∧ ⟨f⟩P → ⟨α', P'⟩ ∧ FL-valid-Tree_α P' t_{α'})*
proof

assume *FL-valid-Tree_α P (Act_α f α t_α)*
moreover have *Act_α f α t_α = abs-Tree_α (tAct f α (rep-Tree_α t_α))*
by (*metis Act_α.abs-eq Tree_α-abs-rep*)
ultimately show *∃ α' t_{α'} P'. Act_α f α t_α = Act_α f α' t_{α'} ∧ ⟨f⟩P → ⟨α', P'⟩*
∧ *FL-valid-Tree_α P' t_{α'}*
by (*metis Act_α.abs-eq Tree_α.abs-eq-iff FL-valid-Tree.simps(4) FL-valid-Tree_α.abs-eq*)
next
assume *∃ α' t_{α'} P'. Act_α f α t_α = Act_α f α' t_{α'} ∧ ⟨f⟩P → ⟨α', P'⟩ ∧ FL-valid-Tree_α P' t_{α'}*
moreover have *∧ α' t_{α'}. Act_α f α' t_{α'} = abs-Tree_α (tAct f α' (rep-Tree_α t_{α'}))*
by (*metis Act_α.abs-eq Tree_α-abs-rep*)
ultimately show *FL-valid-Tree_α P (Act_α f α t_α)*
by (*metis Tree_α.abs-eq-iff FL-valid-Tree.simps(4) FL-valid-Tree_α.abs-eq FL-valid-Tree_α.rep-eq*)
qed

13.3 Validity for infinitary formulas

lift-definition *FL-valid* :: *'state ⇒ ('idx, 'pred, 'act, 'effect) formula ⇒ bool* (**infix** \models 70) is
FL-valid-Tree_α
.

lemma *FL-valid-eqvt* [*eqvt*]:
assumes *P ⊨ x* **shows** *(p · P) ⊨ (p · x)*
using *assms* **by** *transfer (metis FL-valid-Tree_α-eqvt)*

lemma *FL-valid-Conj* [*simp*]:
assumes *finite (supp xset)*
shows *P ⊨ Conj xset ↔ (∀ x ∈ set-bset xset. P ⊨ x)*
using *assms* **by** (*simp add: FL-valid-def Conj-def map-bset.rep-eq*)

lemma *FL-valid-Not* [*simp*]: *P ⊨ Not x ↔ ¬ P ⊨ x*
by *transfer simp*

lemma *FL-valid-Pred* [*simp*]: $P \models \text{Pred } f \ \varphi \longleftrightarrow \langle f \rangle P \vdash \varphi$
by *transfer simp*

lemma *FL-valid-Act*: $P \models \text{Act } f \ \alpha \ x \longleftrightarrow (\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x')$

proof

assume $P \models \text{Act } f \ \alpha \ x$

moreover have *Rep-formula* (*Abs-formula* ($\text{Act}_\alpha f \ \alpha \ (\text{Rep-formula } x)$)) = $\text{Act}_\alpha f \ \alpha \ (\text{Rep-formula } x)$

by (*metis Act.rep-eq Rep-formula-inverse*)

ultimately show $\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$

by (*auto simp add: FL-valid-def Act-def*) (*metis Abs-formula-inverse Rep-formula' hereditarily-fs-alpha-renaming*)

next

assume $\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$

then show $P \models \text{Act } f \ \alpha \ x$

by (*metis Act.rep-eq FL-valid.rep-eq FL-valid-Tree $_\alpha$ -Act $_\alpha$*)

qed

The binding names in the alpha-variant that witnesses validity may be chosen fresh for any finitely supported context.

lemma *FL-valid-Act-strong*:

assumes *finite* (*supp* X)

shows $P \models \text{Act } f \ \alpha \ x \longleftrightarrow (\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \ \#\!*\ X)$

proof

assume $P \models \text{Act } f \ \alpha \ x$

then obtain $\alpha' \ x' \ P'$ **where** *eq*: $\text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x'$ **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$

by (*metis FL-valid-Act*)

have *finite* (*bn* α')

by (*fact bn-finite*)

moreover note (*finite* (*supp* X))

moreover have *finite* (*supp* (*supp* $x' - \text{bn } \alpha'$, *supp* $\alpha' - \text{bn } \alpha'$, $\langle \alpha', P' \rangle$))

by (*simp add: supp-Pair finite-sets-supp finite-supp*)

moreover have *bn* $\alpha' \ \#\!*$ (*supp* $x' - \text{bn } \alpha'$, *supp* $\alpha' - \text{bn } \alpha'$, $\langle \alpha', P' \rangle$)

by (*simp add: atom-fresh-star-disjoint finite-supp fresh-star-Pair*)

ultimately obtain p **where** *fresh-X*: $(p \cdot \text{bn } \alpha') \ \#\!*\ X$ **and** *fresh-p*: *supp* (*supp* $x' - \text{bn } \alpha'$, *supp* $\alpha' - \text{bn } \alpha'$, $\langle \alpha', P' \rangle$) $\#\!*\ p$

by (*metis at-set-avoiding2*)

from *fresh-p* **have** *supp* (*supp* $x' - \text{bn } \alpha'$) $\#\!*\ p$ **and** *supp* (*supp* $\alpha' - \text{bn } \alpha'$) $\#\!*\ p$ **and** *1*: *supp* $\langle \alpha', P' \rangle \ \#\!*\ p$

by (*meson fresh-Pair fresh-def fresh-star-def*)**+**

then have *2*: (*supp* $x' - \text{bn } \alpha'$) $\#\!*\ p$ **and** *3*: (*supp* $\alpha' - \text{bn } \alpha'$) $\#\!*\ p$

by (*simp add: finite-supp supp-finite-atom-set*)**+**

moreover from 2 have $\text{supp } (p \cdot x') - \text{bn } (p \cdot \alpha') = \text{supp } x' - \text{bn } \alpha'$
by (*metis Diff-eqvt atom-set-perm-eg bn-eqvt supp-eqvt*)
moreover from 3 have $\text{supp } (p \cdot \alpha') - \text{bn } (p \cdot \alpha') = \text{supp } \alpha' - \text{bn } \alpha'$
by (*metis (no-types, hide-lams) Diff-eqvt atom-set-perm-eg bn-eqvt supp-eqvt*)
ultimately have $\text{Act } f \ \alpha' \ x' = \text{Act } f \ (p \cdot \alpha') \ (p \cdot x')$
by (*auto simp add: Act-eg-iff-perm*)

moreover from 1 have $\langle p \cdot \alpha', p \cdot P' \rangle = \langle \alpha', P' \rangle$
by (*metis abs-residual-pair-eqvt supp-perm-eg*)

ultimately show $\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \ \sharp^* X$

using eq and trans and valid and fresh-X by (*metis bn-eqvt FL-valid-eqvt*)

next

assume $\exists \alpha' \ x' \ P'. \text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \ \sharp^* X$

then show $P \models \text{Act } f \ \alpha \ x$ **by** (*metis FL-valid-Act*)

qed

lemma *FL-valid-Act-fresh*:

assumes $\text{bn } \alpha \ \sharp^* \langle f \rangle P$

shows $P \models \text{Act } f \ \alpha \ x \longleftrightarrow (\exists P'. \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x)$

proof

assume $P \models \text{Act } f \ \alpha \ x$

moreover have *finite* ($\text{supp } (\langle f \rangle P)$)

by (*fact finite-supp*)

ultimately obtain $\alpha' \ x' \ P'$ **where**

eq: $\text{Act } f \ \alpha \ x = \text{Act } f \ \alpha' \ x'$ **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$

and *fresh*: $\text{bn } \alpha' \ \sharp^* \langle f \rangle P$

by (*metis FL-valid-Act-strong*)

from eq obtain p **where** $p \cdot \alpha = \alpha'$ **and** $p \cdot x = x'$ **and** $\text{supp } p \subseteq \text{bn } \alpha \cup p \cdot \text{bn } \alpha$

by (*metis Act-eg-iff-perm-renaming*)

from assms and fresh have $(\text{bn } \alpha \cup p \cdot \text{bn } \alpha) \ \sharp^* \langle f \rangle P$

using $p \cdot \alpha$ **by** (*metis bn-eqvt fresh-star-Un*)

then have $\text{supp } p \ \sharp^* \langle f \rangle P$

using $\text{supp } p$ **by** (*metis fresh-star-def subset-eg*)

then have $p \cdot P: -p \cdot \langle f \rangle P = \langle f \rangle P$

by (*metis perm-supp-eg supp-minus-perm*)

from trans have $\langle f \rangle P \rightarrow \langle \alpha, -p \cdot P' \rangle$

using $p \cdot P \ p \cdot \alpha$ **by** (*metis permute-minus-cancel(1) transition-eqvt'*)

moreover from valid have $-p \cdot P' \models x$

using $p \cdot x$ **by** (*metis permute-minus-cancel(1) FL-valid-egvt*)

ultimately show $\exists P'. \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x$

by *meson*


```

next
  assume  $\exists P'. \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x$ 
  then show  $P \models \text{Act } f \ \alpha \ x$ 
  by (metis FL-valid-Act)
qed

end

```

```

end
theory FL-Logical-Equivalence
imports
  FL-Validity
begin

```

14 (Strong) Logical Equivalence

The definition of formulas is parametric in the index type, but from now on we want to work with a fixed (sufficiently large) index type.

```

locale indexed-effect-nominal-ts = effect-nominal-ts satisfies transition effect-apply
  for satisfies :: 'state::fs  $\Rightarrow$  'pred::fs  $\Rightarrow$  bool (infix  $\vdash$  70)
  and transition :: 'state  $\Rightarrow$  ('act::bn, 'state) residual  $\Rightarrow$  bool (infix  $\rightarrow$  70)
  and effect-apply :: 'effect::fs  $\Rightarrow$  'state  $\Rightarrow$  'state ( $\langle \cdot \rangle$ - [0,101] 100) +
  assumes card-idx-perm:  $|UNIV::perm \text{ set}| < o \ |UNIV::'idx \text{ set}|$ 
    and card-idx-state:  $|UNIV::'state \text{ set}| < o \ |UNIV::'idx \text{ set}|$ 
begin

```

```

  definition FL-logically-equivalent :: 'effect first  $\Rightarrow$  'state  $\Rightarrow$  'state  $\Rightarrow$  bool where
    FL-logically-equivalent F P Q  $\equiv$ 
       $\forall x::('idx, 'pred, 'act, 'effect) \text{ formula. } x \in \mathcal{A}[F] \longrightarrow (P \models x \longleftrightarrow Q \models x)$ 

```

We could (but didn't need to) prove that this defines an equivariant equivalence relation.

```
end
```

```

end
theory FL-Bisimilarity-Implies-Equivalence
imports
  FL-Logical-Equivalence
begin

```

15 F/L -Bisimilarity Implies Logical Equivalence

```

context indexed-effect-nominal-ts
begin

```

```

  lemma FL-bisimilarity-implies-equivalence-Act:
    assumes  $f \in_{fs} F$ 

```

and $bn \alpha \#* (F, f)$
and $x \in \mathcal{A}[L(\alpha, F, f)]$
and $\bigwedge P Q. P \sim.[L(\alpha, F, f)] Q \implies P \models x \longleftrightarrow Q \models x$
and $P \sim.[F] Q$
and $P \models Act f \alpha x$
shows $Q \models Act f \alpha x$
proof –
have $finite (supp (\langle f \rangle Q, F, f))$
by (*fact finite-supp*)
with $\langle P \models Act f \alpha x \rangle$ **obtain** $\alpha' x' P'$ **where** $eq: Act f \alpha x = Act f \alpha' x'$
and $trans: \langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** $valid: P' \models x'$ **and** $fresh: bn \alpha' \#* (\langle f \rangle Q, F, f)$
by (*metis FL-valid-Act-strong*)

from $\langle P \sim.[F] Q \rangle$ **and** $\langle f \in_{fs} F \rangle$ **and** $fresh$ **and** $trans$ **obtain** Q' **where**
 $trans': \langle f \rangle Q \rightarrow \langle \alpha', Q' \rangle$ **and** $bisim': P' \sim.[L(\alpha', F, f)] Q'$
by (*metis FL-bisimilar-simulation-step*)

from eq **obtain** p **where** $p\text{-}\alpha: \alpha' = p \cdot \alpha$ **and** $p\text{-}x: x' = p \cdot x$
and $fresh\text{-}p: (supp x - bn \alpha) \#* p \wedge (supp \alpha - bn \alpha) \#* p$
and $supp\text{-}p: supp p \subseteq bn \alpha \cup p \cdot bn \alpha$
by (*metis Act-eq-iff-perm-renaming*)

from $valid$ **and** $p\text{-}x$ **have** $-p \cdot P' \models x$
by (*metis permute-minus-cancel(2) FL-valid-eqt*)

moreover from $fresh$ **and** $p\text{-}\alpha$ **have** $(p \cdot bn \alpha) \#* (F, f)$
by (*simp add: bn-eqt fresh-star-Pair*)
with $\langle bn \alpha \#* (F, f) \rangle$ **and** $supp\text{-}p$ **have** $supp (F, f) \#* p$
by (*meson UnE fresh-def fresh-star-def subsetCE*)
then have $supp F \#* p$ **and** $supp f \#* p$
by (*simp add: fresh-star-Un supp-Pair*)+

with $bisim'$ **and** $p\text{-}\alpha$ **have** $(-p \cdot P') \sim.[L(\alpha, F, f)] (-p \cdot Q')$
by (*metis FL-bisimilar-eqt L-eqt' permute-minus-cancel(2) supp-perm-eq*)

ultimately have $-p \cdot Q' \models x$
using $\langle \bigwedge P Q. P \sim.[L(\alpha, F, f)] Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **by** *metis*

with $p\text{-}x$ **have** $Q' \models x'$
by (*metis permute-minus-cancel(1) FL-valid-eqt*)

with eq **and** $trans'$ **show** $Q \models Act f \alpha x$
unfolding *FL-valid-Act* **by** *metis*
qed

theorem *FL-bisimilarity-implies-equivalence*: **assumes** $P \sim.[F] Q$ **shows** *FL-logically-equivalent*
 $F P Q$
unfolding *FL-logically-equivalent-def* **proof**
fix $x :: ('idx, 'pred, 'act, 'effect) formula$

```

show  $x \in \mathcal{A}[F] \longrightarrow P \models x \longleftrightarrow Q \models x$ 
proof
  assume  $x \in \mathcal{A}[F]$  then show  $P \models x \longleftrightarrow Q \models x$ 
  using assms proof (induction x arbitrary: P Q)
    case Conj then show ?case
      by simp
    next
      case Not then show ?case
        by simp
    next
      case Pred then show ?case
        by (metis FL-bisimilar-is-L-bisimulation is-L-bisimulation-def symp-def
FL-valid-Pred)
    next
      case Act then show ?case
        by (metis FL-bisimilar-symp FL-bisimilarity-implies-equivalence-Act sympE)
    qed
  qed
qed

end

end
theory FL-Equivalence-Implies-Bisimilarity
imports
  FL-Logical-Equivalence
begin

```

16 Logical Equivalence Implies F/L -Bisimilarity

```

context indexed-effect-nominal-ts
begin

```

```

  definition distinguishing-formula :: ('idx, 'pred, 'act, 'effect) formula  $\Rightarrow$  'state
 $\Rightarrow$  'state  $\Rightarrow$  bool

```

```

  (- distinguishes - from - [100,100,100] 100)

```

```

where

```

```

   $x$  distinguishes  $P$  from  $Q \equiv P \models x \wedge \neg Q \models x$ 

```

```

lemma distinguishing-formula-eqt :

```

```

  assumes  $x$  distinguishes  $P$  from  $Q$  shows  $(p \cdot x)$  distinguishes  $(p \cdot P)$  from
 $(p \cdot Q)$ 

```

```

using assms unfolding distinguishing-formula-def

```

```

by (metis permute-minus-cancel(2) FL-valid-eqt)

```

```

lemma FL-equivalent-iff-not-distinguished:

```

```

  FL-logically-equivalent  $F P Q \longleftrightarrow \neg(\exists x. x \in \mathcal{A}[F] \wedge x$  distinguishes  $P$  from
 $Q)$ 

```

```

by (meson FL-logically-equivalent-def Not distinguishing-formula-def FL-valid-Not)

```

There exists a distinguishing formula for P and Q in $\mathcal{A}[F]$ whose support is contained in $\text{supp}(F, P)$.

lemma *FL-distinguished-bounded-support*:

assumes $x \in \mathcal{A}[F]$ **and** x distinguishes P from Q

obtains y **where** $y \in \mathcal{A}[F]$ **and** $\text{supp } y \subseteq \text{supp}(F, P)$ **and** y distinguishes P from Q

proof –

let $?B = \{p \cdot x \mid p. \text{supp}(F, P) \#* p\}$

have $\text{supp}(F, P)$ supports $?B$

unfolding *supports-def* **proof** (*clarify*)

fix $a \ b$

assume $a: a \notin \text{supp}(F, P)$ **and** $b: b \notin \text{supp}(F, P)$

have $(a \rightleftharpoons b) \cdot ?B \subseteq ?B$

proof

fix x'

assume $x' \in (a \rightleftharpoons b) \cdot ?B$

then obtain p **where** $1: x' = (a \rightleftharpoons b) \cdot p \cdot x$ **and** $2: \text{supp}(F, P) \#* p$

by (*auto simp add: permute-set-def*)

let $?q = (a \rightleftharpoons b) + p$

from 1 **have** $x' = ?q \cdot x$

by *simp*

moreover from a **and** b **and** 2 **have** $\text{supp}(F, P) \#* ?q$

by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3)*)

ultimately show $x' \in ?B$ **by** *blast*

qed

moreover have $?B \subseteq (a \rightleftharpoons b) \cdot ?B$

proof

fix x'

assume $x' \in ?B$

then obtain p **where** $1: x' = p \cdot x$ **and** $2: \text{supp}(F, P) \#* p$

by *auto*

let $?q = (a \rightleftharpoons b) + p$

from 1 **have** $x' = (a \rightleftharpoons b) \cdot ?q \cdot x$

by *simp*

moreover from a **and** b **and** 2 **have** $\text{supp}(F, P) \#* ?q$

by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3)*)

ultimately show $x' \in (a \rightleftharpoons b) \cdot ?B$

using *mem-permute-iff* **by** *blast*

qed

ultimately show $(a \rightleftharpoons b) \cdot ?B = ?B$..

qed

then have *supp-B-subset-supp-P*: $\text{supp } ?B \subseteq \text{supp}(F, P)$

by (*metis (erased, lifting) finite-supp supp-is-subset*)

then have *finite-supp-B*: $\text{finite}(\text{supp } ?B)$

using *finite-supp rev-finite-subset* **by** *blast*

have $?B \subseteq (\lambda p. p \cdot x) \text{ ' } UNIV$

by *auto*

then have $|?B| \leq_o |UNIV :: \text{perm set}|$

```

    by (rule surj-imp-ordLeq)
  also have |UNIV :: perm set| <o |UNIV :: 'idx set|
    by (metis card-idx-perm)
  also have |UNIV :: 'idx set| ≤o natLeq + c |UNIV :: 'idx set|
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally have card-B: |?B| <o natLeq + c |UNIV :: 'idx set| .

let ?y = Conj (Abs-bset ?B) :: ('idx, 'pred, 'act, 'effect) formula

from finite-supp-B and card-B and supp-B-subset-supp-P have supp ?y ⊆
supp (F,P)
  by simp
moreover have ?y ∈ A[F]
proof
  show finite (supp (Abs-bset ?B :: (-,-,-,-) formula set['idx]))
    using finite-supp-B card-B by simp
next
  fix x'
  assume x' ∈ set-bset (Abs-bset ?B :: (-,-,-,-) formula set['idx])
  then obtain p where p-x: x' = p · x and fresh-p: supp (F,P) ‡* p
    using card-B by auto
  from fresh-p have p · F = F
    using fresh-star-Pair fresh-star-supp-conv perm-supp-eq by blast
  with ⟨x ∈ A[F]⟩ show x' ∈ A[F]
    using p-x by (metis is-FL-formula-eqvt)
qed
moreover have ?y distinguishes P from Q
  unfolding distinguishing-formula-def proof
    from ⟨x distinguishes P from Q⟩ show P ⊨ ?y
      by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
fresh-star-Un supp-Pair supp-perm-eq FL-valid-eqvt)
    next
      from ⟨x distinguishes P from Q⟩ show ¬ Q ⊨ ?y
        by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
    qed
  ultimately show ?thesis
    using that by blast
qed

lemma FL-equivalence-is-L-bisimulation: is-L-bisimulation FL-logically-equivalent
proof -
  {
    fix F have symp (FL-logically-equivalent F)
      by (rule sympI) (metis FL-logically-equivalent-def)
  }
  moreover
  {
    fix F P Q f φ

```

assume *FL-logically-equivalent* $F P Q$ **and** $f \in_{fs} F$ **and** $\langle f \rangle P \vdash \varphi$
then have $\langle f \rangle Q \vdash \varphi$
by (*metis FL-logically-equivalent-def Pred FL-valid-Pred*)
}
moreover
{
fix $F P Q f \alpha P'$
assume *FL-logically-equivalent* $F P Q$ **and** $f \in_{fs} F$ **and** $bn \alpha \#* (\langle f \rangle Q, F,$
 $f)$ **and** $\langle f \rangle P \rightarrow \langle \alpha, P' \rangle$
then have $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge$ *FL-logically-equivalent* $(L(\alpha, F, f)) P' Q'$
proof –
{
let $?Q' = \{Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle\}$
assume $\forall Q' \in ?Q'. \neg$ *FL-logically-equivalent* $(L(\alpha, F, f)) P' Q'$
then have $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act, 'effect)$ *formula*. $x \in \mathcal{A}[L$
 $(\alpha, F, f)] \wedge x$ *distinguishes* P' *from* Q'
by (*metis FL-equivalent-iff-not-distinguished*)
then have $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act, 'effect)$ *formula*. $x \in \mathcal{A}[L$
 $(\alpha, F, f)] \wedge supp\ x \subseteq supp\ (L(\alpha, F, f), P') \wedge x$ *distinguishes* P' *from* Q'
by (*metis FL-distinguished-bounded-support*)
then obtain $g :: 'state \Rightarrow ('idx, 'pred, 'act, 'effect)$ *formula* **where**
 $*$: $\forall Q' \in ?Q'. g\ Q' \in \mathcal{A}[L(\alpha, F, f)] \wedge supp\ (g\ Q') \subseteq supp\ (L(\alpha, F, f),$
 $P') \wedge (g\ Q')$ *distinguishes* P' *from* Q'
by *metis*
have $supp\ (L(\alpha, F, f), P')$ *supports* $(g\ ' ?Q')$
unfolding *supports-def* **proof** (*clarify*)
fix $a\ b$
assume $a: a \notin supp\ (L(\alpha, F, f), P')$ **and** $b: b \notin supp\ (L(\alpha, F, f), P')$
have $(a \Rightarrow b) \cdot (g\ ' ?Q') \subseteq g\ ' ?Q'$
proof
fix x'
assume $x' \in (a \Rightarrow b) \cdot (g\ ' ?Q')$
then obtain Q' **where** $1: x' = (a \Rightarrow b) \cdot g\ Q'$ **and** $2: \langle f \rangle Q \rightarrow$
 $\langle \alpha, Q' \rangle$
by *auto* (*metis effect-apply-eqv' permute-swap-cancel transition-eqv'*)
with $*$ **and** a **and** b **have** $a \notin supp\ (g\ Q')$ **and** $b \notin supp\ (g\ Q')$
by *auto*
with 1 **have** $x' = g\ Q'$
by (*metis fresh-perm fresh-star-def supp-perm-eq swap-atom*)
with 2 **show** $x' \in g\ ' ?Q'$
by *simp*
qed
moreover have $g\ ' ?Q' \subseteq (a \Rightarrow b) \cdot (g\ ' ?Q')$
proof
fix x'
assume $x' \in g\ ' ?Q'$
then obtain Q' **where** $1: x' = g\ Q'$ **and** $2: \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle$
by *auto*
with $*$ **and** a **and** b **have** $a \notin supp\ (g\ Q')$ **and** $b \notin supp\ (g\ Q')$

```

    by auto
  with 1 have  $x' = (a \equiv b) \cdot g \ Q'$ 
    by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
  with 2 show  $x' \in (a \equiv b) \cdot (g \ ' ?Q')$ 
    using mem-permute-iff by blast
  qed
  ultimately show  $(a \equiv b) \cdot (g \ ' ?Q') = g \ ' ?Q' ..$ 
  qed
  then have supp-image-subset-supp-P':  $\text{supp } (g \ ' ?Q') \subseteq \text{supp } (L (\alpha, F, f))$ ,
P')
    by (metis (erased, lifting) finite-supp supp-is-subset)
  then have finite-supp-image: finite ( $\text{supp } (g \ ' ?Q')$ )
    using finite-supp rev-finite-subset by blast
  have  $|g \ ' ?Q'| \leq o \ |UNIV :: 'state \ set|$ 
    by (metis card-of-UNIV card-of-image ordLeq-transitive)
  also have  $|UNIV :: 'state \ set| < o \ |UNIV :: 'idx \ set|$ 
    by (metis card-idx-state)
  also have  $|UNIV :: 'idx \ set| \leq o \ \text{natLeq} \ +c \ |UNIV :: 'idx \ set|$ 
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally have card-image:  $|g \ ' ?Q'| < o \ \text{natLeq} \ +c \ |UNIV :: 'idx \ set|$  .
  let  $?y = \text{Conj } (\text{Abs-bset } (g \ ' ?Q')) :: ('idx, 'pred, 'act, 'effect) \ \text{formula}$ 
  have Act f  $\alpha$  ?y  $\in \mathcal{A}[F]$ 
  proof
    from  $\langle f \in_{fs} F \rangle$  show  $f \in_{fs} F$  .
  next
    from  $\langle bn \ \alpha \ \sharp^* \ (\langle f \rangle Q, F, f) \rangle$  show  $bn \ \alpha \ \sharp^* \ (F, f)$ 
      using fresh-star-Pair by blast
  next
    show Conj ( $\text{Abs-bset } (g \ ' ?Q') \in \mathcal{A}[L (\alpha, F, f)]$ )
    proof
      show finite ( $\text{supp } (\text{Abs-bset } (g \ ' ?Q') :: (-,-,-) \ \text{formula} \ \text{set}['idx])$ )
        using finite-supp-image card-image by simp
    next
      fix  $x'$ 
    assume  $x' \in \text{set-bset } (\text{Abs-bset } (g \ ' ?Q') :: (-,-,-) \ \text{formula} \ \text{set}['idx])$ 
      then obtain  $Q'$  where  $x' = g \ Q'$  and  $\langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle$ 
        using card-image by auto
      with * show  $x' \in \mathcal{A}[L (\alpha, F, f)]$ 
        using mem-Collect-eq by blast
    qed
  qed
  moreover have  $P \models \text{Act } f \ \alpha \ ?y$ 
  unfolding FL-valid-Act proof (standard+)
  show  $\langle f \rangle P \rightarrow \langle \alpha, P' \rangle$  by fact
  next
  {
    fix  $Q'$ 
    assume  $\langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle$ 
    with * have  $P' \models g \ Q'$ 
  }

```

```

    by (metis distinguishing-formula-def mem-Collect-eq)
  }
  then show  $P' \models ?y$ 
    by (simp add: finite-supp-image card-image)
  qed
  moreover have  $\neg Q \models \text{Act } f \ \alpha \ ?y$ 
  proof
    assume  $Q \models \text{Act } f \ \alpha \ ?y$ 
    then obtain  $Q'$  where 1:  $\langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle$  and 2:  $Q' \models ?y$ 
  using  $\langle \text{bn } \alpha \ \# * (\langle f \rangle Q, F, f) \rangle$  by (metis fresh-star-Pair FL-valid-Act-fresh)
  from 2 have  $\bigwedge Q''. \langle f \rangle Q \rightarrow \langle \alpha, Q'' \rangle \longrightarrow Q' \models g \ Q''$ 
    by (simp add: finite-supp-image card-image)
  with 1 and * show False
    using distinguishing-formula-def by blast
  qed
  ultimately have False
    by (metis  $\langle \text{FL-logically-equivalent } F \ P \ Q \rangle$  FL-logically-equivalent-def)
  }
  then show ?thesis by auto
  qed
}
ultimately show ?thesis
  unfolding is-L-bisimulation-def by metis
qed

```

theorem *FL-equivalence-implies-bisimilarity*: **assumes** *FL-logically-equivalent* $F \ P \ Q$ **shows** $P \sim_{\cdot[F]} Q$
using *assms* **by** (metis *FL-bisimilar-def FL-equivalence-is-L-bisimulation*)

end

end

theory *L-Transform*

imports

Validity

Bisimilarity-Implies-Equivalence

FL-Equivalence-Implies-Bisimilarity

begin

17 *L-Transform*

17.1 States

The intuition is that states of kind *AC* can perform ordinary actions, and states of kind *EF* can commit effects.

```

datatype ('state, 'effect) L-state =
  AC 'effect  $\times$  'effect fs-set  $\times$  'state
  | EF 'effect fs-set  $\times$  'state

```



```

instantiation L-state :: (pt,pt) pt
begin

  fun permute-L-state :: perm ⇒ ('a,'b) L-state ⇒ ('a,'b) L-state where
    p · (AC x) = AC (p · x)
  | p · (EF x) = EF (p · x)

  instance
  proof
    fix x :: ('a,'b) L-state
    show 0 · x = x by (cases x, simp-all)
  next
    fix p q and x :: ('a,'b) L-state
    show (p + q) · x = p · q · x by (cases x, simp-all)
  qed

end

declare permute-L-state.simps [eqvt]

lemma supp-AC [simp]: supp (AC x) = supp x
unfolding supp-def by simp

lemma supp-EF [simp]: supp (EF x) = supp x
unfolding supp-def by simp

instantiation L-state :: (fs,fs) fs
begin

  instance
  proof
    fix x :: ('a,'b) L-state
    show finite (supp x)
    by (cases x) (simp add: finite-supp)+
  qed

end

17.2 Actions and binding names

datatype ('act,'effect) L-action =
  Act 'act
  | Eff 'effect

instantiation L-action :: (pt,pt) pt
begin

  fun permute-L-action :: perm ⇒ ('a,'b) L-action ⇒ ('a,'b) L-action where

```

```

  p · (Act α) = Act (p · α)
| p · (Eff f) = Eff (p · f)

instance
proof
  fix x :: ('a,'b) L-action
  show 0 · x = x by (cases x, simp-all)
next
  fix p q and x :: ('a,'b) L-action
  show (p + q) · x = p · q · x by (cases x, simp-all)
qed

end

declare permute-L-action.simps [eqvt]

lemma supp-Act [simp]: supp (Act α) = supp α
unfolding supp-def by simp

lemma supp-Eff [simp]: supp (Eff f) = supp f
unfolding supp-def by simp

instantiation L-action :: (fs,fs) fs
begin

  instance
  proof
    fix x :: ('a,'b) L-action
    show finite (supp x)
      by (cases x) (simp add: finite-supp)+
  qed

end

instantiation L-action :: (bn,fs) bn
begin

  fun bn-L-action :: ('a,'b) L-action ⇒ atom set where
    bn-L-action (Act α) = bn α
  | bn-L-action (Eff -) = {}

  instance
  proof
    fix p and α :: ('a,'b) L-action
    show p · bn α = bn (p · α)
      by (cases α) (simp add: bn-eqvt, simp)
  next
    fix α :: ('a,'b) L-action
    show finite (bn α)

```

by (cases α) (simp add: bn-finite, simp)
qed

end

17.3 Satisfaction

context *effect-nominal-ts*
begin

fun *L-satisfies* :: ('state,'effect) *L-state* \Rightarrow 'pred \Rightarrow bool (infix \vdash_L 70) where
 $AC (-, -, P) \vdash_L \varphi \longleftrightarrow P \vdash \varphi$
 $| EF - \vdash_L \varphi \longleftrightarrow False$

lemma *L-satisfies-eqvt*: assumes $P_L \vdash_L \varphi$ shows $(p \cdot P_L) \vdash_L (p \cdot \varphi)$
proof (cases P_L)
 case (*AC fFP*)
 with *assms* **have** $snd (snd fFP) \vdash \varphi$
 by (metis *L-satisfies.simps(1) prod.collapse*)
 then have $snd (snd (p \cdot fFP)) \vdash p \cdot \varphi$
 by (metis *satisfies-eqvt snd-eqvt*)
 then show ?thesis
 using *AC* **by** (metis *L-satisfies.simps(1) permute-L-state.simps(1) prod.collapse*)
next
 case *EF*
 with *assms* **have** *False*
 by *simp*
 then show ?thesis ..
qed

end

17.4 Transitions

context *effect-nominal-ts*
begin

fun *L-transition* :: ('state,'effect) *L-state* \Rightarrow (('act,'effect) *L-action*, ('state,'effect) *L-state*) *residual* \Rightarrow bool (infix \rightarrow_L 70) where
 $AC (f, F, P) \rightarrow_L \alpha P' \longleftrightarrow (\exists \alpha P'. P \rightarrow \langle \alpha, P \rangle \wedge \alpha P' = \langle Act \alpha, EF (L (\alpha, F, f), P) \rangle \wedge bn \alpha \#* (F, f))$ — note the freshness condition
 $| EF (F, P) \rightarrow_L \alpha P' \longleftrightarrow (\exists f. f \in_{fs} F \wedge \alpha P' = \langle Eff f, AC (f, F, \langle f \rangle P) \rangle)$

lemma *L-transition-eqvt*: assumes $P_L \rightarrow_L \alpha_L P_L'$ shows $(p \cdot P_L) \rightarrow_L (p \cdot \alpha_L P_L')$
proof (cases P_L)
 case *AC*
 {
 fix $f F P$
 assume *: $P_L = AC (f, F, P)$

```

    with assms obtain  $\alpha P'$  where trans:  $P \rightarrow \langle \alpha, P' \rangle$  and  $\alpha P'$ :  $\alpha_L P_L' = \langle Act \alpha, EF (L (\alpha, F, f), P') \rangle$  and fresh:  $bn \alpha \#* (F, f)$ 
      by auto
    from trans have  $p \cdot P \rightarrow \langle p \cdot \alpha, p \cdot P' \rangle$ 
      by (simp add: transition-eqvt)
    moreover from  $\alpha P'$  have  $p \cdot \alpha_L P_L' = \langle Act (p \cdot \alpha), EF (L (p \cdot \alpha, p \cdot F, p \cdot f), p \cdot P') \rangle$ 
      by (simp add: L-eqvt)
    moreover from fresh have  $bn (p \cdot \alpha) \#* (p \cdot F, p \cdot f)$ 
      by (metis bn-eqvt fresh-star-Pair fresh-star-permute-iff)
    ultimately have  $p \cdot P_L \rightarrow_L p \cdot \alpha_L P_L'$ 
      using * by auto
  }
  with AC show ?thesis
    by (metis prod.collapse)
next
case EF
{
  fix  $F P$ 
  assume *:  $P_L = EF (F, P)$ 
  with assms obtain  $f$  where  $f \in_{fs} F$  and  $\alpha_L P_L' = \langle Eff f, AC (f, F, \langle f \rangle P) \rangle$ 
    by auto
  then have  $(p \cdot f) \in_{fs} (p \cdot F)$  and  $p \cdot \alpha_L P_L' = \langle Eff (p \cdot f), AC (p \cdot f, p \cdot F, \langle p \cdot f \rangle (p \cdot P)) \rangle$ 
    by simp+
  then have  $p \cdot P_L \rightarrow_L p \cdot \alpha_L P_L'$ 
    using * L-transition.simps(2) Pair-eqvt permute-L-state.simps(2) by force
}
with EF show ?thesis
  by (metis prod.collapse)
qed

```

The binding names in the alpha-variant that witnesses the L -transition may be chosen fresh for any finitely supported context.

```

lemma L-transition-AC-strong:
  assumes finite (supp  $X$ ) and  $AC (f, F, P) \rightarrow_L \langle \alpha_L, P_L' \rangle$ 
  shows  $\exists \alpha P'. P \rightarrow \langle \alpha, P' \rangle \wedge \langle \alpha_L, P_L' \rangle = \langle Act \alpha, EF (L (\alpha, F, f), P') \rangle \wedge bn \alpha \#* X$ 
  using assms proof –
    from  $\langle AC (f, F, P) \rightarrow_L \langle \alpha_L, P_L' \rangle \rangle$  obtain  $\alpha P'$  where transition:  $P \rightarrow \langle \alpha, P' \rangle$ 
  and alpha:  $\langle \alpha_L, P_L' \rangle = \langle Act \alpha, EF (L (\alpha, F, f), P') \rangle$  and fresh:  $bn \alpha \#* (F, f)$ 
    by (metis L-transition.simps(1))
    let  $?Act = Act \alpha :: ('act, 'effect) L\text{-action}$  — the type annotation prevents a type that is too polymorphic and doesn't fix 'effect
    have finite ( $bn \alpha$ )
      by (fact bn-finite)
    moreover note (finite (supp  $X$ ))
    moreover have finite (supp ( $\langle ?Act, EF (L (\alpha, F, f), P') \rangle, \langle \alpha, P' \rangle, F, f$ ))
      by (metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair)

```

moreover from fresh have $bn \alpha \#* (\langle ?Act, EF (L (\alpha, F, f), P') \rangle, \langle \alpha, P' \rangle, F, f)$
by (*auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
ultimately obtain p **where** *fresh-X*: $(p \cdot bn \alpha) \#* X$ **and** $supp (\langle ?Act, EF (L (\alpha, F, f), P') \rangle, \langle \alpha, P' \rangle, F, f) \#* p$
by (*metis at-set-avoiding2*)
then have $supp \langle ?Act, EF (L (\alpha, F, f), P') \rangle \#* p$ **and** $supp \langle \alpha, P' \rangle \#* p$ **and** $supp (F, f) \#* p$
by (*metis fresh-star-Un supp-Pair*)
then have $p \cdot \langle ?Act, EF (L (\alpha, F, f), P') \rangle = \langle ?Act, EF (L (\alpha, F, f), P') \rangle$ **and** $p \cdot \langle \alpha, P' \rangle = \langle \alpha, P' \rangle$ **and** $p \cdot (F, f) = (F, f)$
by (*metis supp-perm-eq*)
then have $\langle Act (p \cdot \alpha), EF (L (p \cdot \alpha, F, f), p \cdot P') \rangle = \langle ?Act, EF (L (\alpha, F, f), P') \rangle$ **and** $\langle p \cdot \alpha, p \cdot P' \rangle = \langle \alpha, P' \rangle$
using *permute-L-action.simps(1) permute-L-state.simps(2) abs-residual-pair-eqvt L-eqvt' Pair-eqvt* **by** *auto*
then show $\exists \alpha P'. P \rightarrow \langle \alpha, P' \rangle \wedge \langle \alpha_L, P_L' \rangle = \langle Act \alpha, EF (L (\alpha, F, f), P') \rangle \wedge bn \alpha \#* X$
using *transition and alpha and fresh-X* **by** (*metis bn-eqvt*)
qed

lemma *L-transition-AC-fresh*:

assumes $bn \alpha \#* (F, f, P)$

shows $AC (f, F, P) \rightarrow_L \langle Act \alpha, P_L' \rangle \longleftrightarrow (\exists P'. P_L' = EF (L (\alpha, F, f), P') \wedge P \rightarrow \langle \alpha, P' \rangle)$

proof

assume $AC (f, F, P) \rightarrow_L \langle Act \alpha, P_L' \rangle$

moreover have *finite* ($supp (F, f, P)$)

by (*fact finite-supp*)

ultimately obtain $\alpha' P'$ **where** *trans*: $P \rightarrow \langle \alpha', P' \rangle$ **and** *eq*: $\langle Act \alpha :: ('act, 'effect) L-action, P_L' \rangle = \langle Act \alpha', EF (L (\alpha', F, f), P') \rangle$ **and** *fresh*: $bn \alpha' \#* (F, f, P)$

using *L-transition-AC-strong* **by** *blast*

from eq obtain p **where** $p \cdot (Act \alpha :: ('act, 'effect) L-action, P_L') = (Act \alpha', EF (L (\alpha', F, f), P'))$ **and** *supp-p*: $supp p \subseteq bn (Act \alpha :: ('act, 'effect) L-action) \cup p \cdot bn (Act \alpha :: ('act, 'effect) L-action)$

using *residual-eq-iff-perm-renaming* **by** *metis*

from p **have** $p \cdot \alpha = \alpha'$ **and** $p \cdot P_L' = P_L' = EF (L (\alpha', F, f), P')$

by *simp-all*

from *supp-p* **and** $p \cdot \alpha$ **and** *assms* **and** *fresh* **have** $supp p \#* (F, f, P)$

by (*simp add: bn-eqvt fresh-star-def*) *blast*

then have $p \cdot F = F$ **and** $p \cdot f = f$ **and** $p \cdot P = P$

by (*simp-all add: fresh-star-Pair perm-supp-eq*)

from $p \cdot P_L'$ **have** $P_L' = -p \cdot EF (L (\alpha', F, f), P')$

```

    by (metis permute-minus-cancel(2))
  then have  $P_L' = EF (L (\alpha, F, f), -p \cdot P')$ 
    using  $p\text{-}\alpha$   $p\text{-}F$   $p\text{-}f$  by simp (metis (full-types) permute-minus-cancel(2))

  moreover from trans have  $P \rightarrow \langle \alpha, -p \cdot P \rangle$ 
    using  $p\text{-}P$  and  $p\text{-}\alpha$  by (metis permute-minus-cancel(2) transition-eqt')

  ultimately show  $\exists P'. P_L' = EF (L (\alpha, F, f), P') \wedge P \rightarrow \langle \alpha, P \rangle$ 
    by blast
  next
  assume  $\exists P'. P_L' = EF (L (\alpha, F, f), P') \wedge P \rightarrow \langle \alpha, P \rangle$ 
  moreover from assms have  $bn \alpha \#* (F, f)$ 
    by (simp add: fresh-star-Pair)
  ultimately show  $AC (f, F, P) \rightarrow_L \langle Act \alpha, P_L \rangle$ 
    using  $L\text{-transition.simps}(1)$  by blast
qed
end

```

17.5 Translation of F/L -formulas into formulas without effects

Since we defined formulas via a manual quotient construction, we also need to define the L -transform via lifting from the underlying type of infinitely branching trees. As before, we cannot use **nominal_function** because that generates proof obligations where, for formulas of the form $FL\text{-Formula.Conj } xset$, the assumption that $xset$ has finite support is missing.

The following auxiliary function returns trees (modulo α -equivalence) rather than formulas. This allows us to prove equivariance for *all* argument trees, without an assumption that they are (hereditarily) finitely supported. Further below—after this auxiliary function has been lifted to F/L -formulas as arguments—we derive a version that returns formulas.

```

primrec  $L\text{-transform-Tree} :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) Tree \Rightarrow ('idx, 'pred, 'act, 'eff) L\text{-action} Formula.Tree_\alpha$  where
   $L\text{-transform-Tree} (tConj tset) = Formula.Conj_\alpha (map\text{-bset } L\text{-transform-Tree } tset)$ 
|  $L\text{-transform-Tree} (tNot t) = Formula.Not_\alpha (L\text{-transform-Tree } t)$ 
|  $L\text{-transform-Tree} (tPred f \varphi) = Formula.Act_\alpha (Eff f) (Formula.Pred_\alpha \varphi)$ 
|  $L\text{-transform-Tree} (tAct f \alpha t) = Formula.Act_\alpha (Eff f) (Formula.Act_\alpha (Act \alpha) (L\text{-transform-Tree } t))$ 

```

lemma $L\text{-transform-Tree-eqt}$ [eqvt]: $p \cdot L\text{-transform-Tree } t = L\text{-transform-Tree} (p \cdot t)$

```

proof (induct t)
  case (tConj tset)
  then show ?case
    by simp (metis (no-types, hide-lams) bset.map-cong0 map-bset-eqt permute-fun-def permute-minus-cancel(1))

```

qed *simp-all*

L-transform-Tree respects α -equivalence.

lemma *alpha-Tree-L-transform-Tree*:

assumes *alpha-Tree* *t1 t2*

shows *L-transform-Tree* *t1* = *L-transform-Tree* *t2*

using *assms* **proof** (*induction* *t1 t2* *rule: alpha-Tree-induct'*)

case (*alpha-tConj* *tset1 tset2*)

then have *rel-bset* (=) (*map-bset* *L-transform-Tree* *tset1*) (*map-bset* *L-transform-Tree* *tset2*)

by (*simp* *add: bset.rel-map(1) bset.rel-map(2) bset.rel-mono-strong*)

then show *?case*

by (*simp* *add: bset.rel-eq*)

next

case (*alpha-tAct* *f1* $\alpha 1$ *t1* *f2* $\alpha 2$ *t2*)

from \langle *alpha-Tree* (*FL-Formula.Tree.tAct* *f1* $\alpha 1$ *t1*) (*FL-Formula.Tree.tAct* *f2* $\alpha 2$ *t2*) \rangle

obtain *p* **where** $*$: (*bn* $\alpha 1$, *t1*) \approx_{set} *alpha-Tree* (*supp-rel* *alpha-Tree*) *p* (*bn* $\alpha 2$, *t2*)

and $**$: (*bn* $\alpha 1$, $\alpha 1$) \approx_{set} (=) *supp* *p* (*bn* $\alpha 2$, $\alpha 2$) **and** *f1* = *f2*

by *auto*

from $*$ **have** *fresh*: (*supp-rel* *alpha-Tree* *t1* - *bn* $\alpha 1$) $\#*$ *p* **and** *alpha*: *alpha-Tree* (*p* \cdot *t1*) *t2* **and** *eq*: *p* \cdot *bn* $\alpha 1$ = *bn* $\alpha 2$

by (*auto* *simp* *add: alpha-set*)

from *alpha-tAct.IH(2)* **have** *supp-rel* *Formula.alpha-Tree* (*Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t1*)) \subseteq *supp-rel* *alpha-Tree* *t1*

by (*metis* (*no-types, lifting*) *infinite-mono* *Formula.alpha-Tree-permute-rep-commute* *L-transform-Tree-eqt* *mem-Collect-eq* *subsetI* *supp-rel-def*)

with *fresh* **have** *fresh'*: (*supp-rel* *Formula.alpha-Tree* (*Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t1*)) - *bn* $\alpha 1$) $\#*$ *p*

by (*meson* *DiffD1* *DiffD2* *DiffI* *fresh-star-def* *subsetCE*)

moreover from *alpha* **have** *alpha'*: *Formula.alpha-Tree* (*p* \cdot *Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t1*)) (*Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t2*))

using *alpha-tAct.IH(1)* **by** (*metis* *Formula.alpha-Tree-permute-rep-commute* *L-transform-Tree-eqt*)

moreover from *fresh'* *alpha'* *eq* **have** *supp-rel* *Formula.alpha-Tree* (*Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t1*)) - *bn* $\alpha 1$ = *supp-rel* *Formula.alpha-Tree* (*Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t2*)) - *bn* $\alpha 2$

by (*metis* (*mono-tags*) *Diff-eqt* *Formula.alpha-Tree-eqt'* *Formula.alpha-Tree-eqt-aux* *Formula.alpha-Tree-supp-rel* *atom-set-perm-eq*)

ultimately have (*bn* $\alpha 1$, *Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t1*)) \approx_{set} *Formula.alpha-Tree* (*supp-rel* *Formula.alpha-Tree*) *p* (*bn* $\alpha 2$, *Formula.rep-Tree* $_{\alpha}$ (*L-transform-Tree* *t2*))

using *eq* **by** (*simp* *add: alpha-set*)

moreover from $**$ **have** (*bn* $\alpha 1$, *Act* $\alpha 1$) \approx_{set} (=) *supp* *p* (*bn* $\alpha 2$, *Act* $\alpha 2$)

by (*metis* (*mono-tags, lifting*) *L-Transform.supp-Act* *alpha-set* *permute-L-action.simps(1)*)

ultimately have *Formula.Act* $_{\alpha}$ (*Act* $\alpha 1$) (*L-transform-Tree* *t1*) = *Formula.Act* $_{\alpha}$ (*Act* $\alpha 2$) (*L-transform-Tree* *t2*)

by (*auto* *simp* *add: Formula.Act $_{\alpha}$ -eq-iff*)

with $\langle f1 = f2 \rangle$ **show** $?case$
by *simp*
qed *simp-all*

L-transform for trees modulo α -equivalence.

lift-definition *L-transform-Tree* $_{\alpha} :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) Tree_{\alpha} \Rightarrow ('idx, 'pred, ('act, 'eff) L-action) Formula.Tree_{\alpha}$ **is**
L-transform-Tree
by (*fact alpha-Tree-L-transform-Tree*)

lemma *L-transform-Tree* $_{\alpha}$ -*eqvt* [*eqvt*]: $p \cdot L\text{-transform-Tree}_{\alpha} t_{\alpha} = L\text{-transform-Tree}_{\alpha} (p \cdot t_{\alpha})$
by *transfer (simp)*

lemma *L-transform-Tree* $_{\alpha}$ -*Conj* $_{\alpha}$ [*simp*]: $L\text{-transform-Tree}_{\alpha} (Conj_{\alpha} tset_{\alpha}) = Formula.Conj_{\alpha} (map\text{-bset } L\text{-transform-Tree}_{\alpha} tset_{\alpha})$
by (*simp add: Conj_{\alpha}-def' L-transform-Tree_{\alpha}.abs-eq*) (*metis (no-types, lifting) L-transform-Tree_{\alpha}.rep-eq bset.map-comp bset.map-cong0 comp-apply*)

lemma *L-transform-Tree* $_{\alpha}$ -*Not* $_{\alpha}$ [*simp*]: $L\text{-transform-Tree}_{\alpha} (Not_{\alpha} t_{\alpha}) = Formula.Not_{\alpha} (L\text{-transform-Tree}_{\alpha} t_{\alpha})$
by *transfer simp*

lemma *L-transform-Tree* $_{\alpha}$ -*Pred* $_{\alpha}$ [*simp*]: $L\text{-transform-Tree}_{\alpha} (Pred_{\alpha} f \varphi) = Formula.Act_{\alpha} (Eff f) (Formula.Pred_{\alpha} \varphi)$
by *transfer simp*

lemma *L-transform-Tree* $_{\alpha}$ -*Act* $_{\alpha}$ [*simp*]: $L\text{-transform-Tree}_{\alpha} (Act_{\alpha} f \alpha t_{\alpha}) = Formula.Act_{\alpha} (Eff f) (Formula.Act_{\alpha} (Act \alpha) (L\text{-transform-Tree}_{\alpha} t_{\alpha}))$
by *transfer simp*

lemma *finite-supp-map-bset-L-transform-Tree* $_{\alpha}$ [*simp*]:
assumes *finite* (*supp tset* $_{\alpha}$)
shows *finite* (*supp* (*map-bset L-transform-Tree* $_{\alpha}$ *tset* $_{\alpha}$))
proof –

have *eqvt map-bset and eqvt L-transform-Tree* $_{\alpha}$
by (*simp add: eqvtI*)
then have *supp* (*map-bset L-transform-Tree* $_{\alpha}$) = {}
using *supp-fun-eqvt supp-fun-app-eqvt* **by** *blast*
then have *supp* (*map-bset L-transform-Tree* $_{\alpha}$ *tset* $_{\alpha}$) \subseteq *supp tset* $_{\alpha}$
using *supp-fun-app* **by** *blast*
with *assms* **show** *finite* (*supp* (*map-bset L-transform-Tree* $_{\alpha}$ *tset* $_{\alpha}$))
by (*metis finite-subset*)
qed

lemma *L-transform-Tree* $_{\alpha}$ -*preserves-hereditarily-fs*:
assumes *hereditarily-fs* *t* $_{\alpha}$
shows *Formula.hereditarily-fs* (*L-transform-Tree* $_{\alpha}$ *t* $_{\alpha}$)
using *assms* **proof** (*induct rule: hereditarily-fs.induct*)


```

  case (Conjα tsetα)
  then show ?case
    by (auto intro!: Formula.hereditarily-fs.Conjα) (metis imageE map-bset.rep-eq)
next
  case (Notα tα)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Notα)
next
  case (Predα f φ)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Actα Formula.hereditarily-fs.Predα)
next
  case (Actα tα f α)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Actα)
qed

```

L-transform for *F*/*L*-formulas.

lift-definition *L*-transform-formula :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) formula ⇒ ('idx, 'pred, ('act, 'eff) *L*-action) Formula.Tree_α is
L-transform-Tree_α
 .

lemma *L*-transform-formula-eqvt [eqvt]: *p* · *L*-transform-formula *x* = *L*-transform-formula (*p* · *x*)
 by transfer (simp)

lemma *L*-transform-formula-Conj [simp]:
 assumes finite (supp *xset*)
 shows *L*-transform-formula (Conj *xset*) = Formula.Conj_α (map-bset *L*-transform-formula *xset*)
 using assms by (simp add: Conj-def *L*-transform-formula-def bset.map-comp map-fun-def)

lemma *L*-transform-formula-Not [simp]: *L*-transform-formula (Not *x*) = Formula.Not_α (*L*-transform-formula *x*)
 by transfer simp

lemma *L*-transform-formula-Pred [simp]: *L*-transform-formula (Pred *f* φ) = Formula.Act_α (Eff *f*) (Formula.Pred_α φ)
 by transfer simp

lemma *L*-transform-formula-Act [simp]: *L*-transform-formula (FL-Formula.Act *f* α *x*) = Formula.Act_α (Eff *f*) (Formula.Act_α (Act α) (*L*-transform-formula *x*))
 by transfer simp

lemma *L*-transform-formula-hereditarily-fs [simp]: Formula.hereditarily-fs (*L*-transform-formula *x*)
 by transfer (fact *L*-transform-Tree_α-preserves-hereditarily-fs)

Finally, we define the proper L -transform, which returns formulas instead of trees.

definition L -transform $:: ('idx, 'pred::fs, 'act::bn, 'eff::fs) formula \Rightarrow ('idx, 'pred, ('act, 'eff) L$ -action) $Formula$.formula **where**
 L -transform $x = Formula.Abs$ -formula (L -transform-formula x)

lemma L -transform-eqvt [eqvt]: $p \cdot L$ -transform $x = L$ -transform ($p \cdot x$)
unfolding L -transform-def **by** *simp*

lemma *finite-supp-map-bset-L-transform* [simp]:
assumes *finite* (*supp* *xset*)
shows *finite* (*supp* (*map-bset* L -transform *xset*))

proof –

have *eqvt map-bset and eqvt L-transform*
by (*simp add: eqvtI*)
then have *supp* (*map-bset* L -transform) = {}
using *supp-fun-eqvt supp-fun-app-eqvt* **by** *blast*
then have *supp* (*map-bset* L -transform *xset*) \subseteq *supp* *xset*
using *supp-fun-app* **by** *blast*
with *assms* **show** *finite* (*supp* (*map-bset* L -transform *xset*))
by (*metis finite-subset*)

qed

lemma L -transform-Conj [simp]:
assumes *finite* (*supp* *xset*)
shows L -transform (*Conj* *xset*) = $Formula$.Conj (*map-bset* L -transform *xset*)
using *assms* **unfolding** L -transform-def **by** (*simp add: Formula*.Conj-def *bset.map-comp o-def*)

lemma L -transform-Not [simp]: L -transform (*Not* x) = $Formula$.Not (L -transform x)
unfolding L -transform-def **by** (*simp add: Formula*.Not-def)

lemma L -transform-Pred [simp]: L -transform (*Pred* f φ) = $Formula$.Act (*Eff* f) ($Formula$.Pred φ)
unfolding L -transform-def **by** (*simp add: Formula*.Act-def $Formula$.Pred-def $Formula$.hereditarily-fs.Pred $_{\alpha}$)

lemma L -transform-Act [simp]: L -transform (FL - $Formula$.Act f α x) = $Formula$.Act (*Eff* f) ($Formula$.Act (*Act* α) (L -transform x))
unfolding L -transform-def **by** (*simp add: Formula*.Act-def $Formula$.hereditarily-fs.Act $_{\alpha}$)

context *effect-nominal-ts*
begin

interpretation L -transform: *nominal-ts* (\vdash_L) (\rightarrow_L)
by *unfold-locales* (*fact L-satisfies-eqvt*, *fact L-transition-eqvt*)

The L -transform preserves satisfaction of formulas in the following sense:

```

theorem FL-valid-iff-valid-L-transform:
  assumes (x::('idx,'pred,'act,'effect) formula) ∈  $\mathcal{A}[F]$ 
  shows  $FL\text{-valid } P \ x \longleftrightarrow L\text{-transform.valid } (EF \ (F, P)) \ (L\text{-transform } x)$ 
using assms proof (induct x arbitrary: P)
  case (Conj xset F)
  then show ?case
    by auto (metis imageE map-bset.rep-eq, simp add: map-bset.rep-eq)
next
  case (Not F x)
  then show ?case by simp
next
  case (Pred f F  $\varphi$ )
let ? $\varphi$  = Formula.Pred  $\varphi$  :: ('idx, 'pred, ('act,'effect) L-action) Formula.formula
show ?case
proof
  assume  $FL\text{-valid } P \ (Pred \ f \ \varphi)$ 
  then have  $L\text{-transform.valid } (AC \ (f, F, \langle f \rangle P)) \ ?\varphi$ 
    by (simp add: L-transform.valid-Act)
  moreover from  $\langle f \in_{fs} F \rangle$  have  $EF \ (F, P) \rightarrow_L \langle Eff \ f, AC \ (f, F, \langle f \rangle P) \rangle$ 
    by (metis L-transition.simps(2))
  ultimately show  $L\text{-transform.valid } (EF \ (F, P)) \ (L\text{-transform } (Pred \ f \ \varphi))$ 
    using L-transform.valid-Act by fastforce
next
  assume  $L\text{-transform.valid } (EF \ (F, P)) \ (L\text{-transform } (Pred \ f \ \varphi))$ 
  then obtain  $P'$  where  $trans: EF \ (F, P) \rightarrow_L \langle Eff \ f, P' \rangle$  and valid:
 $L\text{-transform.valid } P' \ ?\varphi$ 
    by simp (metis bn-L-action.simps(2) empty-iff fresh-star-def L-transform.valid-Act-fresh
 $L\text{-transform.valid-Pred } L\text{-transition.simps}(2))$ 
  from trans have  $P' = AC \ (f, F, \langle f \rangle P)$ 
    by (simp add: residual-empty-bn-eq-iff)
  with valid show  $FL\text{-valid } P \ (Pred \ f \ \varphi)$ 
    by simp
qed
next
  case (Act f F  $\alpha \ x$ )
  show ?case
proof
  assume  $FL\text{-valid } P \ (FL\text{-Formula.Act } f \ \alpha \ x)$ 
  then obtain  $\alpha' \ x' \ P'$  where eq:  $FL\text{-Formula.Act } f \ \alpha \ x = FL\text{-Formula.Act } f \ \alpha' \ x'$ 
and trans:  $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$  and valid:  $FL\text{-valid } P' \ x'$  and fresh:  $bn \ \alpha' \ \#\ast (F, f)$ 
    by (metis FL-valid-Act-strong finite-supp)
  from eq obtain  $p$  where  $p \cdot x = x'$  and  $p \cdot \alpha = \alpha'$  and supp-p:
 $supp \ p \subseteq bn \ \alpha \cup bn \ \alpha'$ 
    by (metis bn-eqt FL-Formula.Act-eq-iff-perm-renaming)
  from  $\langle bn \ \alpha \ \#\ast (F, f) \rangle$  and fresh have  $supp \ (F, f) \ \#\ast \ p$ 
    using supp-p by (auto simp add: fresh-star-Pair fresh-star-def supp-Pair
fresh-def)
  then have  $p \cdot F = F$  and  $p \cdot f = f$ 

```

```

using supp-perm-eq by fastforce+

from valid have FL-valid  $(-p \cdot P')$  x
  using p-x by (metis FL-valid-eqvt permute-minus-cancel(2))
then have L-transform.valid  $(EF (L (\alpha, F, f), -p \cdot P'))$  (L-transform x)
  using Act.hyps(4) by metis
then have L-transform.valid  $(p \cdot EF (L (\alpha, F, f), -p \cdot P'))$  (p · L-transform
x)
  by (fact L-transform.valid-eqvt)
then have L-transform.valid  $(EF (L (\alpha', F, f), P'))$  (L-transform x')
  using p-x and p-α and  $\langle p \cdot F = F \rangle$  and  $\langle p \cdot f = f \rangle$  by simp

  then have L-transform.valid  $(AC (f, F, \langle f \rangle P))$  (Formula.Act (Act α')
(L-transform x'))
  using trans fresh L-transform.valid-Act by fastforce
  with  $\langle f \in_{f_s} F \rangle$  and eq show L-transform.valid  $(EF (F, P))$  (L-transform
(FL-Formula.Act f α x))
  using L-transform.valid-Act by fastforce
next
  assume *: L-transform.valid  $(EF (F, P))$  (L-transform (FL-Formula.Act f
α x))

  — rename bn α to avoid  $(F, f, P)$ , without touching F or FL-Formula.Act f
α x
  obtain p where 1:  $(p \cdot bn \alpha) \#* (F, f, P)$  and 2: supp  $(F, FL-Formula.Act$ 
f α x)  $\#* p$ 
  proof (rule at-set-avoiding2[of bn α (F, f, P) (F, FL-Formula.Act f α x),
THEN exE])
    show finite  $(bn \alpha)$  by (fact bn-finite)
  next
    show finite  $(supp (F, f, P))$  by (fact finite-supp)
  next
    show finite  $(supp (F, FL-Formula.Act f \alpha x))$  by (simp add: finite-supp)
  next
    from  $\langle bn \alpha \#* (F, f) \rangle$  show  $bn \alpha \#* (F, FL-Formula.Act f \alpha x)$ 
    by (simp add: fresh-star-Pair fresh-star-def fresh-def supp-Pair)
  qed metis
from 2 have supp F  $\#* p$  and Act-fresh: supp  $(FL-Formula.Act f \alpha x)$   $\#* p$ 
  by (simp add: fresh-star-Pair fresh-star-def supp-Pair)+
from  $\langle supp F \#* p \rangle$  have  $p \cdot F = F$ 
  by (metis supp-perm-eq)
from Act-fresh have  $p \cdot f = f$ 
  using fresh-star-Un supp-perm-eq by fastforce
from Act-fresh have eq: FL-Formula.Act f α x = FL-Formula.Act f (p · α)
(p · x)
  by (metis FL-Formula.Act-eq-iff-perm FL-Formula.Act-eqvt supp-perm-eq)

  with * obtain P' where trans: EF (F, P) →L (Eff f, P') and valid:
L-transform.valid P' (Formula.Act (Act (p · α)) (L-transform (p · x)))

```

```

using L-transform-Act by (metis L-transform.valid-Act-fresh bn-L-action.simps(2)
empty-iff fresh-star-def)
from trans have  $P'$ :  $P' = AC (f, F, \langle f \rangle P)$ 
by (simp add: residual-empty-bn-eq-iff)

have supp-f-P:  $supp (\langle f \rangle P) \subseteq supp f \cup supp P$ 
using effect-apply-eqvt supp-fun-app supp-fun-app-eqvt by fastforce
with 1 have  $bn (Act (p \cdot \alpha)) \#* AC (f, F, \langle f \rangle P)$ 
by (auto simp add: bn-eqvt fresh-star-def fresh-def supp-Pair)
with valid obtain  $P''$  where trans':  $AC (f, F, \langle f \rangle P) \rightarrow_L \langle Act (p \cdot \alpha), P'' \rangle$ 
and valid':  $L\text{-transform.valid } P'' (L\text{-transform } (p \cdot x))$ 
using  $P'$  by (metis L-transform.valid-Act-fresh)

from supp-f-P and 1 have  $bn (p \cdot \alpha) \#* (F, f, \langle f \rangle P)$ 
by (auto simp add: bn-eqvt fresh-star-def fresh-def supp-Pair)
with trans' obtain  $P'$  where  $P'' = EF (L (p \cdot \alpha, F, f), P')$  and trans'':
 $\langle f \rangle P \rightarrow \langle p \cdot \alpha, P' \rangle$ 
by (metis L-transition-AC-fresh)

from valid' have  $L\text{-transform.valid } (-p \cdot P'') (L\text{-transform } x)$ 
by (metis (mono-tags) L-transform.valid-eqvt L-transform-eqvt permute-minus-cancel(2))
with  $P'' \langle p \cdot F = F \rangle \langle p \cdot f = f \rangle$  have  $L\text{-transform.valid } (EF (L (\alpha, F, f),$ 
 $- p \cdot P')) (L\text{-transform } x)$ 
by simp (metis permute-minus-self permute-minus-cancel(1))
then have  $FL\text{-valid } P' (p \cdot x)$ 
using Act.hyps(4) by (metis FL-valid-eqvt permute-minus-cancel(1))

with trans'' and eq show  $FL\text{-valid } P (FL\text{-Formula.Act } f \alpha x)$ 
by (metis FL-valid-Act)
qed
qed

```

end

17.6 Bisimilarity in the L -transform

context *effect-nominal-ts*

begin

```

interpretation L-transform: nominal-ts ( $\vdash_L$ ) ( $\rightarrow_L$ )
by unfold-locales (fact L-satisfies-eqvt, fact L-transition-eqvt)

```

```

notation L-transform.bisimilar (infix  $\sim_L$  100)

```

F/L -bisimilarity is equivalent to bisimilarity in the L -transform.

```

inductive L-bisimilar :: ('state,'effect) L-state  $\Rightarrow$  ('state,'effect) L-state  $\Rightarrow$  bool
where

```

```

 $P \sim_L [F] Q \Longrightarrow L\text{-bisimilar } (EF (F,P)) (EF (F,Q))$ 

```

| $P \sim.[F] Q \implies f \in_{fs} F \implies L\text{-bisimilar } (AC(f, F, \langle f \rangle P)) (AC(f, F, \langle f \rangle Q))$

lemma *L-bisimilar-is-bisimulation: L-transform.is-bisimulation L-bisimilar*

unfolding *L-transform.is-bisimulation-def*

proof

show *symp L-bisimilar*

by (*metis FL-bisimilar-symp L-bisimilar.cases L-bisimilar.intros symp-def*)

next

have $\forall P_L Q_L. L\text{-bisimilar } P_L Q_L \longrightarrow (\forall \varphi. P_L \vdash_L \varphi \longrightarrow Q_L \vdash_L \varphi)$ (**is** ?S)

using *FL-bisimilar-is-L-bisimulation L-bisimilar.simps is-L-bisimulation-def*

by *auto*

moreover have $\forall P_L Q_L. L\text{-bisimilar } P_L Q_L \longrightarrow (\forall \alpha_L P_L'. \text{bn } \alpha_L \#^* Q_L \longrightarrow P_L \rightarrow_L \langle \alpha_L, P_L \rangle \longrightarrow (\exists Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L \rangle \wedge L\text{-bisimilar } P_L' Q_L'))$ (**is** ?T)

proof (*clarify*)

fix $P_L Q_L \alpha_L P_L'$

assume *L-bisim: L-bisimilar $P_L Q_L$ and fresh_L: bn $\alpha_L \#^* Q_L$ and trans_L: $P_L \rightarrow_L \langle \alpha_L, P_L \rangle$*

obtain Q_L' **where** $Q_L \rightarrow_L \langle \alpha_L, Q_L \rangle$ **and** *L-bisimilar $P_L' Q_L'$*

using *L-bisim proof (rule L-bisimilar.cases)*

fix $P F Q$

assume $P_L: P_L = EF(F, P)$ **and** $Q_L: Q_L = EF(F, Q)$ **and** *bisim: $P \sim.[F] Q$*

from P_L **and** *trans_L* **obtain** f **where** *effect: $f \in_{fs} F$ and $\alpha_L P_L': \langle \alpha_L, P_L \rangle = \langle Eff f, AC(f, F, \langle f \rangle P) \rangle$*

using *L-transition.simps(2) by blast*

from Q_L **and** *effect* **have** $Q_L \rightarrow_L \langle Eff f, AC(f, F, \langle f \rangle Q) \rangle$

using *L-transition.simps(2) by blast*

moreover from *bisim and effect* **have** *L-bisimilar $(AC(f, F, \langle f \rangle P)) (AC(f, F, \langle f \rangle Q))$*

using *L-bisimilar.intros(2) by blast*

moreover from $\alpha_L P_L'$ **have** $\alpha_L = Eff f$ **and** $P_L' = AC(f, F, \langle f \rangle P)$

by (*metis bn-L-action.simps(2) residual-empty-bn-eq-iff*)**+**

ultimately show *thesis*

using $\langle \bigwedge Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L \rangle \implies L\text{-bisimilar } P_L' Q_L' \implies \text{thesis} \rangle$

by *blast*

next

fix $P F Q f$

assume $P_L: P_L = AC(f, F, \langle f \rangle P)$ **and** $Q_L: Q_L = AC(f, F, \langle f \rangle Q)$

and *bisim: $P \sim.[F] Q$ and effect: $f \in_{fs} F$*

have *finite (supp $(\langle f \rangle Q, F, f)$)*

by (*fact finite-supp*)

with P_L **and** *trans_L* **obtain** $\alpha P'$ **where** *trans-P: $\langle f \rangle P \rightarrow \langle \alpha, P \rangle$ and $\alpha_L P_L': \langle \alpha_L, P_L \rangle = \langle Act \alpha, EF(L(\alpha, F, f), P') \rangle$ and fresh: bn $\alpha \#^* (\langle f \rangle Q, F, f)$*

by (*metis L-transition-AC-strong*)

from *bisim and effect and fresh* **and** *trans-P* **obtain** Q' **where** *trans-Q: $\langle f \rangle Q \rightarrow \langle \alpha, Q \rangle$ and bisim': $P' \sim.[L(\alpha, F, f)] Q'$*

by (*metis FL-bisimilar-simulation-step*)

from *fresh* **have** $\text{bn } \alpha \#^* (F, f)$

by (*meson fresh-PairD(2) fresh-star-def*)
with Q_L **and** *trans-Q* **have** $trans\text{-}Q_L: Q_L \rightarrow_L \langle Act\ \alpha, EF\ (L\ (\alpha, F, f), Q') \rangle$
by (*metis L-transition.simps(1)*)

from $\alpha_L P_L'$ **obtain** p **where** $p: (\alpha_L, P_L') = p \cdot (Act\ \alpha, EF\ (L\ (\alpha, F, f), P'))$ **and** $supp\text{-}p: supp\ p \subseteq bn\ \alpha \cup bn\ \alpha_L$
by (*metis (no-types, lifting) bn-L-action.simps(1) residual-eq-iff-perm-renaming*)
from $supp\text{-}p$ **and** *fresh* **and** $fresh_L$ **and** Q_L **have** $supp\ p \#^* (\langle f \rangle Q, F, f)$
unfolding *fresh-star-def* **by** (*metis (no-types, hide-lams) Un-iff fresh-Pair fresh-def subsetCE supp-AC*)
then **have** $p\text{-}fQ: p \cdot \langle f \rangle Q = \langle f \rangle Q$ **and** $p\text{-}Ff: p \cdot (F, f) = (F, f)$
by (*simp add: fresh-star-def perm-supp-eq+*)
from p **and** $p\text{-}Ff$ **have** $\alpha_L = Act\ (p \cdot \alpha)$ **and** $P_L' = EF\ (L\ (p \cdot \alpha, F, f), p \cdot P')$
by *auto*

moreover **from** Q_L **and** $p\text{-}fQ$ **and** $p\text{-}Ff$ **have** $p \cdot Q_L = Q_L$
by *simp*
with $trans\text{-}Q_L$ **have** $Q_L \rightarrow_L p \cdot \langle Act\ \alpha, EF\ (L\ (\alpha, F, f), Q') \rangle$
by (*metis L-transform.transition-eqt*)
then **have** $Q_L \rightarrow_L \langle Act\ (p \cdot \alpha), EF\ (L\ (p \cdot \alpha, F, f), p \cdot Q') \rangle$
using $p\text{-}Ff$ **by** *simp*

moreover **from** $p\text{-}Ff$ **have** $p \cdot F = F$ **and** $p \cdot f = f$
by *simp+*
with $bisim'$ **have** $(p \cdot P') \sim [L\ (p \cdot \alpha, F, f)]\ (p \cdot Q')$
by (*metis FL-bisimilar-eqt L-eqt'*)
then **have** $L\text{-bisimilar}\ (EF\ (L\ (p \cdot \alpha, F, f), p \cdot P'))\ (EF\ (L\ (p \cdot \alpha, F, f), p \cdot Q'))$
by (*metis L-bisimilar.intros(1)*)

ultimately **show** *thesis*
using $\langle \bigwedge Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \implies L\text{-bisimilar}\ P_L'\ Q_L' \implies thesis \rangle$
by *blast*

qed
then **show** $\exists Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \wedge L\text{-bisimilar}\ P_L'\ Q_L'$
by *auto*
qed
ultimately **show** $?S \wedge ?T$
by *metis*
qed

definition *invL-FL-bisimilar* :: $'effect\ first \Rightarrow 'state \Rightarrow 'state \Rightarrow bool$ **where**
 $invL\text{-}FL\text{-}bisimilar\ F\ P\ Q \equiv EF\ (F, P) \sim_L EF\ (F, Q)$

lemma *invL-FL-bisimilar-is-L-bisimulation*: *is-L-bisimulation invL-FL-bisimilar*
unfolding *is-L-bisimulation-def*
proof

fix F
have symp ($\text{invL-FL-bisimilar } F$) (**is** $?R$)
by ($\text{metis L-transform.bisimilar-symp invL-FL-bisimilar-def symp-def}$)
moreover have $\forall P Q. \text{invL-FL-bisimilar } F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))$ (**is** $?S$)
proof (clarify)
fix $P Q f \varphi$
assume $\text{bisim}: \text{invL-FL-bisimilar } F P Q$ **and** $\text{effect}: f \in_{fs} F$ **and** $\text{satisfies}: \langle f \rangle P \vdash \varphi$
from bisim **have** $EF (F, P) \sim_L EF (F, Q)$
by ($\text{metis invL-FL-bisimilar-def}$)
moreover have $\text{bn} (Eff f) \#^* EF (F, Q)$
by ($\text{simp add: fresh-star-def}$)
moreover from effect **have** $EF (F, P) \rightarrow_L \langle Eff f, AC (f, F, \langle f \rangle P) \rangle$
by ($\text{metis L-transition.simps}(2)$)
ultimately obtain Q_L' **where** $\text{trans}: EF (F, Q) \rightarrow_L \langle Eff f, Q_L' \rangle$ **and**
 $L\text{-bisim}: AC (f, F, \langle f \rangle P) \sim_L Q_L'$
by ($\text{metis L-transform.bisimilar-simulation-step}$)
from trans **obtain** f' **where** $\langle Eff f :: ('act, 'effect) L\text{-action}, Q_L' \rangle = \langle Eff f', AC (f', F, \langle f' \rangle Q) \rangle$
by ($\text{metis L-transition.simps}(2)$)
then have $Q_L': Q_L' = AC (f, F, \langle f \rangle Q)$
by ($\text{metis L-action.inject}(2) \text{bn-L-action.simps}(2) \text{residual-empty-bn-eq-iff}$)

from satisfies **have** $AC (f, F, \langle f \rangle P) \vdash_L \varphi$
by ($\text{metis L-satisfies.simps}(1)$)
with $L\text{-bisim}$ **and** Q_L' **have** $AC (f, F, \langle f \rangle Q) \vdash_L \varphi$
by ($\text{metis L-transform.bisimilar-is-bisimulation L-transform.is-bisimulation-def}$)
then show $\langle f \rangle Q \vdash \varphi$
by ($\text{metis L-satisfies.simps}(1)$)
qed
moreover have $\forall P Q. \text{invL-FL-bisimilar } F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#^* (\langle f \rangle Q, F, f) \longrightarrow \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge \text{invL-FL-bisimilar} (L (\alpha, F, f)) P' Q'))$ (**is** $?T$)
proof (clarify)
fix $P Q f \alpha P'$
assume $\text{bisim}: \text{invL-FL-bisimilar } F P Q$ **and** $\text{effect}: f \in_{fs} F$ **and** $\text{fresh}: \text{bn } \alpha \#^* (\langle f \rangle Q, F, f)$ **and** $\text{trans}: \langle f \rangle P \rightarrow \langle \alpha, P' \rangle$
from bisim **have** $EF (F, P) \sim_L EF (F, Q)$
by ($\text{metis invL-FL-bisimilar-def}$)
moreover have $\text{bn} (Eff f) \#^* EF (F, Q)$
by ($\text{simp add: fresh-star-def}$)
moreover from effect **have** $EF (F, P) \rightarrow_L \langle Eff f, AC (f, F, \langle f \rangle P) \rangle$
by ($\text{metis L-transition.simps}(2)$)
ultimately obtain Q_L' **where** $\text{trans}_L: EF (F, Q) \rightarrow_L \langle Eff f, Q_L' \rangle$ **and**
 $L\text{-bisim}: AC (f, F, \langle f \rangle P) \sim_L Q_L'$
by ($\text{metis L-transform.bisimilar-simulation-step}$)
from trans_L **obtain** f' **where** $\langle Eff f :: ('act, 'effect) L\text{-action}, Q_L' \rangle = \langle Eff f', AC (f', F, \langle f' \rangle Q) \rangle$

$f', AC (f', F, \langle f \rangle Q)$
 by (metis *L-transition.simps(2)*)
 then have $Q_L': Q_L' = AC (f, F, \langle f \rangle Q)$
 by (metis *L-action.inject(2)* *bn-L-action.simps(2)* *residual-empty-bn-eq-iff*)

from *L-bisim* and Q_L' have $AC (f, F, \langle f \rangle P) \sim_L AC (f, F, \langle f \rangle Q)$
 by *metis*
 moreover from *fresh* have $bn (Act \alpha) \#* AC (f, F, \langle f \rangle Q)$
 by (simp add: *fresh-def* *fresh-star-def* *supp-Pair*)
 moreover from *fresh* have $bn \alpha \#* (F, f)$
 by (simp add: *fresh-star-Pair*)
 with *trans* have $AC (f, F, \langle f \rangle P) \rightarrow_L \langle Act \alpha, EF (L (\alpha, F, f), P') \rangle$
 by (metis *L-transition.simps(1)*)
 ultimately obtain Q_L'' where $trans_L': AC (f, F, \langle f \rangle Q) \rightarrow_L \langle Act \alpha,$
 $Q_L'' \rangle$ and *L-bisim'*: $EF (L (\alpha, F, f), P') \sim_L Q_L''$
 by (metis *L-transform.bisimilar-simulation-step*)

have *finite* (*supp* ($\langle f \rangle Q, F, f$))
 by (*fact finite-supp*)
 with $trans_L'$ obtain $\alpha' Q'$ where $trans': \langle f \rangle Q \rightarrow \langle \alpha', Q' \rangle$ and *alpha*: $\langle Act$
 $\alpha :: ('act, 'effect) L\text{-action}, Q_L'' \rangle = \langle Act \alpha', EF (L (\alpha', F, f), Q') \rangle$ and *fresh'*: bn
 $\alpha' \#* (\langle f \rangle Q, F, f)$
 by (metis *L-transition-AC-strong*)

from *alpha* obtain p where $p: (Act \alpha :: ('act, 'effect) L\text{-action}, Q_L'') = p$
 $\cdot (Act \alpha', EF (L (\alpha', F, f), Q'))$ and *supp-p*: $supp p \subseteq bn \alpha \cup bn \alpha'$
 by (metis *Un-commute bn-L-action.simps(1)* *residual-eq-iff-perm-renaming*)
 from *supp-p* and *fresh* and *fresh'* have $supp p \#* (\langle f \rangle Q, F, f)$
 unfolding *fresh-star-def* by (metis (*no-types*, *hide-lams*) *Un-iff subsetCE*)
 then have $p \cdot fQ: p \cdot \langle f \rangle Q = \langle f \rangle Q$ and $p \cdot F: p \cdot F = F$ and $p \cdot f: p \cdot f = f$
 by (simp add: *fresh-star-def* *perm-supp-eq*)
 from p and $p \cdot F$ and $p \cdot f$ have $p \cdot \alpha': p \cdot \alpha' = \alpha$ and $Q_L'': Q_L'' = EF (L$
 $(p \cdot \alpha', F, f), p \cdot Q')$
 by *auto*

from $trans'$ and $p \cdot fQ$ and $p \cdot \alpha'$ have $\langle f \rangle Q \rightarrow \langle \alpha, p \cdot Q' \rangle$
 by (metis *transition-eqvt'*)
 moreover from *L-bisim'* and Q_L'' and $p \cdot \alpha'$ have *invL-FL-bisimilar* (L
 (α, F, f)) $P' (p \cdot Q')$
 by (metis *invL-FL-bisimilar-def*)
 ultimately show $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge$ *invL-FL-bisimilar* ($L (\alpha, F, f)$) P'
 Q'
 by *metis*
 qed

ultimately show $?R \wedge ?S \wedge ?T$
 by *metis*
 qed

theorem $P \sim_{[F]} Q \longleftrightarrow EF (F, P) \sim_L EF (F, Q)$

```

proof
  assume  $P \sim_{\cdot[F]} Q$ 
  then have  $L\text{-bisimilar } (EF (F,P)) (EF (F,Q))$ 
    by (metis L-bisimilar.intros(1))
  then show  $EF (F,P) \sim_L EF(F,Q)$ 
    by (metis L-bisimilar-is-bisimulation L-transform.bisimilar-def)
next
  assume  $EF (F, P) \sim_L EF (F, Q)$ 
  then have  $invL\text{-FL-bisimilar } F P Q$ 
    by (metis invL-FL-bisimilar-def)
  then show  $P \sim_{\cdot[F]} Q$ 
    by (metis invL-FL-bisimilar-is-L-bisimulation FL-bisimilar-def)
qed

```

end

The following (alternative) proof of the “ \leftarrow ” direction of this equivalence, namely that bisimilarity in the L -transform implies F/L -bisimilarity, uses the fact that the L -transform preserves satisfaction of formulas, together with the fact that bisimilarity (in the L -transform) implies logical equivalence. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system with effects where, additionally, the cardinality of the state set of the L -transform is bounded. We could re-organize our formalization to remove this assumption: the proof of $\llbracket indexed\text{-nominal-ts } TYPE(?'idx) ?satisfies ?transition; nominal\text{-ts.bisimilar } ?satisfies ?transition ?P ?Q \rrbracket \implies indexed\text{-nominal-ts.logically-equivalent } TYPE(?'idx) ?satisfies ?transition ?P ?Q$ does not actually make use of the cardinality assumptions provided by indexed nominal transition systems.

```

locale  $L\text{-transform-indexed-effect-nominal-ts} = indexed\text{-effect-nominal-ts } L \text{ satisfies transition effect-apply}$ 
  for  $L :: ('act::bn) \times ('effect::fs) fs\text{-set} \times 'effect \Rightarrow 'effect fs\text{-set}$ 
  and  $satisfies :: 'state::fs \Rightarrow 'pred::fs \Rightarrow bool$  (infix  $\vdash$  70)
  and  $transition :: 'state \Rightarrow ('act,'state) residual \Rightarrow bool$  (infix  $\rightarrow$  70)
  and  $effect\text{-apply} :: 'effect \Rightarrow 'state \Rightarrow 'state$  ( $\langle \cdot \rangle - [0,101] 100$ )  $+$ 
  assumes  $card\text{-idx-}L\text{-transform-state}: |UNIV::('state, 'effect) L\text{-state set}| < o |UNIV::'idx set|$ 
begin

```

interpretation $L\text{-transform}: indexed\text{-nominal-ts } (\vdash_L) (\rightarrow_L)$

by $unfold\text{-locales } (fact L\text{-satisfies-eqvt}, fact L\text{-transition-eqvt}, fact card\text{-idx-perm}, fact card\text{-idx-}L\text{-transform-state})$

notation $L\text{-transform.bisimilar}$ (**infix** \sim_L 100)

theorem $EF (F,P) \sim_L EF(F,Q) \longrightarrow P \sim_{\cdot[F]} Q$

proof

assume $EF (F, P) \sim_L EF (F, Q)$

```

then have L-transform.logically-equivalent (EF (F, P)) (EF (F, Q))
  by (fact L-transform.bisimilarity-implies-equivalence)
with FL-valid-iff-valid-L-transform have FL-logically-equivalent F P Q
  using FL-logically-equivalent-def L-transform.logically-equivalent-def by blast
then show P ~.[F] Q
  by (fact FL-equivalence-implies-bisimilarity)
qed

end

end

theory Weak-Transition-System
imports
  Transition-System
begin

```

18 Nominal Transition Systems and Bisimulations with Unobservable Transitions

18.1 Nominal transition systems with unobservable transitions

```

locale weak-nominal-ts = nominal-ts satisfies transition
  for satisfies :: 'state::fs ⇒ 'pred::fs ⇒ bool (infix ⊢ 70)
  and transition :: 'state ⇒ ('act::bn, 'state) residual ⇒ bool (infix → 70) +
  fixes τ :: 'act
  assumes tau-eqvt [eqvt]: p · τ = τ
begin

  lemma bn-tau-empty [simp]: bn τ = {}
  using bn-eqvt bn-finite tau-eqvt by (metis eqvt-def supp-finite-atom-set supp-fun-eqvt)

  lemma bn-tau-fresh [simp]: bn τ #* P
  by (simp add: fresh-star-def)

  inductive tau-transition :: 'state ⇒ 'state ⇒ bool (infix ⇒ 70) where
    tau-refl [simp]: P ⇒ P
  | tau-step: [ P → ⟨τ, P'⟩; P' ⇒ P'' ] ⇒ P ⇒ P''

  definition observable-transition :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/ ⇒{-}/ -
  [70, 70, 71] 71) where
    P ⇒{-} P' ≡ ∃ Q Q'. P ⇒ Q ∧ Q → ⟨α, Q'⟩ ∧ Q' ⇒ P'

  definition weak-transition :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/ ⇒{-}/ - [70, 70,
  71] 71) where
    P ⇒{-} P' ≡ if α = τ then P ⇒ P' else P ⇒{-} P'

```

The transition relations defined above are equivariant.

```

lemma tau-transition-eqvt :
  assumes  $P \Rightarrow P'$  shows  $p \cdot P \Rightarrow p \cdot P'$ 
using assms proof (induction)
  case (tau-refl  $P$ ) show ?case
    by (fact tau-transition.tau-refl)
next
  case (tau-step  $P P' P''$ )
    from  $\langle P \rightarrow \langle \tau, P' \rangle \rangle$  have  $p \cdot P \rightarrow \langle \tau, p \cdot P' \rangle$ 
      using tau-eqvt transition-eqvt' by fastforce
    with  $\langle p \cdot P' \Rightarrow p \cdot P'' \rangle$  show ?case
      using tau-transition.tau-step by blast
qed

```

```

lemma observable-transition-eqvt :
  assumes  $P \Rightarrow \{\alpha\} P'$  shows  $p \cdot P \Rightarrow \{p \cdot \alpha\} p \cdot P'$ 
using assms unfolding observable-transition-def by (metis transition-eqvt' tau-transition-eqvt)

```

```

lemma weak-transition-eqvt :
  assumes  $P \Rightarrow \langle \alpha \rangle P'$  shows  $p \cdot P \Rightarrow \langle p \cdot \alpha \rangle p \cdot P'$ 
using assms unfolding weak-transition-def by (metis (full-types) observable-transition-eqvt
permute-minus-cancel(2) tau-eqvt tau-transition-eqvt)

```

Additional lemmas about (\Rightarrow) , *observable-transition* and *weak-transition*.

```

lemma tau-transition-trans:
  assumes  $P \Rightarrow Q$  and  $Q \Rightarrow R$ 
  shows  $P \Rightarrow R$ 
using assms by (induction, auto simp add: tau-step)

```

```

lemma observable-transitionI:
  assumes  $P \Rightarrow Q$  and  $Q \rightarrow \langle \alpha, Q' \rangle$  and  $Q' \Rightarrow P'$ 
  shows  $P \Rightarrow \{\alpha\} P'$ 
using assms observable-transition-def by blast

```

```

lemma observable-transition-stepI [simp]:
  assumes  $P \rightarrow \langle \alpha, P' \rangle$ 
  shows  $P \Rightarrow \{\alpha\} P'$ 
using assms observable-transitionI tau-refl by blast

```

```

lemma observable-tau-transition:
  assumes  $P \Rightarrow \{\tau\} P'$ 
  shows  $P \Rightarrow P'$ 
proof –
  from  $\langle P \Rightarrow \{\tau\} P' \rangle$  obtain  $Q Q'$  where  $P \Rightarrow Q$  and  $Q \rightarrow \langle \tau, Q' \rangle$  and  $Q' \Rightarrow P'$ 
    unfolding observable-transition-def by blast
  then show ?thesis
    by (metis tau-step tau-transition-trans)
qed

```

lemma *weak-transition-tau-iff* [simp]:

$P \Rightarrow \langle \tau \rangle P' \iff P \Rightarrow P'$

by (*simp add: weak-transition-def*)

lemma *weak-transition-not-tau-iff* [simp]:

assumes $\alpha \neq \tau$

shows $P \Rightarrow \langle \alpha \rangle P' \iff P \Rightarrow \{\alpha\} P'$

using *assms* **by** (*simp add: weak-transition-def*)

lemma *weak-transition-stepI* [simp]:

assumes $P \Rightarrow \{\alpha\} P'$

shows $P \Rightarrow \langle \alpha \rangle P'$

using *assms* **by** (*cases* $\alpha = \tau$, *simp-all add: observable-tau-transition*)

lemma *weak-transition-weakI*:

assumes $P \Rightarrow Q$ **and** $Q \Rightarrow \langle \alpha \rangle Q'$ **and** $Q' \Rightarrow P'$

shows $P \Rightarrow \langle \alpha \rangle P'$

proof (*cases* $\alpha = \tau$)

case *True* **with** *assms* **show** *?thesis*

by (*metis tau-transition-trans weak-transition-tau-iff*)

next

case *False* **with** *assms* **show** *?thesis*

using *observable-transition-def tau-transition-trans weak-transition-not-tau-iff*

by *blast*

qed

end

18.2 Weak bisimulations

context *weak-nominal-ts*

begin

definition *is-weak-bisimulation* :: ('state \Rightarrow 'state \Rightarrow bool) \Rightarrow bool **where**

is-weak-bisimulation $R \equiv$

symp $R \wedge$

— weak static implication

$(\forall P Q \varphi. R P Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge R P Q' \wedge Q' \vdash \varphi)) \wedge$

— weak simulation

$(\forall P Q. R P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge R P' Q'))))$

definition *weakly-bisimilar* :: 'state \Rightarrow 'state \Rightarrow bool (**infix** $\approx \cdot$ 100) **where**

$P \approx \cdot Q \equiv \exists R. \text{is-weak-bisimulation } R \wedge R P Q$

$(\approx \cdot)$ is an equivariant equivalence relation.

lemma *is-weak-bisimulation-eqvt* :

assumes *is-weak-bisimulation* R **shows** *is-weak-bisimulation* $(p \cdot R)$

using *assms* **unfolding** *is-weak-bisimulation-def*

proof (*clarify*)
assume 1: *symp* R
assume 2: $\forall P Q \varphi. R P Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge R P Q' \wedge Q' \vdash \varphi)$
assume 3: $\forall P Q. R P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge R P' Q'))$
have *symp* $(p \cdot R)$ (**is** ? S)
using 1 **by** (*simp add: symp-eqvt*)
moreover have $\forall P Q \varphi. (p \cdot R) P Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge (p \cdot R) P Q' \wedge Q' \vdash \varphi)$ (**is** ? T)
proof (*clarify*)
fix $P Q \varphi$
assume $pR: (p \cdot R) P Q$ **and** $phi: P \vdash \varphi$
from pR **have** $R (-p \cdot P) (-p \cdot Q)$
by (*simp add: eqvt-lambda permute-bool-def unpermute-def*)
moreover from phi **have** $(-p \cdot P) \vdash (-p \cdot \varphi)$
by (*metis satisfies-eqvt*)
ultimately obtain Q' **where** $*$: $-p \cdot Q \Rightarrow Q'$ **and** $**$: $R (-p \cdot P) Q'$
and $***$: $Q' \vdash (-p \cdot \varphi)$
using 2 **by** *blast*
from $*$ **have** $Q \Rightarrow p \cdot Q'$
by (*metis permute-minus-cancel(1) tau-transition-eqvt*)
moreover from $**$ **have** $(p \cdot R) P (p \cdot Q')$
by (*simp add: eqvt-lambda permute-bool-def unpermute-def*)
moreover from $***$ **have** $p \cdot Q' \vdash \varphi$
by (*metis permute-minus-cancel(1) satisfies-eqvt*)
ultimately show $\exists Q'. Q \Rightarrow Q' \wedge (p \cdot R) P Q' \wedge Q' \vdash \varphi$
by *metis*
qed
moreover have $\forall P Q. (p \cdot R) P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge (p \cdot R) P' Q'))$ (**is** ? U)
proof (*clarify*)
fix $P Q \alpha P'$
assume $*$: $(p \cdot R) P Q$ **and** $**$: $\text{bn } \alpha \#* Q$ **and** $***$: $P \rightarrow \langle \alpha, P' \rangle$
from $*$ **have** $R (-p \cdot P) (-p \cdot Q)$
by (*simp add: eqvt-lambda permute-bool-def unpermute-def*)
moreover have $\text{bn } (-p \cdot \alpha) \#* (-p \cdot Q)$
using $**$ **by** (*metis bn-eqvt fresh-star-permute-iff*)
moreover have $-p \cdot P \rightarrow \langle -p \cdot \alpha, -p \cdot P' \rangle$
using $***$ **by** (*metis transition-eqvt'*)
ultimately obtain Q' **where** T : $-p \cdot Q \Rightarrow \langle -p \cdot \alpha \rangle Q'$ **and** R : $R (-p \cdot P) Q'$
using 3 **by** *metis*
from T **have** $Q \Rightarrow \langle \alpha \rangle (p \cdot Q')$
by (*metis permute-minus-cancel(1) weak-transition-eqvt*)
moreover from R **have** $(p \cdot R) P' (p \cdot Q')$
by (*metis eqvt-apply eqvt-lambda permute-bool-def unpermute-def*)
ultimately show $\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge (p \cdot R) P' Q'$
by *metis*
qed

ultimately show $?S \wedge ?T \wedge ?U$ **by** *simp*
qed

lemma *weakly-bisimilar-eqt* :
assumes $P \approx \cdot Q$ **shows** $(p \cdot P) \approx \cdot (p \cdot Q)$
proof –

from *assms* **obtain** R **where** $*$: *is-weak-bisimulation* $R \wedge R P Q$
unfolding *weakly-bisimilar-def* ..
then have *is-weak-bisimulation* $(p \cdot R)$
by (*simp add: is-weak-bisimulation-eqt*)
moreover from $*$ **have** $(p \cdot R) (p \cdot P) (p \cdot Q)$
by (*metis eqvt-apply permute-boolI*)
ultimately show $(p \cdot P) \approx \cdot (p \cdot Q)$
unfolding *weakly-bisimilar-def* **by** *auto*
qed

lemma *weakly-bisimilar-reflp: reflp weakly-bisimilar*
proof (*rule reflpI*)

fix x
have *is-weak-bisimulation* $(=)$
unfolding *is-weak-bisimulation-def* **by** (*simp add: symp-def*)
then show $x \approx \cdot x$
unfolding *weakly-bisimilar-def* **by** *auto*
qed

lemma *weakly-bisimilar-symp: symp weakly-bisimilar*
proof (*rule sympI*)

fix $P Q$
assume $P \approx \cdot Q$
then obtain R **where** $*$: *is-weak-bisimulation* $R \wedge R P Q$
unfolding *weakly-bisimilar-def* ..
then have $R Q P$
unfolding *is-weak-bisimulation-def* **by** (*simp add: symp-def*)
with $*$ **show** $Q \approx \cdot P$
unfolding *weakly-bisimilar-def* **by** *auto*
qed

lemma *weakly-bisimilar-is-weak-bisimulation: is-weak-bisimulation weakly-bisimilar*
unfolding *is-weak-bisimulation-def* **proof**

show *symp* $(\approx \cdot)$
by (*fact weakly-bisimilar-symp*)
next
show $(\forall P Q \varphi. P \approx \cdot Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge P \approx \cdot Q' \wedge Q' \vdash \varphi)) \wedge$
 $(\forall P Q. P \approx \cdot Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \# * Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha$
 $Q' \wedge P' \approx \cdot Q')))$
by (*auto simp add: is-weak-bisimulation-def weakly-bisimilar-def*) *blast+*
qed

lemma *weakly-bisimilar-tau-simulation-step*:

```

    assumes  $P \approx \cdot Q$  and  $P \Rightarrow P'$ 
    obtains  $Q'$  where  $Q \Rightarrow Q'$  and  $P' \approx \cdot Q'$ 
using  $\langle P \Rightarrow P' \rangle \langle P \approx \cdot Q \rangle$  proof (induct arbitrary:  $Q$ )
  case (tau-refl  $P$ ) then show ?case
    by (metis tau-transition.tau-refl)
next
  case (tau-step  $P P'' P'$ )
  from  $\langle P \rightarrow \langle \tau, P'' \rangle \rangle$  and  $\langle P \approx \cdot Q \rangle$  obtain  $Q''$  where  $Q \Rightarrow Q''$  and  $P'' \approx \cdot Q''$ 
  by (metis bn-tau-fresh is-weak-bisimulation-def weak-transition-def weakly-bisimilar-is-weak-bisimulation)
  then show ?case
    using tau-step.hyps(3) tau-step.prem(1) by (metis tau-transition-trans)
qed

lemma weakly-bisimilar-weak-simulation-step:
  assumes  $P \approx \cdot Q$  and  $\text{bn } \alpha \#* Q$  and  $P \Rightarrow \langle \alpha \rangle P'$ 
  obtains  $Q'$  where  $Q \Rightarrow \langle \alpha \rangle Q'$  and  $P' \approx \cdot Q'$ 
proof (cases  $\alpha = \tau$ )
  case True with  $\langle P \approx \cdot Q \rangle$  and  $\langle P \Rightarrow \langle \alpha \rangle P' \rangle$  and that show ?thesis
    using weak-transition-tau-iff weakly-bisimilar-tau-simulation-step by force
  next
  case False with  $\langle P \Rightarrow \langle \alpha \rangle P' \rangle$  have  $P \Rightarrow \{\alpha\} P'$ 
    by simp
  then obtain  $P1 P2$  where tauP:  $P \Rightarrow P1$  and trans:  $P1 \rightarrow \langle \alpha, P2 \rangle$  and
  tauP2:  $P2 \Rightarrow P'$ 
    using observable-transition-def by blast
  from  $\langle P \approx \cdot Q \rangle$  and tauP obtain  $Q1$  where tauQ:  $Q \Rightarrow Q1$  and  $P1Q1: P1$ 
   $\approx \cdot Q1$ 
    by (metis weakly-bisimilar-tau-simulation-step)

  — rename  $\langle \alpha, P2 \rangle$  to avoid  $Q1$ , without touching  $Q$ 
  obtain  $p$  where 1:  $(p \cdot \text{bn } \alpha) \#* Q1$  and 2:  $\text{supp } (\langle \alpha, P2 \rangle, Q) \#* p$ 
  proof (rule at-set-avoiding2[of  $\text{bn } \alpha Q1 (\langle \alpha, P2 \rangle, Q)$ , THEN exE])
    show finite (bn  $\alpha$ ) by (fact bn-finite)
  next
    show finite (supp  $Q1$ ) by (fact finite-supp)
  next
    show finite (supp  $(\langle \alpha, P2 \rangle, Q)$ ) by (simp add: finite-supp supp-Pair)
  next
    show  $\text{bn } \alpha \#* (\langle \alpha, P2 \rangle, Q)$  using (bn  $\alpha \#* Q$ ) by (simp add: fresh-star-Pair)
  qed metis

  from 2 have 3:  $\text{supp } \langle \alpha, P2 \rangle \#* p$  and 4:  $\text{supp } Q \#* p$ 
  by (simp add: fresh-star-Un supp-Pair)+
  from 3 have  $\langle p \cdot \alpha, p \cdot P2 \rangle = \langle \alpha, P2 \rangle$ 
  using supp-perm-eq by fastforce
  then obtain  $Q2$  where trans':  $Q1 \Rightarrow \langle p \cdot \alpha \rangle Q2$  and P2Q2:  $(p \cdot P2) \approx \cdot Q2$ 
  using P1Q1 trans 1 by (metis (mono-tags, lifting) weakly-bisimilar-is-weak-bisimulation
  bn-eqvt is-weak-bisimulation-def)

  from tauP2 have  $p \cdot P2 \Rightarrow p \cdot P'$ 

```


by (fact tau-transition-eqt)
 with $P2Q2$ obtain Q' where $\text{tau}Q2: Q2 \Rightarrow Q'$ and $P'Q': (p \cdot P') \approx \cdot Q'$
 by (metis weakly-bisimilar-tau-simulation-step)

from $\text{tau}Q$ and trans' and $\text{tau}Q2$ have $Q \Rightarrow \langle p \cdot \alpha \rangle Q'$
 by (rule weak-transition-weakI)
 with \downarrow have $Q \Rightarrow \langle \alpha \rangle (-p \cdot Q')$
 by (metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqt)
 moreover from $P'Q'$ have $P' \approx \cdot (-p \cdot Q')$
 by (metis permute-minus-cancel(2) weakly-bisimilar-eqt)
 ultimately show ?thesis ..

qed

lemma weakly-bisimilar-transp: transp weakly-bisimilar
 proof (rule transpI)
 fix $P Q R$
 assume $PQ: P \approx \cdot Q$ and $QR: Q \approx \cdot R$
 let ?bisim = weakly-bisimilar OO weakly-bisimilar
 have symp ?bisim
 proof (rule sympI)
 fix $P R$
 assume ?bisim $P R$
 then obtain Q where $P \approx \cdot Q$ and $Q \approx \cdot R$
 by blast
 then have $R \approx \cdot Q$ and $Q \approx \cdot P$
 by (metis weakly-bisimilar-symp sympE)+
 then show ?bisim $R P$
 by blast

qed

moreover have $\forall P Q \varphi. ?bisim P Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge ?bisim P Q' \wedge Q' \vdash \varphi)$
 proof (clarify)
 fix $P Q \varphi R$
 assume $\text{phi}: P \vdash \varphi$ and $PR: P \approx \cdot R$ and $RQ: R \approx \cdot Q$
 from PR and phi obtain R' where $R \Rightarrow R'$ and $P \approx \cdot R'$ and $*$: $R' \vdash \varphi$
 using weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by

force

from RQ and $\langle R \Rightarrow R' \rangle$ obtain Q' where $Q \Rightarrow Q'$ and $R' \approx \cdot Q'$
 by (metis weakly-bisimilar-tau-simulation-step)
 from $\langle R' \approx \cdot Q' \rangle$ and $*$ obtain Q'' where $Q' \Rightarrow Q''$ and $R' \approx \cdot Q''$ and
 $**$: $Q'' \vdash \varphi$
 using weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by

force

from $\langle Q \Rightarrow Q' \rangle$ and $\langle Q' \Rightarrow Q'' \rangle$ have $Q \Rightarrow Q''$
 by (fact tau-transition-trans)
 moreover from $\langle P \approx \cdot R' \rangle$ and $\langle R' \approx \cdot Q'' \rangle$ have ?bisim $P Q''$
 by blast
 ultimately show $\exists Q'. Q \Rightarrow Q' \wedge ?bisim P Q' \wedge Q' \vdash \varphi$
 using ** by metis

qed
moreover have $\forall P Q. ?bisim P Q \longrightarrow (\forall \alpha P'. bn \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle)$
 $\longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge ?bisim P' Q')$
proof (*clarify*)
fix $P Q R \alpha P'$
assume $PR: P \approx \cdot R$ **and** $RQ: R \approx \cdot Q$ **and** $fresh: bn \alpha \#* Q$ **and** $trans: P$
 $\rightarrow \langle \alpha, P' \rangle$
— rename $\langle \alpha, P' \rangle$ to avoid R , without touching Q
obtain p **where** $1: (p \cdot bn \alpha) \#* R$ **and** $2: supp (\langle \alpha, P' \rangle, Q) \#* p$
proof (*rule at-set-avoiding2[of bn α $R (\langle \alpha, P' \rangle, Q)$, THEN exE]*)
show *finite* $(bn \alpha)$ **by** (*fact bn-finite*)
next
show *finite* $(supp R)$ **by** (*fact finite-supp*)
next
show *finite* $(supp (\langle \alpha, P' \rangle, Q))$ **by** (*simp add: finite-supp supp-Pair*)
next
show $bn \alpha \#* (\langle \alpha, P' \rangle, Q)$ **by** (*simp add: fresh fresh-star-Pair*)
qed *metis*
from 2 **have** $3: supp \langle \alpha, P' \rangle \#* p$ **and** $4: supp Q \#* p$
by (*simp add: fresh-star-Un supp-Pair*)
from 3 **have** $\langle p \cdot \alpha, p \cdot P' \rangle = \langle \alpha, P' \rangle$
using *supp-perm-eq* **by** *fastforce*
with $trans$ **obtain** pR' **where** $5: R \Rightarrow \langle p \cdot \alpha \rangle pR'$ **and** $6: (p \cdot P') \approx \cdot pR'$
using PR 1 **by** (*metis bn-eqvt weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def*)
from $fresh$ **and** 4 **have** $bn (p \cdot \alpha) \#* Q$
by (*metis bn-eqvt fresh-star-permute-iff supp-perm-eq*)
then **obtain** pQ' **where** $7: Q \Rightarrow \langle p \cdot \alpha \rangle pQ'$ **and** $8: pR' \approx \cdot pQ'$
using RQ 5 **by** (*metis weakly-bisimilar-weak-simulation-step*)
from 7 **have** $Q \Rightarrow \langle \alpha \rangle (-p \cdot pQ')$
using 4 **by** (*metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqvt*)
moreover **from** 6 **and** 8 **have** $?bisim P' (-p \cdot pQ')$
by (*metis (no-types, hide-lams) weakly-bisimilar-eqvt permute-minus-cancel(2) relcompp.simps*)
ultimately **show** $\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge ?bisim P' Q'$
by *metis*
qed
ultimately **have** *is-weak-bisimulation* $?bisim$
unfolding *is-weak-bisimulation-def* **by** *metis*
moreover **have** $?bisim P R$
using $PQ QR$ **by** *blast*
ultimately **show** $P \approx \cdot R$
unfolding *weakly-bisimilar-def* **by** *meson*
qed

lemma *weakly-bisimilar-equivp: equivp weakly-bisimilar*
by (*metis weakly-bisimilar-reflp weakly-bisimilar-symp weakly-bisimilar-transp equivp-reflp-symp-transp*)

end

```

end
theory Weak-Formula
imports
  Weak-Transition-System
  Formula
begin

```

19 Weak Formulas

19.1 Disjunction

definition $Disj :: ('idx, 'pred :: fs, 'act :: bn) formula set['idx] \Rightarrow ('idx, 'pred, 'act) formula$ **where**

$Disj\ xset = Not\ (Conj\ (map\ bset\ Not\ xset))$

lemma $finite\ supp\ map\ bset\ Not$ [simp]:

assumes $finite\ (supp\ xset)$

shows $finite\ (supp\ (map\ bset\ Not\ xset))$

proof –

have $eqvt\ map\ bset$ **and** $eqvt\ Not$

by (simp add: eqvtI)+

then have $supp\ (map\ bset\ Not) = \{\}$

using $supp\ fun\ eqvt\ supp\ fun\ app\ eqvt$ **by** blast

then have $supp\ (map\ bset\ Not\ xset) \subseteq supp\ xset$

using $supp\ fun\ app$ **by** blast

with $assms$ **show** $finite\ (supp\ (map\ bset\ Not\ xset))$

by (metis finite-subset)

qed

lemma $Disj\ eqvt$ [simp]:

assumes $finite\ (supp\ xset)$

shows $p \cdot Disj\ xset = Disj\ (p \cdot xset)$

using $assms$ **unfolding** $Disj\ def$ **by** simp

lemma $Disj\ eq\ iff$ [simp]:

assumes $finite\ (supp\ xset1)$ **and** $finite\ (supp\ xset2)$

shows $Disj\ xset1 = Disj\ xset2 \iff xset1 = xset2$

using $assms$ **unfolding** $Disj\ def$ **by** (metis Conj-eq-iff Not-eq-iff bset.inj-map-strong finite-supp-map-bset-Not)

19.2 Lemmas about α -equivalence involving τ

context $weak\ nominal\ ts$

begin

lemma $Act\ tau\ eq\ iff$ [simp]:

$Act\ \tau\ x1 = Act\ \alpha\ x2 \iff \alpha = \tau \wedge x2 = x1$

(is ?l \iff ?r)

```

proof
  assume ?l
  then obtain  $p$  where  $p\text{-}\alpha: p \cdot \tau = \alpha$  and  $p\text{-}x: p \cdot x1 = x2$  and  $\text{fresh}: (\text{supp } x1 - \text{bn } \tau) \#* p$ 
    by (metis Act-eq-iff-perm)
  from  $p\text{-}\alpha$  have  $\alpha = \tau$ 
    by (metis tau-eqvt)
  moreover from  $\text{fresh}$  and  $p\text{-}x$  have  $x2 = x1$ 
    by (simp add: supp-perm-eq)
  ultimately show ?r ..
next
  assume ?r then show ?l
    by simp
qed

```

end

19.3 Weak action modality

The definition of (strong) formulas is parametric in the index type, but from now on we want to work with a fixed (sufficiently large) index type.

Also, we use τ in our definition of weak formulas.

```

locale indexed-weak-nominal-ts = weak-nominal-ts satisfies transition
  for satisfies :: 'state::fs  $\Rightarrow$  'pred::fs  $\Rightarrow$  bool (infix  $\vdash$  70)
  and transition :: 'state  $\Rightarrow$  ('act::bn, 'state) residual  $\Rightarrow$  bool (infix  $\rightarrow$  70) +
  assumes card-idx-perm:  $|UNIV::\text{perm set}| < o |UNIV::'\text{idx set}|$ 
    and card-idx-state:  $|UNIV::'\text{state set}| < o |UNIV::'\text{idx set}|$ 
    and card-idx-nat:  $|UNIV::\text{nat set}| < o |UNIV::'\text{idx set}|$ 
begin

```

The assumption $|UNIV| < o |UNIV|$ is redundant: it is already implied by $|UNIV| < o |UNIV|$. A formal proof of this fact is left for future work.

```

lemma card-idx-nat' [simp]:
   $|UNIV::\text{nat set}| < o \text{natLeq} + c |UNIV::'\text{idx set}|$ 
proof -
  note card-idx-nat
  also have  $|UNIV::'\text{idx set}| \leq o \text{natLeq} + c |UNIV::'\text{idx set}|$ 
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally show ?thesis .
qed

```

```

primrec tau-steps :: ('idx, 'pred::fs, 'act::bn) formula  $\Rightarrow$  nat  $\Rightarrow$  ('idx, 'pred, 'act)
formula
  where
     $\text{tau-steps } x \ 0 = x$ 
     $|\ \text{tau-steps } x \ (\text{Suc } n) = \text{Act } \tau \ (\text{tau-steps } x \ n)$ 

```

```

lemma tau-steps-eqvt [eqvt]:

```

$p \cdot \text{tau-steps } x \ n = \text{tau-steps } (p \cdot x) \ (p \cdot n)$
by (*induct n*) (*simp-all add: permute-nat-def tau-eqvt*)

lemma *tau-steps-add* [*simp*]:
 $\text{tau-steps } (\text{tau-steps } x \ m) \ n = \text{tau-steps } x \ (m + n)$
by (*induct n*) *auto*

lemma *tau-steps-not-self*:
 $x = \text{tau-steps } x \ n \longleftrightarrow n = 0$
proof
assume $x = \text{tau-steps } x \ n$ **then show** $n = 0$
proof (*induct n arbitrary: x*)
case 0 **show** *?case ..*
next
case (*Suc n*)
then have $x = \text{Act } \tau \ (\text{tau-steps } x \ n)$
by *simp*
then show $\text{Suc } n = 0$
proof (*induct x*)
case (*Act α x*)
then have $x = \text{tau-steps } (\text{Act } \tau \ x) \ n$
by (*metis Act-tau-eq-iff*)
with *Act.hyps* **show** *?thesis*
by (*metis add-Suc tau-steps.simps(2) tau-steps-add*)
qed *simp-all*
qed
next
assume $n = 0$ **then show** $x = \text{tau-steps } x \ n$
by *simp*
qed

definition *weak-tau-modality* :: (*'idx, 'pred::fs, 'act::bn*) *formula* \Rightarrow (*'idx, 'pred, 'act*) *formula*

where
 $\text{weak-tau-modality } x \equiv \text{Disj } (\text{map-bset } (\text{tau-steps } x) \ (\text{Abs-bset } \text{UNIV}))$

lemma *finite-supp-map-bset-tau-steps* [*simp*]:
 $\text{finite } (\text{supp } (\text{map-bset } (\text{tau-steps } x) \ (\text{Abs-bset } \text{UNIV} :: \text{nat set}['idx])))$
proof –
have *eqvt map-bset and eqvt tau-steps*
by (*simp add: eqvtI*)
then have $\text{supp } (\text{map-bset } (\text{tau-steps } x)) \subseteq \text{supp } x$
using *supp-fun-eqvt supp-fun-app supp-fun-app-eqvt* **by** *blast*
moreover have $\text{supp } (\text{Abs-bset } \text{UNIV} :: \text{nat set}['idx]) = \{\}$
by (*simp add: eqvtI supp-fun-eqvt*)
ultimately have $\text{supp } (\text{map-bset } (\text{tau-steps } x) \ (\text{Abs-bset } \text{UNIV} :: \text{nat set}['idx]))$
 $\subseteq \text{supp } x$
using *supp-fun-app* **by** *blast*
then show *?thesis*

by (*metis finite-subset finite-supp*)
qed

lemma *weak-tau-modality-eqvt* [*eqvt*]:
 $p \cdot \text{weak-tau-modality } x = \text{weak-tau-modality } (p \cdot x)$
unfolding *weak-tau-modality-def* **by** (*simp add: map-bset-eqvt*)

lemma *weak-tau-modality-eq-iff* [*simp*]:
 $\text{weak-tau-modality } x = \text{weak-tau-modality } y \iff x = y$

proof
assume $\text{weak-tau-modality } x = \text{weak-tau-modality } y$
then have $\text{map-bset } (\text{tau-steps } x) (\text{Abs-bset UNIV} :: \text{set}['\text{id}x]) = \text{map-bset}$
 $(\text{tau-steps } y) (\text{Abs-bset UNIV})$
unfolding *weak-tau-modality-def* **by** *simp*
with *card-idx-nat'* **have** $\text{range } (\text{tau-steps } x) = \text{range } (\text{tau-steps } y)$
(is $?X = ?Y$ **)**
by (*metis Abs-bset-inverse' map-bset.rep-eq*)
then have $x \in \text{range } (\text{tau-steps } y)$ **and** $y \in \text{range } (\text{tau-steps } x)$
by (*metis range-eqI tau-steps.simps(1)+*)
then obtain $nx \ ny$ **where** $x = \text{tau-steps } y \ nx$ **and** $y = \text{tau-steps } x \ ny$
by *blast*
then have $ny + nx = 0$
by (*simp add: tau-steps-not-self*)
with x **and** y **show** $x = y$
by *simp*
next
assume $x = y$ **then show** $\text{weak-tau-modality } x = \text{weak-tau-modality } y$
by *simp*
qed

lemma *supp-weak-tau-modality* [*simp*]:
 $\text{supp } (\text{weak-tau-modality } x) = \text{supp } x$
unfolding *supp-def* **by** *simp*

lemma *Act-weak-tau-modality-eq-iff* [*simp*]:
 $\text{Act } \alpha 1 (\text{weak-tau-modality } x1) = \text{Act } \alpha 2 (\text{weak-tau-modality } x2) \iff \text{Act } \alpha 1$
 $x1 = \text{Act } \alpha 2 x2$
by (*simp add: Act-eq-iff-perm*)

definition *weak-action-modality* :: $'act \Rightarrow ('id_x, 'pred :: fs, 'act :: bn)$ *formula* \Rightarrow
 $('id_x, 'pred, 'act)$ *formula* ($\langle\langle - \rangle\rangle$)
where
 $\langle\langle \alpha \rangle\rangle x \equiv \text{if } \alpha = \tau \text{ then weak-tau-modality } x \text{ else weak-tau-modality } (\text{Act } \alpha$
 $(\text{weak-tau-modality } x))$

lemma *weak-action-modality-eqvt* [*eqvt*]:
 $p \cdot \langle\langle \alpha \rangle\rangle x = \langle\langle p \cdot \alpha \rangle\rangle (p \cdot x)$
by (*simp add: weak-action-modality-def*)

lemma *weak-action-modality-tau*:
 $\langle\langle\tau\rangle\rangle x = \text{weak-tau-modality } x$
unfolding *weak-action-modality-def* **by** *simp*

lemma *weak-action-modality-not-tau*:
assumes $\alpha \neq \tau$
shows $\langle\langle\alpha\rangle\rangle x = \text{weak-tau-modality } (\text{Act } \alpha (\text{weak-tau-modality } x))$
using *assms* **unfolding** *weak-action-modality-def* **by** *simp*

Equality is modulo α -equivalence.

Note that the converse of the following lemma does not hold. For instance, for $\alpha \neq \tau$ we have $\langle\langle\tau\rangle\rangle \text{Act } \alpha (\text{weak-tau-modality } x) = \langle\langle\alpha\rangle\rangle x$ by definition, but clearly not $\text{Act } \tau (\text{Act } \alpha (\text{weak-tau-modality } x)) = \text{Act } \alpha x$.

lemma *weak-action-modality-eq*:
assumes $\text{Act } \alpha 1 x 1 = \text{Act } \alpha 2 x 2$
shows $\langle\langle\alpha 1\rangle\rangle x 1 = \langle\langle\alpha 2\rangle\rangle x 2$
proof (*cases* $\alpha 1 = \tau$)
case *True*
with *assms* **have** $\alpha 2 = \alpha 1 \wedge x 2 = x 1$
by (*metis* *Act-tau-eq-iff*)
then show *?thesis*
by *simp*
next
case *False*
from *assms* **obtain** p **where** $1: \text{supp } x 1 - \text{bn } \alpha 1 = \text{supp } x 2 - \text{bn } \alpha 2$ **and**
 $2: (\text{supp } x 1 - \text{bn } \alpha 1) \#* p$
and $3: p \cdot x 1 = x 2$ **and** $4: \text{supp } \alpha 1 - \text{bn } \alpha 1 = \text{supp } \alpha 2 - \text{bn } \alpha 2$ **and** $5:$
 $(\text{supp } \alpha 1 - \text{bn } \alpha 1) \#* p$
and $6: p \cdot \alpha 1 = \alpha 2$
by (*metis* *Act-eq-iff-perm*)
from 1 **have** $\text{supp } (\text{weak-tau-modality } x 1) - \text{bn } \alpha 1 = \text{supp } (\text{weak-tau-modality } x 2) - \text{bn } \alpha 2$
by (*metis* *supp-weak-tau-modality*)
moreover from 2 **have** $(\text{supp } (\text{weak-tau-modality } x 1) - \text{bn } \alpha 1) \#* p$
by (*metis* *supp-weak-tau-modality*)
moreover from 3 **have** $p \cdot \text{weak-tau-modality } x 1 = \text{weak-tau-modality } x 2$
by (*metis* *weak-tau-modality-eqvt*)
ultimately have $\text{Act } \alpha 1 (\text{weak-tau-modality } x 1) = \text{Act } \alpha 2 (\text{weak-tau-modality } x 2)$
using 4 **and** 5 **and** 6 **and** *Act-eq-iff-perm* **by** *blast*
moreover from $\langle\alpha 1 \neq \tau\rangle$ **and** *assms* **have** $\alpha 2 \neq \tau$
by (*metis* *Act-tau-eq-iff*)
ultimately show *?thesis*
using $\langle\alpha 1 \neq \tau\rangle$ **by** (*simp* *add: weak-action-modality-not-tau*)
qed

19.4 Weak formulas

inductive *weak-formula* :: $(\text{'idx}, \text{'pred}::\text{fs}, \text{'act}::\text{bn}) \text{ formula} \Rightarrow \text{bool}$

where
 $wf\text{-}Conj: finite (supp\ xset) \implies (\bigwedge x. x \in set\text{-}bset\ xset \implies weak\text{-}formula\ x)$
 $\implies weak\text{-}formula (Conj\ xset)$
| $wf\text{-}Not: weak\text{-}formula\ x \implies weak\text{-}formula (Not\ x)$
| $wf\text{-}Act: weak\text{-}formula\ x \implies weak\text{-}formula (\langle\langle\alpha\rangle\rangle x)$
| $wf\text{-}Pred: weak\text{-}formula\ x \implies weak\text{-}formula (\langle\langle\tau\rangle\rangle (Conj (binsert (Pred\ \varphi) (bsingleton\ x))))$

lemma *finite-suppl-wf-Pred* [simp]: $finite (supp (binsert (Pred\ \varphi) (bsingleton\ x)))$
proof –
have $supp (bsingleton\ x) \subseteq supp\ x$
by (simp add: eqvtI supp-fun-app-eqvt)
moreover have eqvt binsert
by (simp add: eqvtI)
ultimately have $supp (binsert (Pred\ \varphi) (bsingleton\ x)) \subseteq supp\ \varphi \cup supp\ x$
using supp-fun-app supp-fun-app-eqvt **by** fastforce
then show ?thesis
by (metis finite-UnI finite-suppl rev-finite-subset)
qed

weak-formula is equivariant.

lemma *weak-formula-eqvt* [eqvt]: $weak\text{-}formula\ x \implies weak\text{-}formula (p \cdot x)$
proof (induct rule: weak-formula.induct)
case (wf-Conj xset) **then show** ?case
by simp (metis (no-types, lifting) imageE permute-finite permute-set-eq-image set-bset-eqvt supp-eqvt weak-formula.wf-Conj)
next
case (wf-Not x) **then show** ?case
by (simp add: weak-formula.wf-Not)
next
case (wf-Act x α) **then show** ?case
by (simp add: weak-formula.wf-Act)
next
case (wf-Pred x φ) **then show** ?case
by (simp add: tau-eqvt weak-action-modality-eqvt weak-formula.wf-Pred)
qed

end

end

theory *Weak-Validity*

imports

Weak-Formula

Validity

begin

20 Weak Validity

Weak formulas are a subset of (strong) formulas, and the definition of validity is simply taken from the latter. Here we prove some useful lemmas about the validity of weak modalities. These are similar to corresponding lemmas about the validity of the (strong) action modality.

context *nominal-ts*
begin

lemma *valid-Disj [simp]*:
assumes *finite (supp xset)*
shows $P \models \text{Disj } xset \longleftrightarrow (\exists x \in \text{set-bset } xset. P \models x)$
using *assms* **by** (*simp add: Disj-def map-bset.rep-eq*)

end

context *indexed-weak-nominal-ts*
begin

lemma *valid-weak-tau-modality-iff-tau-steps*:
 $P \models \text{weak-tau-modality } x \longleftrightarrow (\exists n. P \models \text{tau-steps } x \ n)$
unfolding *weak-tau-modality-def* **by** (*auto simp add: map-bset.rep-eq*)

lemma *tau-steps-iff-tau-transition*:
 $(\exists n. P \models \text{tau-steps } x \ n) \longleftrightarrow (\exists P'. P \Rightarrow P' \wedge P' \models x)$

proof

assume $\exists n. P \models \text{tau-steps } x \ n$
then obtain n **where** $P \models \text{tau-steps } x \ n$
by *meson*
then show $\exists P'. P \Rightarrow P' \wedge P' \models x$
proof (*induct n arbitrary: P*)
case 0
then show *?case*
by *simp (metis tau-refl)*
next
case (*Suc n*)
then obtain P' **where** $P \rightarrow \langle \tau, P' \rangle$ **and** $P' \models \text{tau-steps } x \ n$
by (*auto simp add: valid-Act-fresh[OF bn-tau-fresh]*)
with *Suc.hyps* **show** *?case*
using *tau-step* **by** *blast*

qed

next

assume $\exists P'. P \Rightarrow P' \wedge P' \models x$
then obtain P' **where** $P \Rightarrow P'$ **and** $P' \models x$
by *meson*
then show $\exists n. P \models \text{tau-steps } x \ n$
proof (*induct*)
case (*tau-refl P*) **then have** $P \models \text{tau-steps } x \ 0$

```

    by simp
  then show ?case
    by meson
next
case (tau-step P P' P'')
then obtain n where P' ⊨ tau-steps x n
  by meson
with ⟨P → ⟨τ, P'⟩⟩ have P ⊨ tau-steps x (Suc n)
  by (auto simp add: valid-Act-fresh[OF bn-tau-fresh])
then show ?case
  by meson
qed
qed

```

lemma *valid-weak-tau-modality*:

$P \models \text{weak-tau-modality } x \longleftrightarrow (\exists P'. P \Rightarrow P' \wedge P' \models x)$

by (*metis valid-weak-tau-modality-iff-tau-steps tau-steps-iff-tau-transition*)

lemma *valid-weak-action-modality*:

$P \models (\langle\langle\alpha\rangle\rangle x) \longleftrightarrow (\exists \alpha' x' P'. \text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \Rightarrow \langle\alpha'\rangle P' \wedge P' \models x')$
(is ?l \longleftrightarrow ?r)

proof (*cases $\alpha = \tau$*)

case *True* **show** ?thesis

proof

assume ?l

with $\langle\alpha = \tau\rangle$ **obtain** P' **where** $\text{trans}: P \Rightarrow P'$ **and** $\text{valid}: P' \models x$

by (*metis valid-weak-tau-modality weak-action-modality-tau*)

from trans **have** $P \Rightarrow \langle\tau\rangle P'$

by *simp*

with $\langle\alpha = \tau\rangle$ **and** valid **show** ?r

by *blast*

next

assume ?r

then obtain $\alpha' x' P'$ **where** $\text{eq}: \text{Act } \alpha x = \text{Act } \alpha' x'$ **and** $\text{trans}: P \Rightarrow \langle\alpha'\rangle$
 P' **and** $\text{valid}: P' \models x'$

by *blast*

from eq **have** $\alpha' = \tau \wedge x' = x$

using $\langle\alpha = \tau\rangle$ **by** *simp*

with trans **and** valid **have** $P \Rightarrow P'$ **and** $P' \models x$

by *simp+*

with $\langle\alpha = \tau\rangle$ **show** ?l

by (*metis valid-weak-tau-modality weak-action-modality-tau*)

qed

next

case *False* **show** ?thesis

proof

assume ?l

with $\langle\alpha \neq \tau\rangle$ **obtain** Q **where** $\text{trans}: P \Rightarrow Q$ **and** $\text{valid}: Q \models \text{Act } \alpha$
(weak-tau-modality x)

by (*metis valid-weak-tau-modality weak-action-modality-not-tau*)
 from valid obtain $\alpha' x' Q'$ where $eq: Act \alpha (weak-tau-modality x) = Act$
 $\alpha' x'$ and $trans': Q \rightarrow \langle \alpha', Q' \rangle$ and $valid': Q' \models x'$
 by (*metis valid-Act*)

from eq obtain p where $p-\alpha: \alpha' = p \cdot \alpha$ and $p-x: x' = p \cdot weak-tau-modality$
 x

by (*metis Act-eq-iff-perm*)
 with eq have $Act \alpha x = Act \alpha' (p \cdot x)$
 using *Act-weak-tau-modality-eq-iff* by *simp*

moreover from $valid'$ and $p-x$ have $Q' \models weak-tau-modality (p \cdot x)$
 by *simp*
 then obtain P' where $trans'': Q' \Rightarrow P'$ and $valid'': P' \models p \cdot x$
 by (*metis valid-weak-tau-modality*)
 from $trans$ and $trans'$ and $trans''$ have $P \Rightarrow \langle \alpha' \rangle P'$
 by (*metis observable-transitionI weak-transition-stepI*)
 ultimately show *?r*
 using $valid''$ by *blast*

next
 assume *?r*
 then obtain $\alpha' x' P'$ where $eq: Act \alpha x = Act \alpha' x'$ and $trans: P \Rightarrow \langle \alpha' \rangle$
 P' and $valid: P' \models x'$
 by *blast*
 with $\langle \alpha \neq \tau \rangle$ have $\alpha': \alpha' \neq \tau$
 using eq by (*metis Act-tau-eq-iff*)
 with $trans$ obtain $Q Q'$ where $trans': P \Rightarrow Q$ and $trans'': Q \rightarrow \langle \alpha', Q' \rangle$
 and $trans''': Q' \Rightarrow P'$
 by (*meson observable-transition-def weak-transition-def*)
 from $trans'''$ and $valid$ have $Q' \models weak-tau-modality x'$
 by (*metis valid-weak-tau-modality*)
 with $trans''$ have $Q \models Act \alpha' (weak-tau-modality x')$
 by (*metis valid-Act*)
 with $trans'$ and α' have $P \models \langle \langle \alpha' \rangle \rangle x'$
 by (*metis valid-weak-tau-modality weak-action-modality-not-tau*)
 moreover from eq have $(\langle \langle \alpha \rangle \rangle x) = (\langle \langle \alpha' \rangle \rangle x')$
 by (*metis weak-action-modality-eq*)
 ultimately show *?l*
 by *simp*

qed
 qed

The binding names in the alpha-variant that witnesses validity may be chosen fresh for any finitely supported context.

lemma *valid-weak-action-modality-strong*:
 assumes *finite (supp X)*
 shows $P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists \alpha' x' P'. Act \alpha x = Act \alpha' x' \wedge P \Rightarrow \langle \alpha' \rangle P' \wedge$
 $P' \models x' \wedge bn \alpha' \#* X)$
proof

assume $P \models \langle\langle\alpha\rangle\rangle x$
then obtain $\alpha' x' P'$ **where** $eq: Act \alpha x = Act \alpha' x'$ **and** $trans: P \Rightarrow \langle\alpha\rangle P'$
and $valid: P' \models x'$
by (*metis valid-weak-action-modality*)
show $\exists \alpha' x' P'. Act \alpha x = Act \alpha' x' \wedge P \Rightarrow \langle\alpha\rangle P' \wedge P' \models x' \wedge bn \alpha' \#* X$
proof (*cases* $\alpha' = \tau$)
case *True*
then show *?thesis*
using *eq and trans and valid and bn-tau-fresh* **by** *blast*
next
case *False*
with *trans* **obtain** $Q Q'$ **where** $trans': P \Rightarrow Q$ **and** $trans'': Q \rightarrow \langle\alpha', Q'\rangle$
and $trans''': Q' \Rightarrow P'$
by (*metis weak-transition-def observable-transition-def*)
have *finite* ($bn \alpha'$)
by (*fact bn-finite*)
moreover note (*finite* ($supp X$))
moreover have *finite* ($supp (Act \alpha' x', \langle\alpha', Q'\rangle)$)
by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)
moreover have $bn \alpha' \#* (Act \alpha' x', \langle\alpha', Q'\rangle)$
by (*auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
ultimately obtain p **where** $fresh-X: (p \cdot bn \alpha') \#* X$ **and** $supp (Act \alpha' x', \langle\alpha', Q'\rangle) \#* p$
by (*metis at-set-avoiding2*)
then have $supp (Act \alpha' x') \#* p$ **and** $supp \langle\alpha', Q'\rangle \#* p$
by (*metis fresh-star-Un supp-Pair*)
then have $1: Act (p \cdot \alpha') (p \cdot x') = Act \alpha' x'$ **and** $2: \langle p \cdot \alpha', p \cdot Q' \rangle = \langle \alpha', Q' \rangle$
by (*metis Act-eqvt supp-perm-eq, metis abs-residual-pair-eqvt supp-perm-eq*)
from *trans'* **and** *trans''* **and** *trans'''* **have** $P \Rightarrow \langle p \cdot \alpha' \rangle (p \cdot P')$
using 2 **by** (*metis observable-transitionI tau-transition-eqvt weak-transition-stepI*)
then show *?thesis*
using *eq and 1 and valid and fresh-X* **by** (*metis bn-eqvt valid-eqvt*)
qed
next
assume $\exists \alpha' x' P'. Act \alpha x = Act \alpha' x' \wedge P \Rightarrow \langle\alpha\rangle P' \wedge P' \models x' \wedge bn \alpha' \#* X$
then show $P \models \langle\langle\alpha\rangle\rangle x$
by (*metis valid-weak-action-modality*)
qed

lemma *valid-weak-action-modality-fresh*:
assumes $bn \alpha \#* P$
shows $P \models \langle\langle\alpha\rangle\rangle x \iff (\exists P'. P \Rightarrow \langle\alpha\rangle P' \wedge P' \models x)$
proof
assume $P \models \langle\langle\alpha\rangle\rangle x$

moreover have *finite* ($supp P$)
by (*fact finite-supp*)

ultimately obtain $\alpha' x' P'$ **where**
eq: $Act\ \alpha\ x = Act\ \alpha'\ x'$ **and** *trans*: $P \Rightarrow \langle \alpha' \rangle P'$ **and** *valid*: $P' \models x'$ **and**
fresh: $bn\ \alpha' \#^* P$
by (*metis valid-weak-action-modality-strong*)

from *eq* **obtain** p **where** $p\text{-}\alpha$: $\alpha' = p \cdot \alpha$ **and** $p\text{-}x$: $x' = p \cdot x$ **and** *supp-p*:
 $supp\ p \subseteq bn\ \alpha \cup p \cdot bn\ \alpha$
by (*metis Act-eg-iff-perm-renaming*)

from *assms* **and** *fresh* **have** $(bn\ \alpha \cup p \cdot bn\ \alpha) \#^* P$
using $p\text{-}\alpha$ **by** (*metis bn-eqt fresh-star-Un*)
then **have** $supp\ p \#^* P$
using $p\text{-}p$ **by** (*metis fresh-star-def subset-eq*)
then **have** $p\text{-}P$: $-p \cdot P = P$
by (*metis perm-supp-eq supp-minus-perm*)

from *trans* **have** $P \Rightarrow \langle \alpha \rangle (-p \cdot P')$
using $p\text{-}P$ $p\text{-}\alpha$ **by** (*metis permute-minus-cancel(1) weak-transition-eqt*)
moreover **from** *valid* **have** $-p \cdot P' \models x$
using $p\text{-}x$ **by** (*metis permute-minus-cancel(1) valid-eqt*)
ultimately show $\exists P'. P \Rightarrow \langle \alpha \rangle P' \wedge P' \models x$
by *meson*

next
assume $\exists P'. P \Rightarrow \langle \alpha \rangle P' \wedge P' \models x$ **then show** $P \models \langle \langle \alpha \rangle \rangle x$
by (*metis valid-weak-action-modality*)

qed

end

end

theory *Weak-Logical-Equivalence*

imports

Weak-Formula

Weak-Validity

begin

21 Weak Logical Equivalence

context *indexed-weak-nominal-ts*

begin

Two states are weakly logically equivalent if they validate the same weak formulas.

definition *weakly-logically-equivalent* $:: 'state \Rightarrow 'state \Rightarrow bool$ **where**
weakly-logically-equivalent $P\ Q \equiv (\forall x::('idx, 'pred, 'act)\ formula.\ weak\ formula\ x \longrightarrow P \models x \longleftrightarrow Q \models x)$

notation *weakly-logically-equivalent* (**infix** \equiv 50)

```

lemma logically-equivalent-eqt:
  assumes  $P \equiv Q$  shows  $p \cdot P \equiv p \cdot Q$ 
unfolding weakly-logically-equivalent-def proof (clarify)
  fix  $x :: ('idx, 'pred, 'act)$  formula
  assume weak-formula  $x$ 
  then have weak-formula  $(-p \cdot x)$ 
    by (simp add: weak-formula-eqt)
  then show  $p \cdot P \models x \longleftrightarrow p \cdot Q \models x$ 
    using assms by (metis (no-types, lifting) weakly-logically-equivalent-def
permute-minus-cancel(2) valid-eqt)
  qed

end

end
theory Weak-Bisimilarity-Implies-Equivalence
imports
  Weak-Logical-Equivalence
begin

```

22 Weak Bisimilarity Implies Weak Logical Equivalence

```

context indexed-weak-nominal-ts
begin

```

```

lemma weak-bisimilarity-implies-weak-equivalence-Act:
  assumes  $\bigwedge P Q. P \approx Q \implies P \models x \longleftrightarrow Q \models x$ 
  and  $P \approx Q$ 
  — not needed: and weak-formula  $x$ 
  and  $P \models \langle\langle\alpha\rangle\rangle x$ 
  shows  $Q \models \langle\langle\alpha\rangle\rangle x$ 
proof —
  have finite (supp  $Q$ )
    by (fact finite-supp)
  with  $\langle P \models \langle\langle\alpha\rangle\rangle x \rangle$  obtain  $\alpha' x' P'$  where eq:  $Act \alpha x = Act \alpha' x'$  and trans:
 $P \Rightarrow \langle\alpha'\rangle P'$  and valid:  $P' \models x'$  and fresh:  $bn \alpha' \#* Q$ 
    by (metis valid-weak-action-modality-strong)

  from  $\langle P \approx Q \rangle$  and fresh and trans obtain  $Q'$  where trans':  $Q \Rightarrow \langle\alpha'\rangle Q'$ 
and bisim':  $P' \approx Q'$ 
    by (metis weakly-bisimilar-weak-simulation-step)

from eq obtain  $p$  where px:  $x' = p \cdot x$ 
    by (metis Act-eq-iff-perm)

with valid have  $-p \cdot P' \models x$ 
    by (metis permute-minus-cancel(1) valid-eqt)

```

moreover from bisim' have $(-p \cdot P') \approx (-p \cdot Q')$
by (*metis weakly-bisimilar-eqvt*)
ultimately have $-p \cdot Q' \models x$
using $\langle \bigwedge P Q. P \approx Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **by** *metis*
with px have $Q' \models x'$
by (*metis permute-minus-cancel(1) valid-eqvt*)

with eq and trans' show $Q \models \langle \langle \alpha \rangle \rangle x$
unfolding *valid-weak-action-modality* **by** *metis*

qed

lemma *weak-bisimilarity-implies-weak-equivalence-Pred:*

assumes $\bigwedge P Q. P \approx Q \implies P \models x \longleftrightarrow Q \models x$
and $P \approx Q$

— not needed: and *weak-formula x*

and $P \models \langle \langle \tau \rangle \rangle (\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x)))$

shows $Q \models \langle \langle \tau \rangle \rangle (\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x)))$

proof —

let $?c = \text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x))$

from $\langle P \models \langle \langle \tau \rangle \rangle ?c \rangle$ **obtain** P' **where** $\text{trans}: P \Rightarrow P'$ **and** $\text{valid}: P' \models ?c$
using *valid-weak-action-modality* **by** *auto*

have $bn \tau \#* Q$

by (*simp add: fresh-star-def*)

with $\langle P \approx Q \rangle$ **and** trans **obtain** Q' **where** $\text{trans}': Q \Rightarrow Q'$ **and** $\text{bisim}': P' \approx Q'$

by (*metis weakly-bisimilar-weak-simulation-step weak-transition-tau-iff*)

from *valid* **have** $*: P' \vdash \varphi$ **and** $** : P' \models x$

by (*simp add: binsert.rep-eq*) $+$

from bisim' **and** $*$ **obtain** Q'' **where** $\text{trans}'': Q' \Rightarrow Q''$ **and** $\text{bisim}'': P' \approx Q''$
and $*** : Q'' \vdash \varphi$

by (*metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation*)

from bisim'' **and** $**$ **have** $Q'' \models x$

using $\langle \bigwedge P Q. P \approx Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **by** *metis*

with $***$ **have** $Q'' \models ?c$

by (*simp add: binsert.rep-eq*)

moreover from trans' **and** trans'' **have** $Q \Rightarrow \langle \tau \rangle Q''$

by (*metis tau-transition-trans weak-transition-tau-iff*)

ultimately show $Q \models \langle \langle \tau \rangle \rangle ?c$

unfolding *valid-weak-action-modality* **by** *metis*

qed

theorem *weak-bisimilarity-implies-weak-equivalence:* **assumes** $P \approx Q$ **shows** P

```

≡· Q
proof -
{
  fix x :: ('idx, 'pred, 'act) formula
  assume weak-formula x
  then have  $\bigwedge P Q. P \approx \cdot Q \implies P \models x \longleftrightarrow Q \models x$ 
  proof (induct rule: weak-formula.induct)
  case (wf-Conj xset) then show ?case
    by simp
  next
  case (wf-Not x) then show ?case
    by simp
  next
  case (wf-Act x  $\alpha$ ) then show ?case
  by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Act
sympE)
  next
  case (wf-Pred x  $\varphi$ ) then show ?case
  by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Pred
sympE)
  qed
}
with assms show ?thesis
unfolding weakly-logically-equivalent-def by simp
qed

end

end
theory Weak-Equivalence-Implies-Bisimilarity
imports
  Weak-Logical-Equivalence
begin

```

23 Weak Logical Equivalence Implies Weak Bisimilarity

```

context indexed-weak-nominal-ts
begin

```

```

definition distinguishing-formula :: ('idx, 'pred, 'act) formula  $\Rightarrow$  'state  $\Rightarrow$  'state
 $\Rightarrow$  bool

```

```

(- distinguishes - from - [100,100,100] 100)

```

```

where

```

```

x distinguishes P from Q  $\equiv P \models x \wedge \neg Q \models x$ 

```

```

lemma distinguishing-formula-eqvt [eqvt]:

```

```

assumes x distinguishes P from Q shows (p · x) distinguishes (p · P) from

```


($p \cdot Q$)
using *assms unfolding distinguishing-formula-def*
by (*metis permute-minus-cancel(2) valid-eqvt*)

lemma *weakly-equivalent-iff-not-distinguished*: $(P \equiv Q) \longleftrightarrow \neg(\exists x. \text{weak-formula } x \wedge x \text{ distinguishes } P \text{ from } Q)$
by (*meson distinguishing-formula-def weakly-logically-equivalent-def valid-Not wf-Not*)

There exists a distinguishing weak formula for P and Q whose support is contained in $\text{supp } P$.

lemma *distinguished-bounded-support*:
assumes *weak-formula x and x distinguishes P from Q*
obtains y where *weak-formula y and $\text{supp } y \subseteq \text{supp } P$ and y distinguishes P from Q*

proof –

let $?B = \{p \cdot x \mid p. \text{supp } P \#* p\}$

have $\text{supp } P \text{ supports } ?B$

unfolding *supports-def* **proof** (*clarify*)

fix $a \ b$

assume $a: a \notin \text{supp } P$ **and** $b: b \notin \text{supp } P$

have $(a \rightleftharpoons b) \cdot ?B \subseteq ?B$

proof

fix x'

assume $x' \in (a \rightleftharpoons b) \cdot ?B$

then obtain p **where** $1: x' = (a \rightleftharpoons b) \cdot p \cdot x$ **and** $2: \text{supp } P \#* p$

by (*auto simp add: permute-set-def*)

let $?q = (a \rightleftharpoons b) + p$

from 1 **have** $x' = ?q \cdot x$

by *simp*

moreover from a **and** b **and** 2 **have** $\text{supp } P \#* ?q$

by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3)*)

ultimately show $x' \in ?B$ **by** *blast*

qed

moreover have $?B \subseteq (a \rightleftharpoons b) \cdot ?B$

proof

fix x'

assume $x' \in ?B$

then obtain p **where** $1: x' = p \cdot x$ **and** $2: \text{supp } P \#* p$

by *auto*

let $?q = (a \rightleftharpoons b) + p$

from 1 **have** $x' = (a \rightleftharpoons b) \cdot ?q \cdot x$

by *simp*

moreover from a **and** b **and** 2 **have** $\text{supp } P \#* ?q$

by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3)*)

ultimately show $x' \in (a \rightleftharpoons b) \cdot ?B$

using *mem-permute-iff* **by** *blast*

qed

ultimately show $(a \rightleftharpoons b) \cdot ?B = ?B$..

qed

```

then have supp-B-subset-supp-P: supp ?B  $\subseteq$  supp P
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have finite-supp-B: finite (supp ?B)
  using finite-supp rev-finite-subset by blast
have ?B  $\subseteq$  ( $\lambda p. p \cdot x$ ) ‘ UNIV
  by auto
then have |?B|  $\leq o$  |UNIV :: perm set|
  by (rule surj-imp-ordLeq)
also have |UNIV :: perm set|  $< o$  |UNIV :: ‘idx set|
  by (metis card-idx-perm)
also have |UNIV :: ‘idx set|  $\leq o$  natLeq + c |UNIV :: ‘idx set|
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-B: |?B|  $< o$  natLeq + c |UNIV :: ‘idx set| .
let ?y = Conj (Abs-bset ?B) :: (‘idx, ‘pred, ‘act) formula
have weak-formula ?y
proof
  show finite (supp (Abs-bset ?B :: - set[‘idx]))
    using finite-supp-B card-B by simp
next
  fix x' assume x' ∈ set-bset (Abs-bset ?B :: - set[‘idx])
  with card-B obtain p where x' = p · x
    using Abs-bset-inverse mem-Collect-eq by auto
  then show weak-formula x'
    using (‘weak-formula x) by (metis weak-formula-eqvt)
qed
  moreover from finite-supp-B and card-B and supp-B-subset-supp-P have
supp ?y  $\subseteq$  supp P
  by simp
  moreover have ?y distinguishes P from Q
  unfolding distinguishing-formula-def proof
    from assms show P  $\models$  ?y
      by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
supp-perm-eq valid-eqvt)
  next
    from assms show  $\neg$  Q  $\models$  ?y
      by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
  qed
  ultimately show ?thesis ..
qed

```

lemma *weak-equivalence-is-weak-bisimulation*: *is-weak-bisimulation* *weakly-logically-equivalent*

```

proof –
  have symp weakly-logically-equivalent
    by (metis weakly-logically-equivalent-def sympI)
  moreover — weak static implication
  {
    fix P Q  $\varphi$  assume P  $\equiv$ · Q and P  $\vdash$   $\varphi$ 
    then have  $\exists Q'. Q \Rightarrow Q' \wedge P \equiv \cdot Q' \wedge Q' \vdash \varphi$ 
  }

```

proof –
 {
 let $?Q' = \{Q'. Q \Rightarrow Q' \wedge Q' \vdash \varphi\}$
 assume $\forall Q' \in ?Q'. \neg P \equiv Q'$
 then have $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. weak-formula } x$
 $\wedge x \text{ distinguishes } P \text{ from } Q'$
 by (*metis weakly-equivalent-iff-not-distinguished*)
 then have $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. weak-formula } x$
 $\wedge \text{supp } x \subseteq \text{supp } P \wedge x \text{ distinguishes } P \text{ from } Q'$
 by (*metis distinguished-bounded-support*)
 then obtain $f :: 'state \Rightarrow ('idx, 'pred, 'act) \text{ formula where}$
 $*: \forall Q' \in ?Q'. \text{weak-formula } (f Q') \wedge \text{supp } (f Q') \subseteq \text{supp } P \wedge (f Q')$
 $\text{distinguishes } P \text{ from } Q'$
 by *metis*
 have $\text{supp } P \text{ supports } (f ' ?Q')$
 unfolding *supports-def* **proof** (*clarify*)
 fix $a \ b$
 assume $a: a \notin \text{supp } P \text{ and } b: b \notin \text{supp } P$
 have $(a \equiv b) \cdot (f ' ?Q') \subseteq f ' ?Q'$
 proof
 fix x
 assume $x \in (a \equiv b) \cdot (f ' ?Q')$
 then have $(a \equiv b) \cdot x \in f ' ?Q'$
 by (*metis (full-types) mem-permute-iff permute-swap-cancel*)
 then obtain Q' **where** $1: (a \equiv b) \cdot x = f Q'$ **and** $2: Q \Rightarrow Q' \wedge$
 $Q' \vdash \varphi$
 by *auto*
 with $*$ **and** a **and** b **have** $a \notin \text{supp } (f Q') \text{ and } b \notin \text{supp } (f Q')$
 by *auto*
 with 1 **have** $x = f Q'$
 by (*metis permute-swap-cancel supp-supports supports-def*)
 with 2 **show** $x \in f ' ?Q'$
 by *simp*
 qed
 moreover have $f ' ?Q' \subseteq (a \equiv b) \cdot (f ' ?Q')$
 proof
 fix x
 assume $x \in f ' ?Q'$
 then obtain Q' **where** $1: x = f Q'$ **and** $2: Q \Rightarrow Q' \wedge Q' \vdash \varphi$
 by *auto*
 with $*$ **and** a **and** b **have** $a \notin \text{supp } (f Q') \text{ and } b \notin \text{supp } (f Q')$
 by *auto*
 with 1 **have** $x = (a \equiv b) \cdot f Q'$
 by (*metis fresh-perm fresh-star-def supp-perm-eq swap-atom*)
 with 2 **show** $x \in (a \equiv b) \cdot (f ' ?Q')$
 using *mem-permute-iff* **by** *blast*
 qed
 ultimately show $(a \equiv b) \cdot (f ' ?Q') = f ' ?Q' ..$
 qed

```

then have supp-image-subset-supp-P:  $\text{supp } (f \text{ ' ?}Q') \subseteq \text{supp } P$ 
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have finite-supp-image:  $\text{finite } (\text{supp } (f \text{ ' ?}Q'))$ 
  using finite-supp rev-finite-subset by blast
have  $|f \text{ ' ?}Q'| \leq_o |UNIV :: \text{'state set}|$ 
  using card-of-UNIV card-of-image ordLeq-transitive by blast
also have  $|UNIV :: \text{'state set}| <_o |UNIV :: \text{'idx set}|$ 
  by (metis card-idx-state)
also have  $|UNIV :: \text{'idx set}| \leq_o \text{natLeq} + c |UNIV :: \text{'idx set}|$ 
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-image:  $|f \text{ ' ?}Q'| <_o \text{natLeq} + c |UNIV :: \text{'idx set}|$  .

let ?y = Conj (Abs-bset (f ' ?Q')) :: ('idx, 'pred, 'act) formula
have weak-formula ?y
  proof (standard+)
    show  $\text{finite } (\text{supp } (\text{Abs-bset } (f \text{ ' ?}Q') :: - \text{set}[\text{'idx}]))$ 
      using finite-supp-image card-image by simp
    next
      fix x assume  $x \in \text{set-bset } (\text{Abs-bset } (f \text{ ' ?}Q') :: - \text{set}[\text{'idx}])$ 
      with card-image obtain Q' where  $Q' \in ?Q'$  and  $x = f Q'$ 
        using Abs-bset-inverse imageE set-bset set-bset-to-set-bset by auto
      then show weak-formula x
        using * by metis
  qed

let ?z =  $\langle\langle\tau\rangle\rangle(\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } ?y)))$ 
have weak-formula ?z
  by standard (fact <weak-formula ?y>)
moreover have  $P \models ?z$ 
  proof -
    have  $P \Rightarrow \langle\tau\rangle P$ 
      by simp
    moreover
      {
        fix Q'
        assume  $Q \Rightarrow Q' \wedge Q' \vdash \varphi$ 
        with * have  $P \models f Q'$ 
          by (metis distinguishing-formula-def mem-Collect-eq)
      }
    with  $\langle P \vdash \varphi \rangle$  have  $P \models \text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } ?y))$ 
      by (simp add: binsert.rep-eq finite-supp-image card-image)
    ultimately show thesis
      using valid-weak-action-modality by blast
  qed
moreover have  $\neg Q \models ?z$ 
  proof
    assume  $Q \models ?z$ 
    then obtain Q' where  $1: Q \Rightarrow Q'$  and  $Q' \models \text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } ?y))$ 

```

```

    using valid-weak-action-modality by auto
  then have 2:  $Q' \vdash \varphi$  and 3:  $Q' \models ?y$ 
    by (simp add: binsert.rep-eq finite-supp-image card-image)+
  from 3 have  $\bigwedge Q''. Q \Rightarrow Q'' \wedge Q'' \vdash \varphi \longrightarrow Q' \models f Q''$ 
    by (simp add: finite-supp-image card-image)
  with 1 and 2 and * show False
    using distinguishing-formula-def by blast
  qed
  ultimately have False
    by (metis  $\langle P \equiv \cdot Q \rangle$  weakly-logically-equivalent-def)
}
then show ?thesis
  by blast
qed
}
moreover — weak simulation
{
  fix  $P Q \alpha P'$  assume  $P \equiv \cdot Q$  and  $bn \alpha \#* Q$  and  $P \rightarrow \langle \alpha, P \rangle$ 
  then have  $\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge P' \equiv \cdot Q'$ 
    proof —
      {
        let  $?Q' = \{Q'. Q \Rightarrow \langle \alpha \rangle Q'\}$ 
        assume  $\forall Q' \in ?Q'. \neg P' \equiv \cdot Q'$ 
        then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } \text{weak-formula } x$ 
           $\wedge x \text{ distinguishes } P' \text{ from } Q'$ 
          by (metis weakly-equivalent-iff-not-distinguished)
        then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } \text{weak-formula } x$ 
           $\wedge \text{supp } x \subseteq \text{supp } P' \wedge x \text{ distinguishes } P' \text{ from } Q'$ 
          by (metis distinguished-bounded-support)
        then obtain  $f :: 'state \Rightarrow ('idx, 'pred, 'act) \text{ formula}$  where
          * :  $\forall Q' \in ?Q'. \text{weak-formula } (f Q') \wedge \text{supp } (f Q') \subseteq \text{supp } P' \wedge (f Q')$ 
          distinguishes } P' \text{ from } Q'
          by metis
        have  $\text{supp } P' \text{ supports } (f \text{ ' } ?Q')$ 
          unfolding supports-def proof (clarify)
          fix  $a b$ 
          assume  $a: a \notin \text{supp } P'$  and  $b: b \notin \text{supp } P'$ 
          have  $(a \Rightarrow b) \cdot (f \text{ ' } ?Q') \subseteq f \text{ ' } ?Q'$ 
          proof
            fix  $x$ 
            assume  $x \in (a \Rightarrow b) \cdot (f \text{ ' } ?Q')$ 
            then obtain  $Q'$  where 1:  $x = (a \Rightarrow b) \cdot f Q'$  and 2:  $Q \Rightarrow \langle \alpha \rangle Q'$ 
            by auto (metis (no-types, lifting) imageE image-eqv mem-Collect-eq
              permute-set-eq-image)
            with * and  $a$  and  $b$  have  $a \notin \text{supp } (f Q')$  and  $b \notin \text{supp } (f Q')$ 
              by auto
            with 1 have  $x = f Q'$ 
              by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
            with 2 show  $x \in f \text{ ' } ?Q'$ 

```

```

    by simp
qed
moreover have  $f' \ ?Q' \subseteq (a \Rightarrow b) \cdot (f' \ ?Q')$ 
proof
  fix x
  assume  $x \in f' \ ?Q'$ 
  then obtain  $Q'$  where  $1: x = f \ Q'$  and  $2: Q \Rightarrow \langle \alpha \rangle \ Q'$ 
    by auto
  with * and a and b have  $a \notin \text{supp} (f \ Q')$  and  $b \notin \text{supp} (f \ Q')$ 
    by auto
  with 1 have  $x = (a \Rightarrow b) \cdot f \ Q'$ 
    by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
  with 2 show  $x \in (a \Rightarrow b) \cdot (f' \ ?Q')$ 
    using mem-permute-iff by blast
qed
ultimately show  $(a \Rightarrow b) \cdot (f' \ ?Q') = f' \ ?Q' ..$ 
qed
then have supp-image-subset-supp-P':  $\text{supp} (f' \ ?Q') \subseteq \text{supp} \ P'$ 
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have finite-supp-image: finite ( $\text{supp} (f' \ ?Q')$ )
  using finite-supp rev-finite-subset by blast
have  $|f' \ ?Q'| \leq o \ |UNIV :: 'state \ set|$ 
  by (metis card-of-UNIV card-of-image ordLeq-transitive)
also have  $|UNIV :: 'state \ set| < o \ |UNIV :: 'idx \ set|$ 
  by (metis card-idx-state)
also have  $|UNIV :: 'idx \ set| \leq o \ \text{natLeq} + c \ |UNIV :: 'idx \ set|$ 
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-image:  $|f' \ ?Q'| < o \ \text{natLeq} + c \ |UNIV :: 'idx \ set| .$ 

let  $?y = \text{Conj} (\text{Abs-bset} (f' \ ?Q')) :: ('idx, 'pred, 'act) \ \text{formula}$ 
have weak-formula ( $\langle \langle \alpha \rangle \rangle ?y$ )
proof (standard+)
  show finite ( $\text{supp} (\text{Abs-bset} (f' \ ?Q') :: - \ \text{set}['idx])$ )
    using finite-supp-image card-image by simp
next
  fix x assume  $x \in \text{set-bset} (\text{Abs-bset} (f' \ ?Q') :: - \ \text{set}['idx])$ 
  with card-image obtain  $Q'$  where  $Q' \in \ ?Q'$  and  $x = f \ Q'$ 
    using Abs-bset-inverse imageE set-bset set-bset-to-set-bset by auto
  then show weak-formula x
    using * by metis
qed
moreover have  $P \models \langle \langle \alpha \rangle \rangle ?y$ 
unfolding valid-weak-action-modality proof (standard+)
  from  $\langle P \rightarrow \langle \alpha, P' \rangle \rangle$  show  $P \Rightarrow \langle \alpha \rangle \ P'$ 
    by simp
next
{
  fix  $Q'$ 
  assume  $Q \Rightarrow \langle \alpha \rangle \ Q'$ 

```

```

    with * have  $P' \models f Q'$ 
      by (metis distinguishing-formula-def mem-Collect-eq)
    }
    then show  $P' \models ?y$ 
      by (simp add: finite-supp-image card-image)
    qed
  moreover have  $\neg Q \models \langle\langle\alpha\rangle\rangle ?y$ 
    proof
      assume  $Q \models \langle\langle\alpha\rangle\rangle ?y$ 
      then obtain  $Q'$  where 1:  $Q \Rightarrow \langle\alpha\rangle Q'$  and 2:  $Q' \models ?y$ 
        using  $\langle bn \ \alpha \ \#\ * \ Q \rangle$  by (metis valid-weak-action-modality-fresh)
      from 2 have  $\bigwedge Q''. Q \Rightarrow \langle\alpha\rangle Q'' \longrightarrow Q' \models f Q''$ 
        by (simp add: finite-supp-image card-image)
      with 1 and * show False
        using distinguishing-formula-def by blast
      qed
    ultimately have False
      by (metis  $\langle P \equiv \cdot Q \rangle$  weakly-logically-equivalent-def)
    }
  then show ?thesis by auto
  qed
}
ultimately show ?thesis
  unfolding is-weak-bisimulation-def by metis
qed

theorem weak-equivalence-implies-weak-bisimilarity: assumes  $P \equiv \cdot Q$  shows  $P \approx \cdot Q$ 
using assms by (metis weakly-bisimilar-def weak-equivalence-is-weak-bisimulation)

end

end

```

References

- [1] J. Parrow, J. Borgström, L. Eriksson, R. Gutkovas, and T. Weber. Modal logics for nominal transition systems. In L. Aceto and D. de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPICs*, pages 198–211. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.