

Modal Logics for Nominal Transition Systems

Tjark Weber et al.

December 14, 2021

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	12
4.1	Basic Lemmas	12
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	19
5.3	Constructors for trees modulo α -equivalence	33
5.4	Induction over trees modulo α -equivalence	36
5.5	Hereditarily finitely supported trees	37

5.6	Infinitary formulas	38
5.7	Constructors for infinitary formulas	40
5.8	Induction over infinitary formulas	44
5.9	Strong induction over infinitary formulas	45
6	Validity	46
6.1	Validity for infinitely branching trees	48
6.2	Validity for trees modulo α -equivalence	51
6.3	Validity for infinitary formulas	52
7	(Strong) Logical Equivalence	54
8	Bisimilarity Implies Logical Equivalence	55
9	Logical Equivalence Implies Bisimilarity	56
10	Disjunction	60
11	Expressive Completeness	61
11.1	Distinguishing formulas	61
11.2	Characteristic formulas	65
11.3	Expressive completeness	68
12	Finitely Supported Sets	69
13	Nominal Transition Systems with Effects and F/L-Bisimilarity	70
13.1	Nominal transition systems with effects	70
13.2	L -bisimulations and F/L -bisimilarity	71
14	Infinitary Formulas With Effects	76
14.1	Infinitely branching trees	76
14.2	Trees modulo α -equivalence	78
14.3	Constructors for trees modulo α -equivalence	93
14.4	Induction over trees modulo α -equivalence	96
14.5	Hereditarily finitely supported trees	96
14.6	Infinitary formulas	98
14.7	Constructors for infinitary formulas	100
14.8	F/L -formulas	104
14.9	Induction over infinitary formulas	105
14.10	Strong induction over infinitary formulas	105
15	Validity With Effects	105
15.1	Validity for infinitely branching trees	107
15.2	Validity for trees modulo α -equivalence	110
15.3	Validity for infinitary formulas	111

16 (Strong) Logical Equivalence	114
17 F/L-Bisimilarity Implies Logical Equivalence	114
18 Logical Equivalence Implies F/L-Bisimilarity	116
19 L-Transform	121
19.1 States	121
19.2 Actions and binding names	122
19.3 Satisfaction	123
19.4 Transitions	123
19.5 Translation of F/L -formulas into formulas without effects . .	126
19.6 Bisimilarity in the L -transform	134
20 Nominal Transition Systems and Bisimulations with Unob-	
servable Transitions	139
20.1 Nominal transition systems with unobservable transitions . .	139
20.2 Weak bisimulations	141
21 Weak Formulas	147
21.1 Lemmas about α -equivalence involving τ	147
21.2 Weak action modality	148
21.3 Weak formulas	151
22 Weak Validity	152
23 Weak Logical Equivalence	157
24 Weak Bisimilarity Implies Weak Logical Equivalence	158
25 Weak Logical Equivalence Implies Weak Bisimilarity	160
26 Weak Expressive Completeness	166
26.1 Distinguishing weak formulas	166
26.2 Characteristic weak formulas	171
26.3 Weak expressive completeness	175
27 S-Transform: State Predicates as Actions	177
27.1 Actions and binding names	177
27.2 Satisfaction	178
27.3 Transitions	178
27.4 Strong Bisimilarity in the S -transform	179
27.5 Weak Bisimilarity in the S -transform	182
27.6 Translation of (strong) formulas into formulas without pred-	
icates	186
27.7 Translation of weak formulas into formulas without predicates	193

```

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  $|A| <_o \text{natLeq} + c \mid \text{UNIV} :: 'k \text{ set}$ 
  shows set-bset (Abs-bset  $A :: 'a \text{ set}['k]$ ) =  $A$ 
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 ::  $\text{perm} \Rightarrow 'a \text{ set}['b] \Rightarrow 'a \text{ set}['b]$  is
    permute

```

```

proof –

```

```

  fix  $p$  and  $A :: 'a \text{ set}$ 

```

```

  have  $|p \cdot A| \leq_o |A|$  by (simp add: permute-set-eq-image)

```

```

  also assume  $|A| <_o \text{natLeq} + c \mid \text{UNIV} :: 'b \text{ set}$ 

```

```

  finally show  $|p \cdot A| <_o \text{natLeq} + c \mid \text{UNIV} :: 'b \text{ set}$  .

```

```

qed

```

```

instance

```

```

by standard (transfer, simp)+

```

```

end

```

```

lemma Abs-bset-eqvt [simp]:
  assumes  $|A| <_o \text{natLeq} + c \mid \text{UNIV} :: 'k \text{ set}$ 
  shows  $p \cdot (\text{Abs-bset } A :: 'a::\text{pt} \text{ set}['k]) = \text{Abs-bset } (p \cdot A)$ 
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  $|A| <_o \text{natLeq} + c \mid \text{UNIV} :: 'k \text{ set}$ 

```

```

  shows supp (Abs-bset  $A :: 'a::\text{pt} \text{ set}['k]$ ) = supp  $A$ 

```

```

proof –

```

```

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

```

$\neq A$
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 \circ 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, opaque-lifting) 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 ( $\alpha 1, P 1$ ) ( $\alpha 2, P 2$ )  $\longleftrightarrow$  [bn  $\alpha 1$ ]set. ( $\alpha 1, P 1$ ) = [bn  $\alpha 2$ ]set. ( $\alpha 2, P 2$ )

```

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

```

— The following lemma is not about residuals, but we have no better place for it.

```

lemma set-bounded-supp:
  assumes finite S and ∧x. x∈X ⟹ supp x ⊆ S
  shows supp X ⊆ S
proof —
  have S supports X
    unfolding supports-def proof (clarify)
      fix a b
      assume a: a ∉ S and b: b ∉ S
      {
        fix x
        assume x ∈ X
        then have (a ≡ b) · x = x
          using a b ⟨∧x. x∈X ⟹ supp x ⊆ S⟩ by (meson fresh-def subsetCE)
      }
    qed

```

```

swap-fresh-fresh)
}
then show  $(a \equiv b) \cdot X = X$ 
  by auto (metis (full-types) eqvt-bound mem-permute-iff, metis mem-permute-iff)
qed
then show  $\text{supp } X \subseteq S$ 
  using assms(1) by (fact supp-is-subset)
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 ( $p \cdot R$ )
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** $\sim \cdot$ 100) **where**
 $P \sim \cdot Q \equiv \exists R. \text{is-bisimulation } R \wedge R P Q$

($\sim \cdot$) 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 3 **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  $?bisim = bisimilar OO bisimilar$ 
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  $\forall P Q. ?bisim P Q \longrightarrow$ 
  ( $\forall \alpha P'. bn \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \rightarrow \langle \alpha, Q' \rangle \wedge ?bisim P'$ 
 $Q')$ )
proof ( $clarify$ )
  fix  $P R Q \alpha P'$ 
  assume  $PR: P \sim R$  and  $RQ: R \sim Q$  and  $fresh: bn \alpha \#* Q$  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  $bn \alpha R (\langle \alpha, P' \rangle, Q)$ , 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  $fastforce$ 
then obtain  $pR'$  where  $5: R \rightarrow \langle p \cdot \alpha, pR' \rangle$  and  $6: (p \cdot P') \sim pR'$ 
using  $PR trans 1$  by ( $metis (mono-tags, lifting) bisimilar-is-bisimulation$ 
 $bn-eqvt is-bisimulation-def$ )
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  $7: Q \rightarrow \langle p \cdot \alpha, pQ' \rangle$  and  $8: pR' \sim pQ'$ 
using  $RQ 5$  by ( $metis (full-types) bisimilar-is-bisimulation is-bisimulation-def$ )
from  $7$  have  $Q \rightarrow \langle \alpha, -p \cdot pQ' \rangle$ 
using  $4$  by ( $metis permute-minus-cancel(2) supp-perm-eq transition-eqvt'$ )
moreover from  $6$  and  $8$  have  $?bisim P' (-p \cdot pQ')$ 
by ( $metis (no-types, opaque-lifting) bisimilar-eqvt permute-minus-cancel(2)$ 
 $relcompp.simps$ )

```

```

    ultimately show  $\exists Q'. Q \rightarrow \langle \alpha, Q' \rangle \wedge ?bisim P' Q'$ 
      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  $P \rightarrow \langle \alpha, P' \rangle$ 
  obtains  $Q'$  where  $Q \rightarrow \langle \alpha, Q' \rangle$  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 ⇒ (-,-,-) Tree ⇒ (-,-,-) Tree **where**

p · (tConj tset) = tConj (map-bset (permute p) tset) — neat trick to get around

the fact that tset is not of permutation type yet

| p · (tNot t) = tNot (p · t)

| p · (tPred φ) = tPred (p · φ)

| p · (tAct α t) = tAct (p · α) (p · t)

instance

proof

fix t :: (-,-,-) Tree

show 0 · 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) · t = p · q · 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) ∈ Tree-wf shows (p · t1, p · t2) ∈ 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 ∈ set-bset tset then show (p · t, p · tConj tset) ∈ 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 · t, p · tNot t) ∈ Tree-wf
    by (metis Tree-wf.intros(2) permute-Tree.simps(2))
next
  fix t :: ('a,'b,'c) Tree and α
  show (p · t, p · tAct α t) ∈ Tree-wf
    by (metis Tree-wf.intros(3) permute-Tree.simps(4))
qed

```

lemma *Tree-wf-eqt* [eqvt, simp]: $p \cdot Tree-wf = Tree-wf$

proof

```

  show  $p \cdot Tree-wf \subseteq Tree-wf$ 
    by (auto simp add: permute-set-def) (rule Tree-wf-eqt-aux)
next
  show  $Tree-wf \subseteq p \cdot Tree-wf$ 
    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*]: $\text{supp } (t\text{Not } t) = \text{supp } t$
unfolding *supp-def* **by** *simp*

lemma *supp-tPred* [*simp*]: $\text{supp } (t\text{Pred } \varphi) = \text{supp } \varphi$
unfolding *supp-def* **by** *simp*

lemma *supp-tAct* [*simp*]: $\text{supp } (t\text{Act } \alpha t) = \text{supp } \alpha \cup \text{supp } t$
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*]:
 $p \cdot \text{supp-rel } R x = \text{supp-rel } (p \cdot R) (p \cdot x)$
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* ($t\text{Conj } tset) = \text{supp-rel } \alpha\text{-Tree } (t\text{Conj } tset)$.

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

$\llbracket bs=bs'; x=x'; R (p' \cdot x') y' \longleftrightarrow R' (p' \cdot x') y'; f x' = f' x'; f y' = f' y'; p=p'; cs=cs'; y=y' \rrbracket \implies$
 $\text{alpha-set } (bs, x) R f p (cs, y) \longleftrightarrow \text{alpha-set } (bs', x') R' f' p' (cs', y')$
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 $\text{wf } (\text{inv-image } (\text{hull-rel } O \text{ Tree-wf}) \text{ rep-Tree}_p)$

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

next

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

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

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

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: rep-Treep bp = q · b
  by (metis permute-rep-abs-Treep)
have (p · a, q · a) ∈ hull-rel
  by (metis hull-rel.simps permute-minus-cancel(2) permute-plus)
moreover from 3 have (q · a, q · b) ∈ Tree-wf
  by (rule Tree-wf-eqvt-aux)
ultimately show (ap, bp) ∈ inv-image (hull-rel O Tree-wf) rep-Treep
  using 4 5 by auto
qed
qed

```

```

fun alpha-Tree-termination :: ('a, 'b, 'c) Tree × ('a, 'b, 'c) Tree ⇒ ('a, 'b::pt,
'c::bn) Treep set where
  alpha-Tree-termination (t1, t2) = {abs-Treep t1, abs-Treep t2}

```

Here it comes ...

```

function (sequential)
  alpha-Tree :: ('idx, 'pred::pt, 'act::bn) Tree ⇒ ('idx, 'pred, 'act) Tree ⇒ bool (infix
=α 50) where
  — (=α)
  | alpha-tConj: tConj tset1 =α tConj tset2 ↔ rel-bset alpha-Tree tset1 tset2
  | alpha-tNot: tNot t1 =α tNot t2 ↔ t1 =α t2
  | alpha-tPred: tPred φ1 =α tPred φ2 ↔ φ1 = φ2
  — the action may have binding names
  | alpha-tAct: tAct α1 t1 =α tAct α2 t2 ↔
    (∃ p. (bn α1, t1) ≈set alpha-Tree (supp-rel alpha-Tree) p (bn α2, t2) ∧ (bn α1,
α1) ≈set ((=)) supp p (bn α2, α2))
  | alpha-other: - =α - ↔ False
  — 254 subgoals (!)
by pat-completeness auto
termination
proof
  let ?R = inv-image (max-ext Tree-wfp) alpha-Tree-termination
  show wf ?R
  by (metis max-ext-wf wf-Tree-wfp 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 =α t2
  and ∧tset1 tset2. (∧ a b. a ∈ set-bset tset1 ⇒ b ∈ set-bset tset2 ⇒ a =α b
⇒ P a b) ⇒

```

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

α -equivalence is equivariant.

lemma *alpha-Tree-eqvt-aux*:

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

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

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) eqvt-bound permute-eqvt swap-eqvt)*
 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 =α tConj tset2
    {
      fix x
      assume x ∈ set-bset (p · tset1)
      then obtain x' where 1: x' ∈ set-bset tset1 and 2: x = p · x'
        by (metis imageE permute-bset.rep-eq permute-set-eq-image)
      from 1 obtain y' where 3: y' ∈ set-bset tset2 and 4: x' =α y'
        using * by (metis (mono-tags, lifting) Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
      from 3 have p · y' ∈ set-bset (p · tset2)
        by (metis mem-permute-iff set-bset-eqt)
      moreover from 1 and 2 and 3 and 4 have x =α p · y'
        using alpha-tConj.IH by blast
      ultimately have ∃ y ∈ set-bset (p · tset2). x =α y ..
    }
    moreover
    {
      fix y
      assume y ∈ set-bset (p · tset2)
      then obtain y' where 1: y' ∈ set-bset tset2 and 2: p · y' = y
        by (metis imageE permute-bset.rep-eq permute-set-eq-image)
      from 1 obtain x' where 3: x' ∈ set-bset tset1 and 4: x' =α y'
        using * by (metis (mono-tags, lifting) Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
      from 3 have p · x' ∈ set-bset (p · tset1)
        by (metis mem-permute-iff set-bset-eqt)
      moreover from 1 and 2 and 3 and 4 have p · x' =α y
        using alpha-tConj.IH by blast
      ultimately have ∃ x ∈ set-bset (p · tset1). x =α y ..
    }
    ultimately show p · tConj tset1 =α p · tConj tset2
      by (simp add: rel-bset-def rel-set-def)
  next
    assume *: p · tConj tset1 =α p · 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-eqt)
      then obtain y' where 2: y' ∈ set-bset (p · tset2) and 3: p · x =α y'
        using * by (metis 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  $\exists y \in \text{set-bset } tset2. x =_\alpha y ..$ 
}
moreover
{
  fix y
  assume 1:  $y \in \text{set-bset } tset2$ 
  then have  $p \cdot y \in \text{set-bset } (p \cdot tset2)$ 
    by (metis mem-permute-iff set-bset-eqvt)
  then obtain x' where 2:  $x' \in \text{set-bset } (p \cdot tset1)$  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  $tConj\ tset1 =_\alpha tConj\ tset2$ 
  by (simp add: rel-bset-def rel-set-def)
qed
next
case (alpha-tAct  $\alpha1\ t1\ \alpha2\ t2$ )
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:  $(bn\ \alpha1, t1) \approx_{\text{set}} (=_\alpha)\ (\text{supp-rel } (=_\alpha))\ q\ (bn\ \alpha2, t2)$ 
and 2:  $(bn\ \alpha1, \alpha1) \approx_{\text{set}} (=)\ \text{supp}\ q\ (bn\ \alpha2, \alpha2)$ 
  by auto
  from 1 and t1 and t2 have  $\text{supp-rel } (=_\alpha)\ (p \cdot t1) - bn\ (p \cdot \alpha1) = \text{supp-rel } (=_\alpha)\ (p \cdot t2) - bn\ (p \cdot \alpha2)$ 
  by (metis Diff-eqvt alpha-set bn-eqvt)
  moreover from 1 and t1 have  $(\text{supp-rel } (=_\alpha)\ (p \cdot t1) - bn\ (p \cdot \alpha1)) \#^*\ (p + q - p)$ 
  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  $(bn\ (p \cdot \alpha1), p \cdot t1) \approx_{\text{set}} (=_\alpha)\ (\text{supp-rel } (=_\alpha))\ (p + q - p)$ 
  by (metis Diff-eqvt alpha-set bn-eqvt fresh-star-permute-iff permute-perm-def)
  by (simp add: alpha-set)
  moreover from 2 have  $(bn\ (p \cdot \alpha1), p \cdot \alpha1) \approx_{\text{set}} (=)\ \text{supp}\ (p + q - p)\ (bn\ (p \cdot \alpha2), p \cdot \alpha2)$ 
  by (metis Diff-eqvt alpha-set bn-eqvt fresh-star-permute-iff permute-perm-def)
  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 $p \cdot tAct \alpha 1 t1 =_{\alpha} p \cdot tAct \alpha 2 t2$
by auto
next
assume $p \cdot tAct \alpha 1 t1 =_{\alpha} p \cdot tAct \alpha 2 t2$
then obtain q **where** $1: (bn (p \cdot \alpha 1), p \cdot t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) q$
 $(bn (p \cdot \alpha 2), p \cdot t2)$ **and** $2: (bn (p \cdot \alpha 1), p \cdot \alpha 1) \approx_{set} (=) supp q (bn (p \cdot \alpha 2),$
 $p \cdot \alpha 2)$
by auto
{
from 1 **and** $t1$ **and** $t2$ **have** $supp-rel (=_{\alpha}) t1 - bn \alpha 1 = supp-rel (=_{\alpha}) t2$
 $- bn \alpha 2$
by $(metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff)$
moreover with 1 **and** $t2$ **have** $(supp-rel (=_{\alpha}) t1 - bn \alpha 1) \#* (-p + q +$
 $p)$
by $(auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting)$
 $DiffI bn-eqvt mem-permute-iff permute-minus-cancel(2))$
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** $-p \cdot q \cdot p \cdot bn \alpha 1 = bn \alpha 2$
by $(metis alpha-set bn-eqvt permute-minus-cancel(2))$
ultimately have $(bn \alpha 1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (-p + q + p) (bn$
 $\alpha 2, t2)$
by $(simp add: alpha-set)$
}
moreover
{
from 2 **have** $supp \alpha 1 - bn \alpha 1 = supp \alpha 2 - bn \alpha 2$
by $(metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff$
 $supp-eqvt)$
moreover with 2 **have** $(supp \alpha 1 - bn \alpha 1) \#* (-p + q + p)$
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 $-p \cdot q \cdot p \cdot bn \alpha 1 = bn \alpha 2$
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 $tAct \alpha 1 t1 =_{\alpha} tAct \alpha 2 t2$
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) 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) 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  $(bn \alpha 1, t1) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) p (bn \alpha 2, t2) \wedge$ 
 $(bn \alpha 1, \alpha 1) \approx_{set} (=) supp p (bn \alpha 2, \alpha 2)$ 
      by auto
      then have  $(bn \alpha 2, t2) \approx_{set} (=_{\alpha}) (supp-rel (=_{\alpha})) (-p) (bn \alpha 1, t1) \wedge (bn \alpha 2,$ 
 $\alpha 2) \approx_{set} (=) supp (-p) (bn \alpha 1, \alpha 1)$ 
      using tAct.IH by (metis (mono-tags, lifting) alpha-Tree-egvt 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) 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  $z = tConj tset-z$ 
        have  $rel-bset (=_{\alpha}) tset-x tset-z$ 
        unfolding rel-bset.rep-eq rel-set-def Ball-def Bex-def
        proof

```

```

show  $\forall x'. x' \in \text{set-bset } tset-x \longrightarrow (\exists z'. z' \in \text{set-bset } tset-z \wedge x' =_\alpha z')$ 
proof (rule allI, rule impI)
  fix  $x'$  assume  $1: x' \in \text{set-bset } tset-x$ 
  then obtain  $y'$  where  $2: y' \in \text{set-bset } tset-y$  and  $3: x' =_\alpha y'$ 
    by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
  from  $2$  obtain  $z'$  where  $4: z' \in \text{set-bset } tset-z$  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  $\exists z'. z' \in \text{set-bset } tset-z \wedge x' =_\alpha z'$ 
    by auto
  qed
next
show  $\forall z'. z' \in \text{set-bset } tset-z \longrightarrow (\exists x'. x' \in \text{set-bset } tset-x \wedge x' =_\alpha z')$ 
proof (rule allI, rule impI)
  fix  $z'$  assume  $1: z' \in \text{set-bset } tset-z$ 
  then obtain  $y'$  where  $2: y' \in \text{set-bset } tset-y$  and  $3: y' =_\alpha z'$ 
    by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1)
  from  $2$  obtain  $x'$  where  $4: x' \in \text{set-bset } tset-x$  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  $\exists x'. x' \in \text{set-bset } tset-x \wedge x' =_\alpha z'$ 
    by auto
  qed
qed
with  $z$  show  $tConj \text{ } tset-x =_\alpha z$ 
  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 ( $tAct \alpha1 t1 \alpha2 t2$ ) show ?case
proof (cases  $z$ )
  fix  $\alpha t$ 
  assume  $z: z = tAct \alpha t$ 
  obtain  $p$  where  $1: (bn \alpha1, t1) \approx_{set} (=_\alpha) (supp-rel (=_\alpha)) p (bn \alpha2, t2) \wedge$ 
 $(bn \alpha1, \alpha1) \approx_{set} (=) supp p (bn \alpha2, \alpha2)$ 
    using  $tAct.hyps$  by auto
  obtain  $q$  where  $2: (bn \alpha2, t2) \approx_{set} (=_\alpha) (supp-rel (=_\alpha)) q (bn \alpha, t) \wedge (bn$ 
 $\alpha2, \alpha2) \approx_{set} (=) supp q (bn \alpha, \alpha)$ 
    using  $tAct.prem1$  by auto
  have  $(bn \alpha1, t1) \approx_{set} (=_\alpha) (supp-rel (=_\alpha)) (q + p) (bn \alpha, t)$ 
proof -
    have  $supp-rel (=_\alpha) t1 - bn \alpha1 = supp-rel (=_\alpha) t - bn \alpha$ 

```

```

    using 1 and 2 by (metis alpha-set)
    moreover have (supp-rel (=α) t1 - bn α1) #* (q + p)
    using 1 and 2 by (metis alpha-set fresh-star-plus)
    moreover have (q + p) · t1 =α 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) · bn α1 = bn α
    using 1 and 2 by (metis alpha-set permute-plus)
    ultimately show ?thesis
    by (metis alpha-set)
  qed
  moreover have (bn α1, α1) ≈set (=) supp (q + p) (bn α, α)
  using 1 and 2 by (metis (mono-tags) alpha-trans(1) permute-plus)
  ultimately show tAct α1 t1 =α z
  using z by auto
  qed (insert tAct.premis, auto)
  qed
  qed

```

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

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

lemma *alpha-Tree-supp-rel*:

```

  assumes t1 =α t2
  shows supp-rel (=α) t1 = supp-rel (=α) t2
using assms proof (induction rule: alpha-Tree-induct)
case (tConj tset1 tset2)
  have sym:  $\bigwedge x y. \text{rel-bset } (=α) x y \longleftrightarrow \text{rel-bset } (=α) y x$ 
  by (meson alpha-Tree-symp bset.rel-symp sympE)
  {
    fix a b
    from tConj.hyps have  $*$ :  $\text{rel-bset } (=α) ((a \rightleftharpoons b) \cdot tset1) ((a \rightleftharpoons b) \cdot tset2)$ 
    by (metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj)
    have  $\text{rel-bset } (=α) ((a \rightleftharpoons b) \cdot tset1) tset1 \longleftrightarrow \text{rel-bset } (=α) ((a \rightleftharpoons b) \cdot tset2)$ 
    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 α1 t1 α2 t2)
  {
    fix a b
    have tAct α1 t1 =α tAct α2 t2
  }

```

```

    using tAct.hyps by simp
    then have (a  $\equiv$  b)  $\cdot$  tAct  $\alpha$ 1 t1  $=_{\alpha}$  tAct  $\alpha$ 1 t1  $\longleftrightarrow$  (a  $\equiv$  b)  $\cdot$  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 tAct  $\alpha$  t1  $=_{\alpha}$  tAct  $\alpha$  t2

```

proof –

```

  have (bn  $\alpha$ , t1)  $\approx_{\text{set}}$  ( $=_{\alpha}$ ) (supp-rel ( $=_{\alpha}$ )) 0 (bn  $\alpha$ , t2)
    using assms by (simp add: alpha-Tree-supp-rel alpha-set fresh-star-zero)
  moreover have (bn  $\alpha$ ,  $\alpha$ )  $\approx_{\text{set}}$  ( $=$ ) 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 *alpha*-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  $\varphi$ ) = supp  $\varphi$ 
unfolding supp-rel-def supp-def by simp

```

The support modulo α -equivalence of *tAct* α *t* is not easily described: when *t* has infinite support (modulo α -equivalence), the support (modulo α -equivalence) of *tAct* α *t* may still contain names in *bn* α . 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-rel ( $=_{\alpha}$ ) t - bn  $\alpha$ 

```

proof

```

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

```

proof

```

  fix x

```

```

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

```

```

  moreover

```

```

  {

```

```

    assume x1: x  $\in$  supp  $\alpha$  and x2: x  $\notin$  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-of  $x = \text{sort-of } b$ 
    using swap-different-sorts by fastforce
  then have  $(x \rightleftharpoons b) \cdot (\text{supp } \alpha - \text{bn } \alpha) \neq \text{supp } \alpha - \text{bn } \alpha$ 
    using  $b2$   $x1$   $x2$  by (simp add: swap-set-in)
  then have  $b \in \{b. \neg (x \rightleftharpoons b) \cdot \text{tAct } \alpha \ t =_{\alpha} \text{tAct } \alpha \ t\}$ 
    by (auto simp add: alpha-set Diff-eqvt bn-eqvt)
}
ultimately have infinite  $\{b. \neg (x \rightleftharpoons b) \cdot \text{tAct } \alpha \ t =_{\alpha} \text{tAct } \alpha \ t\}$ 
  by (rule infinite-mono)
then have  $x \in \text{supp-rel } (=_{\alpha}) (\text{tAct } \alpha \ t)$ 
  unfolding supp-rel-def ..
}
moreover
{
  assume  $x1: x \in \text{supp-rel } (=_{\alpha}) \ t$  and  $x2: x \notin \text{bn } \alpha$ 
  from  $x1$  have infinite  $\{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
    unfolding supp-rel-def ..
  then have infinite  $\{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\} - \text{supp-rel } (=_{\alpha}) \ t$ 
    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-eqvt bn-eqvt)
    then have  $b \in \{b. \neg (x \rightleftharpoons b) \cdot \text{tAct } \alpha \ t =_{\alpha} \text{tAct } \alpha \ t\}$ 
      by (simp add: alpha-set)
  }
}
ultimately have infinite  $\{b. \neg (x \rightleftharpoons b) \cdot \text{tAct } \alpha \ t =_{\alpha} \text{tAct } \alpha \ t\}$ 
  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 *:  $\text{infinite } \{b. \neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}$ 
    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  $\text{infinite } \{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha \vee \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
    by (metis (mono-tags, lifting) infinite-mono mem-Collect-eq)
  then have  $\text{infinite } \{b. (x \rightleftharpoons b) \cdot \alpha \neq \alpha\} \vee \text{infinite } \{b. \neg (x \rightleftharpoons b) \cdot t =_{\alpha} t\}$ 
    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  $b1: \neg (x \rightleftharpoons b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t$  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  $?p = (x \rightleftharpoons b)$ 
    have  $\text{supp-rel } (=_{\alpha}) ((x \rightleftharpoons b) \cdot t) - \text{bn } ((x \rightleftharpoons b) \cdot \alpha) = \text{supp-rel } (=_{\alpha}) t - \text{bn } \alpha$ 
      using ** and  $b3$  by (metis (no-types, lifting) Diff-eqt Diff-iff alpha-Tree-eqt' alpha-Tree-eqt-aux bn-eqt swap-set-not-in)
    moreover then have  $(\text{supp-rel } (=_{\alpha}) ((x \rightleftharpoons b) \cdot t) - \text{bn } ((x \rightleftharpoons b) \cdot \alpha)) \#* ?p$ 
      using ** and  $b3$  by (metis Diff-iff fresh-perm fresh-star-def swap-atom-simps(3))
    moreover have  $?p \cdot (x \rightleftharpoons b) \cdot t =_{\alpha} t$ 
      using alpha-Tree-reflp reflpE by force
    moreover have  $?p \cdot \text{bn } ((x \rightleftharpoons b) \cdot \alpha) = \text{bn } \alpha$ 
      by (simp add: bn-eqt)
    moreover have  $\text{supp } ((x \rightleftharpoons b) \cdot \alpha) - \text{bn } ((x \rightleftharpoons b) \cdot \alpha) = \text{supp } \alpha - \text{bn } \alpha$ 
      using ** and  $b2$  by (metis (mono-tags, opaque-lifting) Diff-eqt Diff-iff bn-eqt supp-eqt swap-set-not-in)
    moreover then have  $(\text{supp } ((x \rightleftharpoons b) \cdot \alpha) - \text{bn } ((x \rightleftharpoons b) \cdot \alpha)) \#* ?p$ 
      using ** and  $b2$  by (simp add: fresh-star-def fresh-def supp-perm) (metis

```

```

Diff-iff swap-atom-simps( $\beta$ )
  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 $_{\alpha}$ -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) 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 $_{\alpha}$  :: (-,-,-) 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*]: $p \cdot \text{rep-Tree}_\alpha t_\alpha =_\alpha \text{rep-Tree}_\alpha (p \cdot t_\alpha)$
by (*metis Tree_α.abs-eq-iff Tree_α-abs-rep permute-Tree_α.abs-eq*)

5.3 Constructors for trees modulo α -equivalence

The constructors are lifted from trees.

lift-definition *Conj_α* :: ('idx,'pred,'act) *Tree_α* set['idx] \Rightarrow ('idx,'pred::pt,'act::bn) *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) *Tree_α* \Rightarrow ('idx,'pred::pt,'act::bn) *Tree_α* **is**
tNot
by *simp*

lift-definition *Pred_α* :: 'pred \Rightarrow ('idx,'pred::pt,'act::bn) *Tree_α* **is**
tPred
 .

lift-definition *Act_α* :: 'act \Rightarrow ('idx,'pred,'act) *Tree_α* \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*]: $p \cdot \text{Conj}_\alpha tset_\alpha = \text{Conj}_\alpha (p \cdot tset_\alpha)$

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

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

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

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

The lifted constructors are injective (except for Act_α).

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

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

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*]: $Pred_\alpha \varphi1 = Pred_\alpha \varphi2 \longleftrightarrow \varphi1 = \varphi2$

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 $Act_\alpha-eq-iff$: $Act_\alpha \alpha1 t1 = Act_\alpha \alpha2 t2 \longleftrightarrow tAct \alpha1 (rep-Tree_\alpha t1) =_\alpha tAct \alpha2 (rep-Tree_\alpha t2)$

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 $Conj_\alpha tset_\alpha \neq Not_\alpha t_\alpha$
and $Conj_\alpha tset_\alpha \neq Pred_\alpha \varphi$
and $Conj_\alpha tset_\alpha \neq Act_\alpha \alpha t_\alpha$
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 $supp-alpha-supp-rel$: $supp t_\alpha = supp-rel (=_\alpha) (rep-Tree_\alpha t_\alpha)$

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*]: $supp (Conj_\alpha tset_\alpha) = supp tset_\alpha$

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

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

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

lemma *supp-Act $_{\alpha}$* [*simp*]:
assumes *finite* (*supp* t_{α})
shows $\text{supp } (\text{Act}_{\alpha} \alpha t_{\alpha}) = \text{supp } \alpha \cup \text{supp } t_{\alpha} - \text{bn } \alpha$
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 $\bigwedge tset_{\alpha}. (\bigwedge x. x \in \text{set-bset } tset_{\alpha} \implies P x) \implies P (\text{Conj}_{\alpha} tset_{\alpha})$
and $\bigwedge t_{\alpha}. P t_{\alpha} \implies P (\text{Not}_{\alpha} t_{\alpha})$
and $\bigwedge pred. P (\text{Pred}_{\alpha} pred)$
and $\bigwedge act t_{\alpha}. P t_{\alpha} \implies P (\text{Act}_{\alpha} act t_{\alpha})$
shows $P t_{\alpha}$
proof (*rule* *Tree $_{\alpha}$.abs-induct*)
fix t **show** $P (\text{abs-Tree}_{\alpha} t)$
proof (*induction* t)
case (*tConj* $tset$)
let $?tset_{\alpha} = \text{map-bset } \text{abs-Tree}_{\alpha} tset$
have $\text{abs-Tree}_{\alpha} (tConj tset) = \text{Conj}_{\alpha} ?tset_{\alpha}$
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* *Not $_{\alpha}$.abs-eq*)
next
case *tPred* **show** *?case*
using *assms*(3) **by** (*metis* *Pred $_{\alpha}$.abs-eq*)
next
case *tAct* **then show** *?case*
using *assms*(4) **by** (*metis* *Act $_{\alpha}$.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) $Tree_\alpha \Rightarrow bool$ **where**
Conj $_\alpha$: $finite (supp\ tset_\alpha) \Longrightarrow (\bigwedge t_\alpha. t_\alpha \in set\ bset\ tset_\alpha \Longrightarrow hereditarily\ fs\ t_\alpha) \Longrightarrow hereditarily\ fs\ (Conj_\alpha\ tset_\alpha)$
| *Not $_\alpha$* : $hereditarily\ fs\ t_\alpha \Longrightarrow hereditarily\ fs\ (Not_\alpha\ t_\alpha)$
| *Pred $_\alpha$* : $hereditarily\ fs\ (Pred_\alpha\ \varphi)$
| *Act $_\alpha$* : $hereditarily\ fs\ t_\alpha \Longrightarrow hereditarily\ fs\ (Act_\alpha\ \alpha\ t_\alpha)$

hereditarily-fs is equivariant.

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

assumes *hereditarily-fs* t_α

shows *hereditarily-fs* $(p \cdot t_\alpha)$

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

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

by (*metis* (*erased, opaque-lifting*) *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 $Act_\alpha\ \alpha\ t_\alpha = Act_\alpha\ \alpha'\ t'_\alpha$

shows $hereditarily\ fs\ t_\alpha \longleftrightarrow hereditarily\ fs\ t'_\alpha$

proof

assume *hereditarily-fs* t_α

then show *hereditarily-fs* t'_α

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'_α

then show *hereditarily-fs* t_α

```

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 $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha)$
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-eqvt*)
 moreover **have** *hereditarily-fs* $(p \cdot t_\alpha)$
 using *assms* **by** (*metis* *hereditarily-fs-eqvt*)
 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-eqvt* [*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-eqvt*)
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*
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 $Conj_\alpha$ 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 $Conj :: ('idx, 'pred, 'act)\ formula\ set['idx] \Rightarrow ('idx, 'pred::fs, 'act::bn)\ formula$ **where**

$Conj\ xset = Abs\ formula\ (Conj_\alpha\ (map\ bset\ Rep\ formula\ xset))$

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** $t_\alpha \in set\ bset\ (map\ bset\ Rep\ formula\ xset)$

then show *hereditarily-fs* t_α

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

qed

lemma *Conj-rep-eq*:

assumes *finite* (*supp* *xset*)

shows *Rep-formula* ($Conj\ xset$) = $Conj_\alpha\ (map\ bset\ Rep\ formula\ xset)$

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 $p \cdot \text{Conj } xset = \text{Conj } (p \cdot xset)$
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*]: $p \cdot \text{Pred } \varphi = \text{Pred } (p \cdot \varphi)$
by *transfer simp*

lemma *Act-eqvt* [*eqvt, simp*]: $p \cdot \text{Act } \alpha x = \text{Act } (p \cdot \alpha) (p \cdot x)$
by *transfer simp*

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows $\text{supp } (\text{Conj } xset) = \text{supp } xset$
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*]: $\text{supp } (\text{Pred } \varphi) = \text{supp } \varphi$
by (*metis Pred.rep-eq supp-Pred_α supp-Rep-formula*)

lemma *supp-Act* [*simp*]: $\text{supp } (\text{Act } \alpha x) = \text{supp } \alpha \cup \text{supp } x - \text{bn } \alpha$
by (*metis Act.rep-eq finite-supp supp-Act_α supp-Rep-formula*)

lemma *bn-fresh-Act* [*simp*]: $\text{bn } \alpha \#^* \text{Act } \alpha x$
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 $\text{Conj } xset1 = \text{Conj } xset2 \iff xset1 = xset2$
using *assms*
by (*metis (erased, opaque-lifting) 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*]: $\text{Pred } \varphi1 = \text{Pred } \varphi2 \iff \varphi1 = \varphi2$
by (*metis Pred.rep-eq Pred_α-eq-iff*)

lemma *Act-eq-iff*: $\text{Act } \alpha1 x1 = \text{Act } \alpha2 x2 \iff \text{Act}_\alpha \alpha1 (\text{Rep-formula } x1) = \text{Act}_\alpha \alpha2 (\text{Rep-formula } x2)$
by (*metis Act.rep-eq Rep-formula-inverse*)

Helpful lemmas for dealing with equalities involving *Act*.

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

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

(is ?l \longleftrightarrow ?r)

proof

assume ?l

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

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

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

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

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

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 $supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2$

by (*metis alpha-set.simps*)

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

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: $supp\ x1 - bn\ \alpha1 = supp\ x2 - bn\ \alpha2$ and 2: $(supp\ x1 - bn\ \alpha1) \#* p$ and 3: $p \cdot x1 = x2$

and 4: $supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2$ and 5: $(supp\ \alpha1 - bn\ \alpha1) \#* p$ and 6: $p \cdot \alpha1 = \alpha2$

by *metis*

from 1 2 3 6 have $(bn\ \alpha1, rep\ Tree_\alpha\ (Rep\ formula\ x1)) \approx_{set}\ (=_\alpha)\ (supp\ rel\ (=_\alpha))\ p\ (bn\ \alpha2, rep\ Tree_\alpha\ (Rep\ formula\ x2))$

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 $(bn\ \alpha1, \alpha1) \approx_{set}\ (=)\ supp\ p\ (bn\ \alpha2, \alpha2)$

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

ultimately show $Act\ \alpha1\ x1 = Act\ \alpha2\ x2$

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

qed

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

$(\exists p. supp\ x1 - bn\ \alpha1 = supp\ x2 - bn\ \alpha2 \wedge (supp\ x1 - bn\ \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2 \wedge (supp\ \alpha1 - bn\ \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2 \wedge supp\ p \subseteq bn\ \alpha1 \cup p \cdot bn\ \alpha1)$

(is ?l \longleftrightarrow ?r)

proof
assume ?l
then 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 $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Pred } \varphi$
and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Act } \alpha x$
and $\text{Not } x \neq \text{Pred } \varphi$
and $\text{Not } x1 \neq \text{Act } \alpha x2$
and $\text{Pred } \varphi \neq \text{Act } \alpha x$

proof –

show $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Not } x$
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 t $_{\alpha}$ )
  proof (induction t $_{\alpha}$ )
  case (Conj $_{\alpha}$  tset $_{\alpha}$ ) show ?case
  proof -
  let ?tset = map-bset Abs-formula tset $_{\alpha}$ 
  have  $\bigwedge$ t $_{\alpha}'$ . t $_{\alpha}'$   $\in$  set-bset tset $_{\alpha}$   $\implies$  t $_{\alpha}'$  = (Rep-formula  $\circ$  Abs-formula) t $_{\alpha}'$ 
    by (simp add: Conj $_{\alpha}$ .hyps)
  then have tset $_{\alpha}$  = map-bset (Rep-formula  $\circ$  Abs-formula) tset $_{\alpha}$ 
    by (metis bset.map-cong0 bset.map-id id-apply)
  then have *: tset $_{\alpha}$  = map-bset Rep-formula ?tset
    by (metis bset.map-comp)
  then have Abs-formula (Conj $_{\alpha}$  tset $_{\alpha}$ ) = Conj ?tset
    by (metis Conj-def)
  moreover from * have finite (supp ?tset)
    using Conj $_{\alpha}$ .hyps(1) by (metis supp-map-bset-Rep-formula)
  moreover have ( $\bigwedge$ t. t  $\in$  set-bset ?tset  $\implies$  P t)
    using Conj $_{\alpha}$ .IH by (metis imageE map-bset.rep-eq)
  
```

```

      ultimately show ?thesis
      using assms(1) by metis
    qed
  next
    case Not $\alpha$  then show ?case
    using assms(2) by (metis Formula.Abs-formula-inverse Not.rep-eq Rep-formula-inverse)
  next
    case Pred $\alpha$  show ?case
    using assms(3) by (metis Pred.abs-eq)
  next
    case Act $\alpha$  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  $\bigwedge xset\ c.\ finite\ (supp\ xset) \implies (\bigwedge x.\ x \in\ set-bset\ xset \implies (\bigwedge c.\ P\ c\ x))$ 
   $\implies P\ c\ (Conj\ xset)$ 
    and  $\bigwedge formula\ c.\ (\bigwedge c.\ P\ c\ formula) \implies P\ c\ (Not\ formula)$ 
    and  $\bigwedge pred\ c.\ P\ c\ (Pred\ pred)$ 
    and  $\bigwedge act\ formula\ c.\ bn\ act\ \#\ast\ c \implies (\bigwedge c.\ P\ c\ formula) \implies P\ c\ (Act\ act\ formula)$ 
  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  $(\bigwedge x\ c.\ x \in\ set-bset\ (p \cdot xset) \implies P\ c\ x)$ 
      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 ·  $\alpha$ ) to avoid c, without touching Act (p ·  $\alpha$ ) (p · x)
        obtain q where 1:  $(q \cdot bn\ (p \cdot \alpha))\ \#\ast\ c$  and 2:  $supp\ (Act\ (p \cdot \alpha)\ (p \cdot x))\ \#\ast\ q$ 
        proof (rule at-set-avoiding2[of bn (p ·  $\alpha$ ) c Act (p ·  $\alpha$ ) (p · x), THEN exE])
          show finite (bn (p ·  $\alpha$ )) by (fact bn-finite)
        next
          show finite (supp c) by (fact finite-supp)
      end
  end

```

```

next
  show finite (supp (Act ( $p \cdot \alpha$ ) ( $p \cdot x$ ))) by (simp add: finite-supp)
next
  show  $bn$  ( $p \cdot \alpha$ )  $\#*$  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-eqt)
moreover from Act.IH have  $\bigwedge c. P$   $c$  ( $q \cdot p \cdot x$ )
  by (metis permute-plus)
ultimately have  $P$   $c$  (Act ( $q \cdot p \cdot \alpha$ ) ( $q \cdot p \cdot x$ ))
  using assms(4) by simp
moreover from 2 have Act ( $q \cdot p \cdot \alpha$ ) ( $q \cdot p \cdot x$ ) = 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**
alpha-Tree-rel $\equiv \{(x,y). x =_{\alpha} y\}$

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

lemma *alpha-Tree-relE*:
assumes $(x,y) \in$ *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 $x :: ('d, 'e, 'f) \text{Tree} \times ('d, 'e, 'f) \text{Tree}$
 assume $x \in (\text{hull-rel } O \text{ Tree-wf}) \text{ } O \text{ alpha-Tree-rel}$
 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 $tset :: ('d, 'e, 'f) \text{Tree set}['d]$
 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 *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 $(p \cdot t', t') \in \text{hull-rel}$
 by (*rule hull-rel.intros*)
 moreover from x_4 have $(t', x_4) \in \text{Tree-wf}$
 using *Tree-wf.intros(2)* by *blast*
 ultimately show $(x_1, x_4) \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$
 by *auto*

```

next
  — tAct
  fix  $\alpha$  t
  assume *:  $(x1, t) \in \text{hull-rel}$  and **:  $(tAct\ \alpha\ t, x4) \in \text{alpha-Tree-rel}$ 
  from * obtain p where  $x1: x1 = p \cdot t$ 
    using hull-rel.cases by blast
  from ** have  $tAct\ \alpha\ t =_{\alpha}\ x4$ 
    by (rule alpha-Tree-relE)
  then obtain q t' where  $x4: x4 = tAct\ (q \cdot \alpha)\ t'$  and  $q \cdot t =_{\alpha}\ t'$ 
    by (cases x4) (auto simp add: alpha-set)
  with x1 have  $(x1, p \cdot -q \cdot t') \in \text{alpha-Tree-rel}$ 
    by (metis Tree $_{\alpha}$ .abs-eq-iff alpha-Tree-relI permute-Tree $_{\alpha}$ .abs-eq permute-minus-cancel(1))
  moreover have  $(p \cdot -q \cdot t', t') \in \text{hull-rel}$ 
    by (metis hull-rel.simps permute-plus)
  moreover from x4 have  $(t', x4) \in \text{Tree-wf}$ 
    by (simp add: Tree-wf.intros(3))
  ultimately show  $(x1, x4) \in \text{alpha-Tree-rel} \ O\ \text{hull-rel} \ O\ \text{Tree-wf}$ 
    by auto
  qed
with x show  $x \in \text{alpha-Tree-rel} \ O\ \text{hull-rel} \ O\ \text{Tree-wf}$ 
  by simp
qed
qed

```

```

lemma alpha-Tree-rel-relcomp-trivialI [simp]:
  assumes  $(x, y) \in R$ 
  shows  $(x, y) \in \text{alpha-Tree-rel} \ O\ R$ 
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  $(x', y) \in R$ 
  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  $\varphi$ )  $\longleftrightarrow$   $P \vdash \varphi$ 
| valid-Tree P (tAct  $\alpha t$ )  $\longleftrightarrow$  ( $\exists \alpha' t' P'. tAct \alpha t =_{\alpha} tAct \alpha' t' \wedge P \rightarrow \langle \alpha', P' \rangle$ )
 $\wedge \text{valid-Tree } P' t'$ )
by pat-completeness auto
termination proof
  let ?R = 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) 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) 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  $\cdot$  P) (p  $\cdot$  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  $\in$  p  $\cdot$  set-bset tset
      with 1.IH and * have valid-Tree (p  $\cdot$  P) t
      by (metis (no-types, lifting) imageE permute-set-eq-image valid-Tree.simps(1))
    }
    then show valid-Tree (p  $\cdot$  P) (p  $\cdot$  tConj tset)
    by simp
  next
    assume *: valid-Tree (p  $\cdot$  P) (p  $\cdot$  tConj tset)
    {
      fix t
      assume t  $\in$  set-bset tset

```

```

    with 1.IH and * have valid-Tree P t
    by (metis mem-permute-iff permute-Tree-tConj set-bset-eqvt 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-eqvt)
next
  case (4 P α t) show ?case
  proof
    assume valid-Tree P (tAct α t)
    then obtain α' t' P' where *: tAct α t =α tAct α' t' ∧ P → ⟨α', P'⟩ ∧
    valid-Tree P' t'
    by auto
    with 4.IH have valid-Tree (p · P') (p · t')
    by blast
    moreover from * have p · P → ⟨p · α', p · P'⟩
    by (metis transition-eqvt')
    moreover from * have p · tAct α t =α tAct (p · α') (p · t')
    by (metis alpha-Tree-eqvt 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 α' t' P' where *: p · tAct α t =α tAct α' t' ∧ (p · P) →
    ⟨α', P'⟩ ∧ valid-Tree P' t'
    by auto
    then have eq: tAct α t =α tAct (-p · α') (-p · t')
    by (metis alpha-Tree-eqvt permute-Tree.simps(4) permute-minus-cancel(2))
    moreover from * have P → ⟨-p · α', -p · P'⟩
    by (metis permute-minus-cancel(2) transition-eqvt')
    moreover with 4.IH have valid-Tree (-p · P') (-p · t')
    using eq and * by simp
    ultimately show valid-Tree P (tAct α t)
    by auto
  qed
  qed
qed

```

lemma *valid-Tree-eqvt* :

assumes *valid-Tree P t* shows *valid-Tree (p · P) (p · 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 *valid-Tree P t1* \longleftrightarrow *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  $\alpha 1$  t1  $\alpha 2$  t2) show ?case
  proof
    assume valid-Tree P (tAct  $\alpha 1$  t1)
    then obtain  $\alpha' t' P'$  where tAct  $\alpha 1$  t1  $=_{\alpha}$  tAct  $\alpha' t'$   $\wedge P \rightarrow \langle \alpha', P \rangle \wedge$ 
valid-Tree  $P' t'$ 
    by auto
    moreover from tAct.hyps have tAct  $\alpha 1$  t1  $=_{\alpha}$  tAct  $\alpha 2$  t2
    using alpha-tAct by blast
    ultimately show valid-Tree P (tAct  $\alpha 2$  t2)
    by (metis Treeα.abs-eq-iff valid-Tree.simps(4))
  next
    assume valid-Tree P (tAct  $\alpha 2$  t2)
    then obtain  $\alpha' t' P'$  where tAct  $\alpha 2$  t2  $=_{\alpha}$  tAct  $\alpha' t'$   $\wedge P \rightarrow \langle \alpha', P \rangle \wedge$ 
valid-Tree  $P' t'$ 
    by auto
    moreover from tAct.hyps have tAct  $\alpha 1$  t1  $=_{\alpha}$  tAct  $\alpha 2$  t2
    using alpha-tAct by blast
    ultimately show valid-Tree P (tAct  $\alpha 1$  t1)
    by (metis Treeα.abs-eq-iff valid-Tree.simps(4))
  qed
qed simp-all

```

6.2 Validity for trees modulo α -equivalence

```

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

```

```

lemma valid-Treeα-eqvt :
  assumes valid-Treeα P t shows valid-Treeα (p  $\cdot$  P) (p  $\cdot$  t)
using assms by transfer (fact valid-Tree-eqvt)

```

```

lemma valid-Treeα-Conjα [simp]: valid-Treeα P (Conjα tsetα)  $\longleftrightarrow$  ( $\forall t_{\alpha} \in \text{set-bset}$ 
tsetα. valid-Treeα P tα)
proof –
  have valid-Tree P (rep-Treeα (abs-Treeα (tConj (map-bset rep-Treeα tsetα))))
 $\longleftrightarrow$  valid-Tree P (tConj (map-bset rep-Treeα tsetα))
  by (metis Treeα-rep-abs alpha-Tree-valid-Tree)
  then show ?thesis
  by (simp add: valid-Treeα-def Conjα-def map-bset.rep-eq)
qed

```

```

lemma valid-Treeα-Notα [simp]: valid-Treeα P (Notα tα)  $\longleftrightarrow$   $\neg$  valid-Treeα P
tα
by transfer simp

```

lemma *valid-Tree $_{\alpha}$ -Pred $_{\alpha}$ [simp]*: $\text{valid-Tree}_{\alpha} P (\text{Pred}_{\alpha} \varphi) \longleftrightarrow P \vdash \varphi$
by *transfer simp*

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

proof

assume $\text{valid-Tree}_{\alpha} P (\text{Act}_{\alpha} \alpha t_{\alpha})$

moreover have $\text{Act}_{\alpha} \alpha t_{\alpha} = \text{abs-Tree}_{\alpha} (t\text{Act } \alpha (\text{rep-Tree}_{\alpha} t_{\alpha}))$

by (*metis Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.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 $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-eq-iff valid-Tree.simps(4) valid-Tree $_{\alpha}$.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 $\bigwedge \alpha' t_{\alpha}'. \text{Act}_{\alpha} \alpha' t_{\alpha}' = \text{abs-Tree}_{\alpha} (t\text{Act } \alpha' (\text{rep-Tree}_{\alpha} t_{\alpha}'))$

by (*metis Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-rep*)

ultimately show $\text{valid-Tree}_{\alpha} P (\text{Act}_{\alpha} \alpha t_{\alpha})$

by (*metis Tree $_{\alpha}$.abs-eq-iff valid-Tree.simps(4) valid-Tree $_{\alpha}$.abs-eq valid-Tree $_{\alpha}$.rep-eq*)

qed

6.3 Validity for infinitary formulas

lift-definition *valid* :: 'state \Rightarrow ('idx, 'pred, 'act) formula \Rightarrow bool (**infix** \models 70) is
 $\text{valid-Tree}_{\alpha}$

.

lemma *valid-eqvt* :

assumes $P \models x$ **shows** $(p \cdot P) \models (p \cdot x)$

using *assms* **by** *transfer (metis valid-Tree $_{\alpha}$ -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 $\text{Rep-formula } (\text{Abs-formula } (\text{Act}_{\alpha} \alpha (\text{Rep-formula } x))) = \text{Act}_{\alpha} \alpha (\text{Rep-formula } x)$

$\alpha (\text{Rep-formula } x)$

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*: $\text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P'$
and *valid*: $P' \models x'$
 by (*metis valid-Act*)
have *finite (bn α')*
 by (*fact bn-finite*)
moreover note $\langle \text{finite (supp X)} \rangle$
moreover have *finite (supp (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 (Act $\alpha' x'$, $\langle \alpha', P' \rangle$) $\#* p$*
 by (*metis at-set-avoiding2*)
then have *supp (Act $\alpha' x'$) $\#* p$ and *supp $\langle \alpha', P' \rangle \#* p$*
 by (*metis fresh-star-Un supp-Pair*)
then have $\text{Act } (p \cdot \alpha') (p \cdot x') = \text{Act } \alpha' x' \wedge P$ **and** $\langle p \cdot \alpha', p \cdot P' \rangle = \langle \alpha', P' \rangle$
 by (*metis Act-eqvt supp-perm-eq, metis abs-residual-pair-eqvt supp-perm-eq*)
then show $\exists \alpha' x' P'. \text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \#* X$
 using *eq and trans and valid and fresh-X* by (*metis bn-eqvt valid-eqvt*)
next
assume $\exists \alpha' x' P'. \text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \#* X$
then show $P \models \text{Act } \alpha x$
 by (*metis valid-Act*)
qed*

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

```

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

moreover have  $\text{finite } (\text{supp } P)$ 
  by (fact finite-supp)
ultimately 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
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  $P \rightarrow \langle \alpha, -p \cdot P' \rangle$ 
  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  $\exists P'. P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x$ 
  by meson
next
  assume  $\exists P'. P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x$  then show  $P \models \text{Act } \alpha \ x$ 
  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 ::  $'state::fs \Rightarrow 'pred::fs \Rightarrow \text{bool}$  (infix  $\vdash 70$ )
  and transition ::  $'state \Rightarrow ('act::bn, 'state) \text{residual} \Rightarrow \text{bool}$  (infix  $\rightarrow 70$ ) +
  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 *logically-equivalent* :: 'state \Rightarrow 'state \Rightarrow bool **where**
logically-equivalent $P\ Q \equiv (\forall x::('idx,'pred,'act)\ \text{formula}. P \models x \longleftrightarrow Q \models x)$

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

lemma *logically-equivalent-eqv*:

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-eqv*)

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

shows $Q \models \text{Act}\ \alpha\ x$

proof –

have *finite* (*supp* Q)

by (*fact finite-supp*)

with $\langle P \models \text{Act}\ \alpha\ x \rangle$ **obtain** $\alpha'\ x'\ P'$ **where** *eq*: $\text{Act}\ \alpha\ x = \text{Act}\ \alpha'\ x'$ **and**
trans: $P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$ **and** *fresh*: $\text{bn}\ \alpha' \#* Q$

by (*metis valid-Act-strong*)

from $\langle P \sim Q \rangle$ **and** *fresh* **and** *trans* **obtain** Q' **where** *trans'*: $Q \rightarrow \langle \alpha', Q' \rangle$
and *bisim'*: $P' \sim Q'$

by (*metis bisimilar-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-eqv*)

moreover from *bisim'* **have** $(-p \cdot P') \sim (-p \cdot Q')$

by (*metis bisimilar-eqv*)

ultimately have $-p \cdot Q' \models x$

```

  using ⟨ $\wedge P Q. P \sim Q \implies P \models x \longleftrightarrow Q \models x$ ⟩ by metis
  with px have  $Q' \models x'$ 
  by (metis permute-minus-cancel(1) valid-eqvt)

```

```

  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) 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  $\alpha x$ ) 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 *is-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 *is-distinguishing-formula-eqvt* :

assumes x distinguishes P from Q shows $(p \cdot x)$ distinguishes $(p \cdot P)$ from $(p \cdot Q)$

using *assms unfolding is-distinguishing-formula-def*
by (*metis permute-minus-cancel(2) valid-eqvt*)

lemma *equivalent-iff-not-distinguished*: $(P =\cdot Q) \longleftrightarrow \neg(\exists x. x \text{ distinguishes } P \text{ from } Q)$
by (*metis (full-types) is-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 $\text{supp } P$ 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*: $\text{supp } ?B \subseteq \text{supp } P$

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 is-distinguishing-formula-def proof
  from assms show P ⊨ ?y
  by (auto simp add: card-B finite-supp-B) (metis is-distinguishing-formula-def
supp-perm-eq valid-eqt)
next
  from assms show ¬ Q ⊨ ?y
  by (auto simp add: card-B finite-supp-B) (metis is-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 φ assume P =· Q then have P ⊢ φ → Q ⊢ φ

by (metis logically-equivalent-def valid-Pred)

}

moreover

{

fix P Q α P' assume P =· Q and bn α #* Q and P → ⟨α, P'⟩

then have ∃ Q'. Q → ⟨α, Q'⟩ ∧ P' =· Q'

proof –

{

let ?Q' = {Q'. Q → ⟨α, Q'⟩}

assume ∀ Q' ∈ ?Q'. ¬ P' =· Q'

then have ∀ Q' ∈ ?Q'. ∃ x :: ('idx, 'pred, 'act) formula. x distinguishes P'

from Q'

by (metis equivalent-iff-not-distinguished)

then have ∀ Q' ∈ ?Q'. ∃ x :: ('idx, 'pred, 'act) formula. supp x ⊆ supp P'

```

 $\wedge x$  distinguishes  $P'$  from  $Q'$ 
  by (metis distinguished-bounded-support)
  then obtain  $f :: 'state \Rightarrow ('idx, 'pred, 'act)$  formula where
    *:  $\forall Q' \in ?Q'. \text{supp } (f Q') \subseteq \text{supp } P' \wedge (f Q')$  distinguishes  $P'$  from  $Q'$ 
    by metis
  have  $\text{supp } (f ' ?Q') \subseteq \text{supp } P'$ 
    by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)
  then have finite-supp-image: finite (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:  $|f ' ?Q'| < o \text{ natLeq } + c \mid UNIV :: 'idx \text{ set}$  .
  let  $?y = \text{Conj } (\text{Abs-bset } (f ' ?Q')) :: ('idx, 'pred, 'act)$  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 is-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  $\langle \text{bn } \alpha \ \# * \ Q \rangle$  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 is-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 *Disjunction*

imports

Formula

Validity

begin

10 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 $p \cdot \text{Disj } xset = \text{Disj } (p \cdot xset)$

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

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

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

shows $\text{Disj } xset1 = \text{Disj } xset2 \iff xset1 = xset2$

using *assms* **unfolding** *Disj-def* **by** (*metis Conj-eq-iff Not-eq-iff bset.inj-map-strong finite-supp-map-bset-Not*)

context *nominal-ts*

begin

lemma *valid-Disj* [*simp*]:

assumes *finite (supp xset)*

```

    shows  $P \models \text{Disj } xset \iff (\exists x \in \text{set-bset } xset. P \models x)$ 
    using assms by (simp add: Disj-def map-bset.rep-eq)

end

end

theory Expressive-Completeness
imports
  Bisimilarity-Implies-Equivalence
  Equivalence-Implies-Bisimilarity
  Disjunction
begin

```

11 Expressive Completeness

```

context indexed-nominal-ts
begin

```

11.1 Distinguishing formulas

Lemma *distinguished_bounded_support* only shows the existence of a distinguishing formula, without stating what this formula looks like. We now define an explicit function that returns a distinguishing formula, in a way that this function is equivariant (on pairs of non-equivalent states).

Note that this definition uses Hilbert's choice operator ε , which is not necessarily equivariant. This is immediately remedied by a hull construction.

definition *distinguishing-formula* :: *'state* \Rightarrow *'state* \Rightarrow (*'idx*, *'pred*, *'act*) *formula*
where

distinguishing-formula $P Q \equiv \text{Conj } (\text{Abs-bset } \{-p \cdot (\varepsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))\} | p. \text{True})$

— just an auxiliary lemma that will be useful further below

lemma *distinguishing-formula-card-aux*:

$|\{-p \cdot (\varepsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))\} | p. \text{True}| <_o \text{natLeq } +c \mid \text{UNIV} :: \text{'idx set}$

proof —

let *?some* = $\lambda p. (\varepsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))$

let *?B* = $\{-p \cdot ?some p \mid p. \text{True}\}$

have $?B \subseteq (\lambda p. -p \cdot ?some p) \text{' UNIV}$

by *auto*

then have $|?B| \leq_o \mid \text{UNIV} :: \text{perm set}$

by (*rule surj-imp-ordLeq*)

also have $\mid \text{UNIV} :: \text{perm set} \mid <_o \mid \text{UNIV} :: \text{'idx set}$

by (*metis card-idx-perm*)

also have $\mid \text{UNIV} :: \text{'idx set} \mid \leq_o \text{natLeq } +c \mid \text{UNIV} :: \text{'idx set}$

```

    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally show ?thesis .
qed

— just an auxiliary lemma that will be useful further below
lemma distinguishing-formula-supp-aux:
  assumes  $\neg (P =\cdot Q)$ 
  shows  $\text{supp } (Abs\text{-bset } \{-p \cdot (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))\} | p. \text{True}\} :: \text{set}[idx]) \subseteq \text{supp } P$ 
  proof –
    let ?some =  $\lambda p. (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))$ 
    let ?B =  $\{-p \cdot ?some\ p | p. \text{True}\}$ 

    {
      fix p
      from assms have  $\neg (p \cdot P =\cdot p \cdot Q)$ 
        by (metis logically-equivalent-eqvt permute-minus-cancel(2))
      then have  $\text{supp } (?some\ p) \subseteq \text{supp } (p \cdot P)$ 
        using distinguished-bounded-support by (metis (mono-tags, lifting) equivalent-iff-not-distinguished someI-ex)
    }
    note supp-some = this

    {
      fix x
      assume  $x \in ?B$ 
      then obtain p where  $x = -p \cdot ?some\ p$ 
        by blast
      with supp-some have  $\text{supp } (p \cdot x) \subseteq \text{supp } (p \cdot P)$ 
        by simp
      then have  $\text{supp } x \subseteq \text{supp } P$ 
        by (metis (full-types) permute-boolE subset-eqvt supp-eqvt)
    }
    note * = this
    have supp-B:  $\text{supp } ?B \subseteq \text{supp } P$ 
      by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)

    from supp-B and distinguishing-formula-card-aux show ?thesis
      using supp-Abs-bset by blast
qed

lemma distinguishing-formula-eqvt [simp]:
  assumes  $\neg (P =\cdot Q)$ 
  shows  $p \cdot \text{distinguishing-formula } P\ Q = \text{distinguishing-formula } (p \cdot P)\ (p \cdot Q)$ 
  proof –
    let ?some =  $\lambda p. (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))$ 
    let ?B =  $\{-p \cdot ?some\ p | p. \text{True}\}$ 

```

```

from assms have supp (Abs-bset ?B :: - set['idx])  $\subseteq$  supp P
  by (rule distinguishing-formula-supp-aux)
then have finite (supp (Abs-bset ?B :: - set['idx]))
  using finite-supp rev-finite-subset by blast
with distinguishing-formula-card-aux have  $*$ :  $p \cdot \text{Conj (Abs-bset ?B)} = \text{Conj (Abs-bset (p \cdot ?B))}$ 
  by simp

let  $?some' = \lambda p'. (\epsilon x. \text{supp } x \subseteq \text{supp } (p' \cdot p \cdot P) \wedge x \text{ distinguishes } (p' \cdot p \cdot P)$ 
from (p' \cdot p \cdot Q))
let  $?B' = \{-p' \cdot ?some' p' | p'. \text{True}\}$ 

have  $p \cdot ?B = ?B'$ 
proof
  {
    fix  $px$ 
    assume  $px \in p \cdot ?B$ 
    then obtain  $x$  where  $1: px = p \cdot x$  and  $2: x \in ?B$ 
      by (metis (no-types, lifting) image-iff permute-set-eq-image)
    from  $2$  obtain  $p'$  where  $3: x = -p' \cdot ?some' p'$ 
      by blast
    from  $1$  and  $3$  have  $px = -(p' - p) \cdot ?some' (p' - p)$ 
      by simp
    then have  $px \in ?B'$ 
      by blast
  }
then show  $p \cdot ?B \subseteq ?B'$ 
  by blast
next
  {
    fix  $x$ 
    assume  $x \in ?B'$ 
    then obtain  $p'$  where  $x = -p' \cdot ?some' p'$ 
      by blast
    then have  $x = p \cdot -(p' + p) \cdot ?some' (p' + p)$ 
      by (simp add: add.inverse-distrib-swap)
    then have  $x \in p \cdot ?B$ 
      using mem-permute-iff by blast
  }
then show  $?B' \subseteq p \cdot ?B$ 
  by blast
qed

with  $*$  show ?thesis
  unfolding distinguishing-formula-def by simp
qed

```

lemma *supp-distinguishing-formula*:

```

assumes  $\neg (P =\cdot Q)$ 
shows  $\text{supp } (\text{distinguishing-formula } P Q) \subseteq \text{supp } P$ 
proof -
  let  $?some = \lambda p. (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p$ 
   $\cdot Q))$ 
  let  $?B = \{- p \cdot ?some \mid p. \text{True}\}$ 

  from assms have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]) \subseteq \text{supp } P$ 
  by (rule distinguishing-formula-supp-aux)
  moreover from this have  $\text{finite } (\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]))$ 
  using finite-supp rev-finite-subset by blast
  ultimately show ?thesis
  unfolding distinguishing-formula-def by simp
qed

lemma distinguishing-formula-distinguishes:
  assumes  $\neg (P =\cdot Q)$ 
  shows  $(\text{distinguishing-formula } P Q) \text{ distinguishes } P \text{ from } Q$ 
proof -
  let  $?some = \lambda p. (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p$ 
   $\cdot Q))$ 
  let  $?B = \{- p \cdot ?some \mid p. \text{True}\}$ 

  {
    fix  $p$ 
    have  $(?some \ p) \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q)$ 
    using assms
    by (metis (mono-tags, lifting) is-distinguishing-formula-eqvt distinguished-bounded-support
equivalent-iff-not-distinguished someI-ex)
  }
  note some-distinguishes = this

  {
    fix  $P'$ 
    from assms have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]) \subseteq \text{supp } P$ 
    by (rule distinguishing-formula-supp-aux)
    then have  $\text{finite } (\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]))$ 
    using finite-supp rev-finite-subset by blast
    with distinguishing-formula-card-aux have  $P' \models \text{distinguishing-formula } P Q$ 
 $\iff (\forall x \in ?B. P' \models x)$ 
    unfolding distinguishing-formula-def by simp
  }
  note valid-distinguishing-formula = this

  {
    fix  $p$ 
    have  $P \models -p \cdot ?some \ p$ 
    by (metis (mono-tags) is-distinguishing-formula-def permute-minus-cancel(2)
some-distinguishes valid-eqvt)
  }

```



```

}
then have  $P \models \text{distinguishing-formula } P \ Q$ 
using valid-distinguishing-formula by blast

moreover have  $\neg Q \models \text{distinguishing-formula } P \ Q$ 
using valid-distinguishing-formula by (metis (mono-tags, lifting) is-distinguishing-formula-def
mem-Collect-eq permute-minus-cancel(1) some-distinguishes valid-eqvt)

ultimately show (distinguishing-formula  $P \ Q$ ) distinguishes  $P$  from  $Q$ 
using is-distinguishing-formula-def by blast
qed

```

11.2 Characteristic formulas

A *characteristic formula* for a state P is valid for (exactly) those states that are bisimilar to P .

definition *characteristic-formula* $:: 'state \Rightarrow ('idx, 'pred, 'act) \text{ formula where}$
characteristic-formula $P \equiv \text{Conj } (Abs\text{-bset } \{ \text{distinguishing-formula } P \ Q \mid Q. \neg$
 $(P =. Q) \})$

— just an auxiliary lemma that will be useful further below

lemma *characteristic-formula-card-aux*:

$|\{ \text{distinguishing-formula } P \ Q \mid Q. \neg (P =. Q) \}| < o \ \text{natLeq} + c \mid UNIV :: 'idx \ \text{set}$

proof —

let $?B = \{ \text{distinguishing-formula } P \ Q \mid Q. \neg (P =. Q) \}$

have $?B \subseteq (\text{distinguishing-formula } P) \ 'UNIV$

by *auto*

then have $|?B| \leq o \mid UNIV :: 'state \ \text{set}$

by (*rule surj-imp-ordLeq*)

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 show *?thesis* .

qed

— just an auxiliary lemma that will be useful further below

lemma *characteristic-formula-supp-aux*:

shows $\text{supp } (Abs\text{-bset } \{ \text{distinguishing-formula } P \ Q \mid Q. \neg (P =. Q) \}) :: - \ \text{set}['idx]$
 $\subseteq \text{supp } P$

proof —

let $?B = \{ \text{distinguishing-formula } P \ Q \mid Q. \neg (P =. Q) \}$

{

fix x

assume $x \in ?B$

then obtain Q **where** $x = \text{distinguishing-formula } P \ Q$ **and** $\neg (P =. Q)$

by *blast*

```

    with supp-distinguishing-formula have  $\text{supp } x \subseteq \text{supp } P$ 
      by metis
  }
  note * = this
  have supp-B:  $\text{supp } ?B \subseteq \text{supp } P$ 
    by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)

  from supp-B and characteristic-formula-card-aux show ?thesis
    using supp-Abs-bset by blast
qed

lemma characteristic-formula-eqvt [simp]:
   $p \cdot \text{characteristic-formula } P = \text{characteristic-formula } (p \cdot P)$ 
proof -
  let  $?B = \{\text{distinguishing-formula } P \ Q \mid Q. \neg (P = \cdot Q)\}$ 

  have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['\text{id}x]) \subseteq \text{supp } P$ 
    by (fact characteristic-formula-supp-aux)
  then have finite ( $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['\text{id}x])$ )
    using finite-supp rev-finite-subset by blast
  with characteristic-formula-card-aux have *:  $p \cdot \text{Conj } (\text{Abs-bset } ?B) = \text{Conj } (\text{Abs-bset } (p \cdot ?B))$ 
    by simp

  let  $?B' = \{\text{distinguishing-formula } (p \cdot P) \ Q \mid Q. \neg ((p \cdot P) = \cdot Q)\}$ 

  have  $p \cdot ?B = ?B'$ 
proof
  {
    fix px
    assume  $px \in p \cdot ?B$ 
    then obtain x where 1:  $px = p \cdot x$  and 2:  $x \in ?B$ 
      by (metis (no-types, lifting) image-iff permute-set-eq-image)
    from 2 obtain Q where 3:  $x = \text{distinguishing-formula } P \ Q$  and 4:  $\neg (P = \cdot Q)$ 
  }
  by blast
  with 1 have  $px = \text{distinguishing-formula } (p \cdot P) \ (p \cdot Q)$ 
    by simp
  moreover from 4 have  $\neg ((p \cdot P) = \cdot (p \cdot Q))$ 
    by (metis logically-equivalent-eqvt permute-minus-cancel(2))
  ultimately have  $px \in ?B'$ 
    by blast
}
then show  $p \cdot ?B \subseteq ?B'$ 
  by blast
next
{
  fix x
  assume  $x \in ?B'$ 

```

```

    then obtain  $Q$  where 1:  $x = \text{distinguishing-formula } (p \cdot P) Q$  and 2:  $\neg$ 
     $((p \cdot P) = \cdot Q)$ 
    by blast
    from 2 have  $\neg (P = \cdot (-p \cdot Q))$ 
    by (metis logically-equivalent-eqvt permute-minus-cancel(1))
    moreover from this and 1 have  $x = p \cdot \text{distinguishing-formula } P (-p \cdot$ 
 $Q)$ 
    by simp
    ultimately have  $x \in p \cdot ?B$ 
    using mem-permute-iff by blast
  }
  then show  $?B' \subseteq p \cdot ?B$ 
  by blast
qed

with * show ?thesis
  unfolding characteristic-formula-def by simp
qed

lemma characteristic-formula-eqvt-raw [simp]:
   $p \cdot \text{characteristic-formula} = \text{characteristic-formula}$ 
  by (simp add: permute-fun-def)

lemma characteristic-formula-is-characteristic':
   $Q \models \text{characteristic-formula } P \longleftrightarrow P = \cdot Q$ 
proof -
  let  $?B = \{\text{distinguishing-formula } P Q \mid Q. \neg (P = \cdot Q)\}$ 

  {
    fix  $P'$ 
    have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['\text{id}x]) \subseteq \text{supp } P$ 
    by (fact characteristic-formula-supp-aux)
    then have finite ( $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['\text{id}x])$ )
    using finite-supp rev-finite-subset by blast
    with characteristic-formula-card-aux have  $P' \models \text{characteristic-formula } P \longleftrightarrow$ 
     $(\forall x \in ?B. P' \models x)$ 
    unfolding characteristic-formula-def by simp
  }
  note valid-characteristic-formula = this

  show ?thesis
  proof
    assume *:  $Q \models \text{characteristic-formula } P$ 
    show  $P = \cdot Q$ 
    proof (rule ccontr)
      assume  $\neg (P = \cdot Q)$ 
      with * show False
      using distinguishing-formula-distinguishes is-distinguishing-formula-def
      valid-characteristic-formula by auto
    end
  end

```

```

qed
next
assume P =· Q
moreover have P ⊨ characteristic-formula P
  using distinguishing-formula-distinguishes is-distinguishing-formula-def
valid-characteristic-formula by auto
ultimately show Q ⊨ characteristic-formula P
  using logically-equivalent-def by blast
qed
qed

```

```

lemma characteristic-formula-is-characteristic:
  Q ⊨ characteristic-formula P ⟷ P ∼· Q
  using characteristic-formula-is-characteristic' by (meson bisimilarity-implies-equivalence
equivalence-implies-bisimilarity)

```

11.3 Expressive completeness

Every finitely supported set of states that is closed under bisimulation can be described by a formula; namely, by a disjunction of characteristic formulas.

theorem *expressive-completeness*:

```

assumes finite (supp S)
and ∧P Q. P ∈ S ⟹ P ∼· Q ⟹ Q ∈ S
shows P ⊨ Disj (Abs-bset (characteristic-formula ‘ S)) ⟷ P ∈ S

```

proof –

```

let ?B = characteristic-formula ‘ S

```

```

have ?B ⊆ characteristic-formula ‘ UNIV
  by auto

```

```

then have |?B| ≤o |UNIV :: 'state set|
  by (rule surj-imp-ordLeq)

```

```

also have |UNIV :: 'state set| <o |UNIV :: 'idx set|
  by (metis card-idx-state)

```

```

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| .

```

```

have eqvt image and eqvt characteristic-formula
  by (simp add: eqvtI)+

```

```

then have supp ?B ⊆ supp S

```

```

  using supp-fun-eqvt supp-fun-app supp-fun-app-eqvt by blast

```

```

with card-B have supp (Abs-bset ?B :: - set['idx]) ⊆ supp S

```

```

  using supp-Abs-bset by blast

```

```

with ⟨finite (supp S)⟩ have finite (supp (Abs-bset ?B :: - set['idx']))

```

```

  using finite-supp rev-finite-subset by blast

```

```

  with card-B have P ⊨ Disj (Abs-bset (characteristic-formula ‘ S)) ⟷

```

```
(∃ x ∈ ?B. P ⊨ x)
```

```

  by simp

```

```

with  $\langle \bigwedge P Q. P \in S \implies P \sim \cdot Q \implies Q \in S \rangle$  show ?thesis
using characteristic-formula-is-characteristic characteristic-formula-is-characteristic'
logically-equivalent-def by fastforce
qed

```

```
end
```

```
end
```

```
theory FS-Set
```

```
imports
```

```
  Nominal2.Nominal2
```

```
begin
```

12 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 =  $\{x::'a::pt \text{ set. finite (supp } x)\}$ 
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 ::  $perm \Rightarrow 'a \text{ fs-set} \Rightarrow 'a \text{ fs-set}$  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  $x :: 'a \text{ fs-set}$ 
```

```
from Rep-fs-set have finite (supp (Rep-fs-set x)) by simp
```

```
hence finite {a. infinite {b. (a  $\equiv$  b)  $\cdot$  Rep-fs-set x  $\neq$  Rep-fs-set x}} by (unfold
```

supp-def)
hence *finite* {*a*. *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

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

13.1 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-eqt*: eqvt *effect-apply*
and *L-eqt*: eqvt *L* — *L* is assumed to be equivariant.

begin

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

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

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

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

end

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

context *effect-nominal-ts*

begin

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

is-L-bisimulation *R* ≡
 $\forall F. \text{symp } (R F) \wedge$
 $(\forall P Q. R F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))) \wedge$
 $(\forall P Q. R F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow$
 $\langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge R (L (\alpha, F, f)) P' Q'))))$

definition *FL-bisimilar* :: 'effect first ⇒ 'state ⇒ 'state ⇒ bool **where**
 $FL\text{-bisimilar } F P Q \equiv \exists R. \text{is-L-bisimulation } R \wedge (R F) P Q$

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

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

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

have *symp* $((p \cdot R) F)$ **(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 $pR: (p \cdot R) F P Q$ **and** *effect: $f \in_{fs} F$* **and** *satisfies: $\langle f \rangle P \vdash \varphi$*
from pR **have** $R (-p \cdot F) (-p \cdot P) (-p \cdot Q)$
by *(simp add: eqvt-lambda permute-bool-def unpermute-def)*
moreover have $(-p \cdot f) \in_{fs} (-p \cdot F)$
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 $pR: (p \cdot R) F P Q$ **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** $R (-p \cdot F) (-p \cdot P) (-p \cdot Q)$
by *(simp add: eqvt-lambda permute-bool-def unpermute-def)*
moreover have $(-p \cdot f) \in_{fs} (-p \cdot F)$
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** $(p \cdot R) (p \cdot L (-p \cdot \alpha, -p \cdot F, -p \cdot f)) (p \cdot -p \cdot P') (p \cdot Q')$
by *(metis permute-boolI permute-fun-def permute-minus-cancel(2))*
then have $(p \cdot R) (L (\alpha, F, f)) P' (p \cdot Q')$
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 $P \sim_{\cdot[F]} Q$ shows $(p \cdot P) \sim_{\cdot[p \cdot F]} (p \cdot Q)$
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 $x \sim_{\cdot[F]} x$
unfolding *FL-bisimilar-def* by *auto*
qed

lemma *FL-bisimilar-symp*: *symp* (*FL-bisimilar* F)
proof (*rule sympI*)
fix $P Q$
assume $P \sim_{\cdot[F]} Q$
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 $Q \sim_{\cdot[F]} P$
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 $\forall P Q. P \sim_{\cdot[F]} Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \longrightarrow \langle f \rangle Q \vdash \varphi))$ (*is* $?S$)
by (*auto simp add: is-L-bisimulation-def FL-bisimilar-def*)
moreover have $\forall P Q. P \sim_{\cdot[F]} Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha P'. bn \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow \langle f \rangle P \rightarrow \langle \alpha, P \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q \rangle \wedge P' \sim_{\cdot[L(\alpha, F, f)]} Q'))$ (*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 $P \sim_{\cdot[F]} Q$ and $f \in_{fs} F$ and $bn \alpha \#* (\langle f \rangle Q, F, f)$ and $\langle f \rangle P \rightarrow \langle \alpha, P \rangle$

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*: $\text{supp } (\langle \alpha, P' \rangle, (\langle f \rangle Q, F, f)) \#* p$

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*: $\text{supp } \langle \alpha, P' \rangle \#* p$ **and** *4*: $\text{supp } (\langle f \rangle Q, F, f) \#* p$

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-egvt*)
from *fresh* **and** 4 **have** $\text{bn } (p \cdot \alpha) \#^* (\langle f \rangle Q, F, f)$
by (*metis bn-egvt 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** $\text{supp } (\langle f \rangle Q) \#^* p$
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-egvt'*)
moreover from 6 **and** 8 **have** $?FL\text{-bisim } (L (p \cdot \alpha, F, f)) (p \cdot P') pQ'$
by (*metis relcompp.relcompI*)
then have $?FL\text{-bisim } (-p \cdot L (p \cdot \alpha, F, f)) (-p \cdot p \cdot P') (-p \cdot pQ')$
using *FL-bisimilar-egvt* **by** *blast*
then have $?FL\text{-bisim } (L (\alpha, -p \cdot F, -p \cdot f)) P' (-p \cdot pQ')$
by (*simp add: L-egvt'*)
then have $?FL\text{-bisim } (L (\alpha, F, f)) P' (-p \cdot pQ')$
using 4 **by** (*metis fresh-star-Un permute-minus-cancel(2) supp-Pair supp-perm-eq*)
ultimately show $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge ?FL\text{-bisim } (L (\alpha, F, f)) P' Q'$
by *metis*
qed
ultimately have *is-L-bisimulation ?FL-bisim*
unfolding *is-L-bisimulation-def* **by** *metis*
moreover have $?FL\text{-bisim } F P R$
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

14 Infinitary Formulas With Effects

14.1 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 ∀ 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 $Tree :: (type, pt, pt, pt) pt$
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

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 $Tree-wf-eqvt-aux$:

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)

```

14.2 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

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

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 $Tree\text{-wf}_p :: (\text{'idx}, \text{'pred}::pt, \text{'act}::bn, \text{'eff}::fs)\ Tree_p\ rel$ **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** (*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 (erased, lifting) Pair-inject Tree-wf_p.abs-eq prod-fun-imageE*)

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

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

Here it comes ...

function (*sequential*)
alpha-Tree :: $(\text{'idx}, \text{'pred}::\text{pt}, \text{'act}::\text{bn}, \text{'eff}::\text{fs}) \text{Tree} \Rightarrow (\text{'idx}, \text{'pred}, \text{'act}, \text{'eff}) \text{Tree} \Rightarrow \text{bool}$ (**infix** $=_\alpha$ 50) **where**
— $(=_\alpha)$
alpha-tConj: $t\text{Conj } tset1 =_\alpha t\text{Conj } tset2 \iff \text{rel-bset } \text{alpha-Tree } tset1\ tset2$
| *alpha-tNot*: $t\text{Not } t1 =_\alpha t\text{Not } t2 \iff t1 =_\alpha t2$
| *alpha-tPred*: $t\text{Pred } f1\ \varphi1 =_\alpha t\text{Pred } f2\ \varphi2 \iff f1 = f2 \wedge \varphi1 = \varphi2$
— the action may have binding names
| *alpha-tAct*: $t\text{Act } f1\ \alpha1\ t1 =_\alpha t\text{Act } f2\ \alpha2\ t2 \iff$
 $f1 = f2 \wedge (\exists p. (\text{bn } \alpha1, t1) \approx_{\text{set}} \text{alpha-Tree } (\text{supp-rel } \text{alpha-Tree})\ p\ (\text{bn } \alpha2, t2))$
 $\wedge (\text{bn } \alpha1, \alpha1) \approx_{\text{set}} ((=))\ \text{supp } p\ (\text{bn } \alpha2, \alpha2))$
| *alpha-other*: $- =_\alpha - \iff \text{False}$
— 254 subgoals (!)
by *pat-completeness auto*
termination
proof
let $?R = \text{inv-image } (\text{max-ext } \text{Tree-wf}_p)\ \text{alpha-Tree-termination}$
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, 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 $\bigwedge a\ b. (a \rightleftharpoons b) \cdot t =_{\alpha} t \iff p \cdot (a \rightleftharpoons b) \cdot t =_{\alpha} p \cdot t$
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) 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 =α y and y =α z
  then show x =α 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 (=α) tset-x tset-z
          unfolding rel-bset.rep-eq rel-set-def Ball-def Bex-def
        proof
          show  $\forall x'. x' \in \text{set-bset } tset-x \longrightarrow (\exists z'. z' \in \text{set-bset } tset-z \wedge x' =_{\alpha} z')$ 
          proof (rule allI, rule impI)
            fix x' assume 1: x' ∈ set-bset tset-x
            then obtain y' where 2: y' ∈ set-bset tset-y and 3: x' =α y'
              by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
            from 2 obtain z' where 4: z' ∈ set-bset tset-z and 5: y' =α z'
              by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1)
            from 1 2 3 5 have x' =α z'
              by (rule tConj.IH)
            with 4 show  $\exists z'. z' \in \text{set-bset } tset-z \wedge x' =_{\alpha} z'$ 
              by auto
          qed
        next
          show  $\forall z'. z' \in \text{set-bset } tset-z \longrightarrow (\exists x'. x' \in \text{set-bset } tset-x \wedge x' =_{\alpha} z')$ 
          proof (rule allI, rule impI)
            fix z' assume 1: z' ∈ set-bset tset-z
            then obtain y' where 2: y' ∈ set-bset tset-y and 3: y' =α z'
              by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prem1)
            from 2 obtain x' where 4: x' ∈ set-bset tset-x and 5: x' =α y'
              by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
            from 4 2 5 3 have x' =α z'
              by (rule tConj.IH)
            with 4 show  $\exists x'. x' \in \text{set-bset } tset-x \wedge x' =_{\alpha} z'$ 
              by auto
          qed
        qed
      with z show tConj tset-x =α 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 α1 t1 f2 α2 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 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)  $\ddagger^*$  (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:  $\bigwedge x y.$  rel-bset (=  $\alpha$ ) x y  $\longleftrightarrow$  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  $\Rightarrow$  b)  $\cdot$  tAct f1  $\alpha$ 1 t1 = $_{\alpha}$  tAct f1  $\alpha$ 1 t1  $\longleftrightarrow$  (a  $\Rightarrow$  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 $\text{supp-rel } (=_{\alpha}) (tAct f \alpha t) = \text{supp } f \cup (\text{supp } \alpha \cup \text{supp-rel } (=_{\alpha}) t - bn \alpha)$

proof

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

proof

fix *x*

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

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** $x2: x \notin bn \alpha$

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** $b2: b \notin \text{supp } \alpha - bn \alpha$

by *simp+*

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

```

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

```

then have *: *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 *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 *infinite* $\{b. (x \equiv b) \cdot f \neq f\} \vee$ *infinite* $\{b. (x \equiv b) \cdot \alpha \neq \alpha\} \vee$
infinite $\{b. \neg (x \equiv b) \cdot t =_{\alpha} t\}$
by (*metis (mono-tags) finite-Collect-disjI*)
then have $x \in supp f \cup supp \alpha \cup supp-rel (=_{\alpha}) t$
by (*simp add: supp-def supp-rel-def*)
moreover
{
assume **: $x \in bn \alpha \wedge x \notin supp f$
from * **obtain** b **where** $b0: \neg (x \equiv b) \cdot tAct f \alpha t =_{\alpha} tAct f \alpha t$ **and** $b1: b \notin supp f$ **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-eqt Diff-iff alpha-Tree-eqt' alpha-Tree-eqt-aux bn-eqt 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-eqt*)
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, opaque-lifting) Diff-eqt Diff-iff bn-eqt supp-eqt 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 $\exists p. (bn ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot t) \approx_{set} (=_{\alpha}) supp-rel (=_{\alpha}) p (bn \alpha, t) \wedge$
 $(bn ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot \alpha) \approx_{set} (=) supp p (bn \alpha, \alpha)$
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_{α}) = t_{α}
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*]: $p \cdot rep\text{-Tree}_{\alpha}\ t_{\alpha} =_{\alpha} rep\text{-Tree}_{\alpha}\ (p \cdot t_{\alpha})$

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

14.3 Constructors for trees modulo α -equivalence

The constructors are lifted from trees.

lift-definition $Conj_\alpha :: ('idx, 'pred, 'act, 'eff) Tree_\alpha \text{ set}['idx] \Rightarrow ('idx, 'pred::pt, 'act::bn, 'eff::fs) Tree_\alpha$ **is**
 $tConj$
by *simp*

lemma $map\text{-}bset\text{-}abs\text{-}rep\text{-}Tree_\alpha: map\text{-}bset\ abs\text{-}Tree_\alpha (map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha) = tset_\alpha$
by (*metis (full-types) Quotient-Tree $_\alpha$ Quotient-abs-rep bset-lifting.bset-quot-map*)

lemma $Conj_\alpha\text{-}def': Conj_\alpha\ tset_\alpha = abs\text{-}Tree_\alpha (tConj (map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha))$
by (*metis Conj $_\alpha$.abs-eq map-bset-abs-rep-Tree $_\alpha$*)

lift-definition $Not_\alpha :: ('idx, 'pred, 'act, 'eff) Tree_\alpha \Rightarrow ('idx, 'pred::pt, 'act::bn, 'eff::fs) Tree_\alpha$ **is**
 $tNot$
by *simp*

lift-definition $Pred_\alpha :: 'eff \Rightarrow 'pred \Rightarrow ('idx, 'pred::pt, 'act::bn, 'eff::fs) Tree_\alpha$ **is**
 $tPred$

.

lift-definition $Act_\alpha :: 'eff \Rightarrow 'act \Rightarrow ('idx, 'pred, 'act, 'eff) Tree_\alpha \Rightarrow ('idx, 'pred::pt, 'act::bn, 'eff::fs) Tree_\alpha$ **is**
 $tAct$
by (*fact alpha-Tree-tAct*)

The lifted constructors are equivariant.

lemma $Conj_\alpha\text{-}eqvt [eqvt, simp]: p \cdot Conj_\alpha\ tset_\alpha = Conj_\alpha (p \cdot tset_\alpha)$

proof –

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

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

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

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

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

The lifted constructors are injective (except for Act_α).

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

lemma $\text{Not}_\alpha\text{-eq-iff } [\text{simp}]: \text{Not}_\alpha t1_\alpha = \text{Not}_\alpha t2_\alpha \longleftrightarrow t1_\alpha = t2_\alpha$
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 Not_α.abs-eq Tree_α.abs-eq-iff Tree_α-abs-rep*)
 then have *rep-Tree_α t1_α =_α rep-Tree_α t2_α*
 using *alpha-Tree.cases* by *auto*
 then show *t1_α = t2_α*
 by (*metis Tree_α.abs-eq-iff Tree_α-abs-rep*)
 next
 assume *t1_α = t2_α* then show *Not_α t1_α = Not_α t2_α*
 by *simp*
 qed

lemma *Pred_α-eq-iff* [*simp*]: *Pred_α f1 φ1 = Pred_α f2 φ2 ↔ f1 = f2 ∧ φ1 = φ2*
proof
 assume *Pred_α f1 φ1 = Pred_α f2 φ2*
 then have (*tPred f1 φ1 :: ('e, 'b, 'f, 'd) Tree*) =_α *tPred f2 φ2* — note the
 unrelated type
 by (*metis Pred_α.abs-eq Tree_α.abs-eq-iff*)
 then show *f1 = f2 ∧ φ1 = φ2*
 using *alpha-Tree.cases* by *auto*
 next
 assume *f1 = f2 ∧ φ1 = φ2* then show *Pred_α f1 φ1 = Pred_α f2 φ2*
 by *simp*
 qed

lemma *Act_α-eq-iff*: *Act_α f1 α1 t1 = Act_α f2 α2 t2 ↔ tAct f1 α1 (rep-Tree_α t1) =_α tAct f2 α2 (rep-Tree_α t2)*
 by (*metis Act_α.abs-eq Tree_α.abs-eq-iff Tree_α-abs-rep*)

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

lemma *Tree_α-free* [*simp*]:
 shows *Conj_α tset_α ≠ Not_α t_α*
 and *Conj_α tset_α ≠ Pred_α f φ*
 and *Conj_α tset_α ≠ Act_α f α t_α*
 and *Not_α t_α ≠ Pred_α f φ*
 and *Not_α t1_α ≠ Act_α f α t2_α*
 and *Pred_α f1 φ ≠ Act_α f2 α t_α*
 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 *supp-alpha-supp-rel*: *supp t_α = supp-rel (=α) (rep-Tree_α t_α)*
unfolding *supp-def supp-rel-def* by (*metis (mono-tags, lifting) Collect-cong Tree_α.abs-eq-iff Tree_α-abs-rep alpha-Tree-permute-rep-commute*)

lemma *supp-Conj_α* [*simp*]: *supp (Conj_α tset_α) = supp tset_α*
unfolding *supp-def* by *simp*

lemma *supp-Not_α* [*simp*]: *supp (Not_α t_α) = supp t_α*
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*)

14.4 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 $?tset_\alpha = \text{map-bset } \text{abs-Tree}_\alpha tset$
have $\text{abs-Tree}_\alpha (tConj tset) = \text{Conj}_\alpha ?tset_\alpha$
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.

14.5 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  $\varphi$ )
| Act $\alpha$ : hereditarily-fs t $\alpha$   $\Longrightarrow$  hereditarily-fs (Act $\alpha$  f  $\alpha$  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, opaque-lifting) 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  $\alpha$  t $\alpha$  = Act $\alpha$  f'  $\alpha'$  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)

```

14.6 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  $t_\alpha$ 
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  $p \cdot Abs\text{-formula } t_\alpha = Abs\text{-formula } (p \cdot t_\alpha)$ 
      using assms by (metis Abs-formula-eqvt)
    moreover have hereditarily-fs  $(p \cdot t_\alpha)$ 
      using assms by (metis hereditarily-fs-eqvt)
    ultimately have  $p \cdot Abs\text{-formula } t_\alpha = Abs\text{-formula } t_\alpha \longleftrightarrow p \cdot t_\alpha = t_\alpha$ 
      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]:  $supp (Rep\text{-formula } x) = supp x$ 
by (metis Rep-formula' Rep-formula-inverse supp-Abs-formula)

```

```

lemma supp-map-bset-Rep-formula [simp]:  $supp (map\text{-bset } Rep\text{-formula } xset) =$ 
 $supp xset$ 

```

```

proof
  have eqvt (map-bset Rep-formula)
    unfolding eqvt-def by (simp add: ext)
  then show  $supp (map\text{-bset } Rep\text{-formula } xset) \subseteq supp xset$ 
    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  $\bigwedge x y. x \neq y \implies map\text{-bset } Rep\text{-formula } x \neq map\text{-bset } Rep\text{-formula}$ 
 $y$ 
      by (metis inj-eq)
    then have  $\{b. (a \rightleftharpoons b) \cdot xset \neq xset\} \subseteq \{b. (a \rightleftharpoons b) \cdot map\text{-bset } Rep\text{-formula}$ 
 $xset \neq map\text{-bset } Rep\text{-formula } xset\}$  (is ?S  $\subseteq$  ?T)
      by auto
    then have infinite ?S  $\implies$  infinite ?T
      by (metis infinite-super)
  }
  then show  $supp xset \subseteq supp (map\text{-bset } Rep\text{-formula } xset)$ 
    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)

```

14.7 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**
 $Conj\ xset = Abs\ formula\ (Conj_\alpha\ (map\ bset\ Rep\ formula\ xset))$

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** $t_\alpha \in set\ bset\ (map\ bset\ Rep\ formula\ xset)$
then show $hereditarily\ fs\ t_\alpha$
by (*auto simp add: bset.set-map*)
qed

lemma $Conj\ rep\ eq$:
assumes $finite\ (supp\ xset)$
shows $Rep\ formula\ (Conj\ xset) = Conj_\alpha\ (map\ bset\ Rep\ formula\ xset)$
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 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*]: $p \cdot \text{Not } x = \text{Not } (p \cdot x)$
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, opaque-lifting) 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*: $\text{Act } f1 \ \alpha1 \ x1 = \text{Act } f2 \ \alpha2 \ x2 \iff$
 $f1 = f2 \wedge (\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2)$

(is ?l \longleftrightarrow ?r)

proof

assume *: ?l

then have f1 = f2

by (metis Act-eq-iff Act $_{\alpha}$ -eq-iff alpha-tAct)

moreover from * **obtain** p **where** alpha: (bn α 1, rep-Tree $_{\alpha}$ (Rep-formula x1)) $\approx_{set} (=_{\alpha})$ (supp-rel (= $_{\alpha}$)) p (bn α 2, rep-Tree $_{\alpha}$ (Rep-formula x2)) **and** eq: (bn α 1, α 1) $\approx_{set} (=)$ supp p (bn α 2, α 2)

by (metis Act-eq-iff Act $_{\alpha}$ -eq-iff alpha-tAct)

from alpha **have** supp x1 - bn α 1 = supp x2 - bn α 2

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

moreover from alpha **have** (supp x1 - bn α 1) $\#^* p$

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 $_{\alpha}$.abs-eq-iff Tree $_{\alpha}$ -abs-rep alpha-Tree-permute-rep-commute alpha-set.simps)

moreover from eq **have** supp α 1 - bn α 1 = supp α 2 - bn α 2

by (metis alpha-set.simps)

moreover from eq **have** (supp α 1 - bn α 1) $\#^* p$

by (metis alpha-set.simps)

moreover from eq **have** p \cdot α 1 = α 2

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: supp x1 - bn α 1 = supp x2 - bn α 2 **and** 2: (supp x1 - bn α 1) $\#^* p$ **and** 3: p \cdot x1 = x2 **and** 4: supp α 1 - bn α 1 = supp α 2 - bn α 2 **and** 5: (supp α 1 - bn α 1) $\#^* p$ **and** 6: p \cdot α 1 = α 2

by metis

from 1 2 3 6 **have** (bn α 1, rep-Tree $_{\alpha}$ (Rep-formula x1)) $\approx_{set} (=_{\alpha})$ (supp-rel (= $_{\alpha}$)) p (bn α 2, rep-Tree $_{\alpha}$ (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** (bn α 1, α 1) $\approx_{set} (=)$ supp p (bn α 2, α 2)

by (simp add: alpha-set.simps bn-eqvt)

ultimately show Act f1 α 1 x1 = Act f2 α 2 x2

by (metis Act-eq-iff Act $_{\alpha}$ -eq-iff alpha-tAct)

qed

lemma Act-eq-iff-perm-renaming: Act f1 α 1 x1 = Act f2 α 2 x2 \longleftrightarrow

 f1 = f2 \wedge (\exists p. supp x1 - bn α 1 = supp x2 - bn α 2 \wedge (supp x1 - bn α 1) $\#^* p \wedge p \cdot$ x1 = x2 \wedge supp α 1 - bn α 1 = supp α 2 - bn α 2 \wedge (supp α 1 - bn α 1) $\#^* p \wedge p \cdot$ α 1 = α 2 \wedge supp p \subseteq bn α 1 \cup p \cdot bn α 1)

(is ?l \longleftrightarrow ?r)

proof

assume $?l$ **then have** $f1 = f2$
by (*metis Act-eq-iff-perm*)
moreover from $\langle ?l \rangle$ **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-eqt*)
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 $\text{Pred } f1 \varphi \neq \text{Act } f2 \alpha x$

proof –

show $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Not } x$
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

```

14.8 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)  $\implies$  ( $\bigwedge x. x \in \text{set-bset } xset \implies \text{is-FL-formula } F x$ )  $\implies$ 
    is-FL-formula F (Conj xset)
| Not: is-FL-formula F x  $\implies$  is-FL-formula F (Not x)
| Pred:  $f \in_{f_s} F \implies \text{is-FL-formula } F (\text{Pred } f \varphi)$ 
| Act:  $f \in_{f_s} F \implies \text{bn } \alpha \ \sharp^* (F,f) \implies \text{is-FL-formula } (L (\alpha,F,f)) x \implies \text{is-FL-formula } F (\text{Act } f \alpha x)$ 

```

```

abbreviation in-A :: ('idx,'pred,'act,'effect) formula  $\implies$  'effect first  $\implies$  bool
  (-  $\in$   $\mathcal{A}[-]$  [51,0] 50) where
   $x \in \mathcal{A}[F] \equiv \text{is-FL-formula } F x$ 

```

```

declare is-FL-formula.induct [case-names Conj Not Pred Act, induct type: formula]

```

```

lemma is-FL-formula-eqvt [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:  $\bigwedge x. x \in \text{set-bset } xset \implies p \cdot x \in \mathcal{A}[p \cdot F]$ 

```

```

  from 1 have finite (supp (p  $\cdot$  xset))

```

```

    by (metis permute-finite supp-eqvt)

```

```

  moreover from 2 have  $\bigwedge x. x \in \text{set-bset } (p \cdot xset) \implies x \in \mathcal{A}[p \cdot F]$ 

```

```

    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 · x ∈  $\mathcal{A}[p \cdot F]$ 
then show p · Not x ∈  $\mathcal{A}[p \cdot F]$ 
  by (simp add: Not)
next
fix f and F :: 'effect first and  $\varphi$ 
assume f ∈fs F
then show p · Pred f  $\varphi$  ∈  $\mathcal{A}[p \cdot F]$ 
  by (simp add: Pred)
next
fix f F  $\alpha$  and x :: ('a, 'pred, 'act, 'effect) formula
assume f ∈fs F and bn  $\alpha$   $\#^*$  (F,f) and p · x ∈  $\mathcal{A}[p \cdot L(\alpha, F, f)]$ 
then show p · Act f  $\alpha$  x ∈  $\mathcal{A}[p \cdot F]$ 
  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

```

14.9 Induction over infinitary formulas

14.10 Strong induction over infinitary formulas

```

end
theory FL-Validity
imports
  FL-Transition-System
  FL-Formula
begin

```

15 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 ($\alpha\text{-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$)
proof ($\text{rule } wf\text{-relcomp-compatible}$)
show wf ($\text{hull-rel } O \text{ Tree-wf}$)
by ($\text{metis Tree-wf-eqt' wf-Tree-wf wf-hull-rel-relcomp}$)
next
show ($\text{hull-rel } O \text{ Tree-wf}$) O $\alpha\text{-Tree-rel} \subseteq \alpha\text{-Tree-rel } O$ ($\text{hull-rel } O \text{ Tree-wf}$)
proof
fix $x :: ('e, 'f, 'g, 'h) \text{ Tree} \times ('e, 'f, 'g, 'h) \text{ Tree}$
assume $x \in (\text{hull-rel } O \text{ Tree-wf}) O \alpha\text{-Tree-rel}$
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** $3: (x_3, x_4) \in \alpha\text{-Tree-rel}$
by auto
from 2 **have** $(x_1, x_4) \in \alpha\text{-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$
using 1 **and** 3 **proof** ($\text{induct rule: Tree-wf.induct}$)
 $- t\text{Conj}$
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** $***$: $(t\text{Conj } tset,$
 $x_4) \in \alpha\text{-Tree-rel}$
from $**$ **obtain** p **where** $x_1: x_1 = p \cdot t$
using hull-rel.cases **by** blast
from $***$ **have** $t\text{Conj } tset =_{\alpha} x_4$
by ($\text{rule } \alpha\text{-Tree-relE}$)
then obtain $tset'$ **where** $x_4: x_4 = t\text{Conj } tset'$ **and** $\text{rel-bset } (=_{\alpha}) \text{ } tset \text{ } tset'$
by ($\text{cases } x_4$) simp-all
with $*$ **obtain** t' **where** $t': t' \in \text{set-bset } tset'$ **and** $t =_{\alpha} t'$
by ($\text{metis rel-bset.rep-eq rel-set-def}$)
with x_1 **have** $(x_1, p \cdot t') \in \alpha\text{-Tree-rel}$
by ($\text{metis Tree}_{\alpha}.\text{abs-eq-iff } \alpha\text{-Tree-relI permute-Tree}_{\alpha}.\text{abs-eq}$)
moreover have $(p \cdot t', t') \in \text{hull-rel}$
by ($\text{rule hull-rel.intros}$)
moreover from x_4 **and** t' **have** $(t', x_4) \in \text{Tree-wf}$
by ($\text{simp add: Tree-wf.intros(1)}$)
ultimately show $(x_1, x_4) \in \alpha\text{-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$
by auto
next
 $- t\text{Not}$
fix t
assume $*$: $(x_1, t) \in \text{hull-rel}$ **and** $**$: $(t\text{Not } t, x_4) \in \alpha\text{-Tree-rel}$
from $*$ **obtain** p **where** $x_1: x_1 = p \cdot t$
using hull-rel.cases **by** blast
from $**$ **have** $t\text{Not } t =_{\alpha} x_4$
by ($\text{rule } \alpha\text{-Tree-relE}$)
then obtain t' **where** $x_4: x_4 = t\text{Not } t'$ **and** $t =_{\alpha} t'$
by ($\text{cases } x_4$) simp-all
with x_1 **have** $(x_1, p \cdot t') \in \alpha\text{-Tree-rel}$
by ($\text{metis Tree}_{\alpha}.\text{abs-eq-iff } \alpha\text{-Tree-relI permute-Tree}_{\alpha}.\text{abs-eq } x_1$)
moreover have $(p \cdot t', t') \in \text{hull-rel}$
by ($\text{rule hull-rel.intros}$)
moreover from x_4 **have** $(t', x_4) \in \text{Tree-wf}$

```

    using Tree-wf.intros(2) by blast
  ultimately show  $(x1, x4) \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$ 
    by auto
next
  — tAct
  fix  $f \alpha t$ 
  assume *:  $(x1, t) \in \text{hull-rel}$  and **:  $(tAct f \alpha t, x4) \in \text{alpha-Tree-rel}$ 
  from * obtain  $p$  where  $x1: x1 = p \cdot t$ 
    using hull-rel.cases by blast
  from ** have  $tAct f \alpha t =_{\alpha} x4$ 
    by (rule alpha-Tree-relE)
  then obtain  $q t'$  where  $x4: x4 = tAct f (q \cdot \alpha) t'$  and  $q \cdot t =_{\alpha} t'$ 
    by (cases x4) (auto simp add: alpha-set)
  with  $x1$  have  $(x1, p \cdot -q \cdot t') \in \text{alpha-Tree-rel}$ 
    by (metis Tree_{\alpha}.abs-eq-iff alpha-Tree-relI permute-Tree_{\alpha}.abs-eq per-
mute-minus-cancel(1))
  moreover have  $(p \cdot -q \cdot t', t') \in \text{hull-rel}$ 
    by (metis hull-rel.simps permute-plus)
  moreover from  $x4$  have  $(t', x4) \in \text{Tree-wf}$ 
    by (simp add: Tree-wf.intros(3))
  ultimately show  $(x1, x4) \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$ 
    by auto
  qed
  with  $x$  show  $x \in \text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf}$ 
    by simp
  qed
qed

```

```

lemma alpha-Tree-rel-relcomp-trivialI [simp]:
  assumes  $(x, y) \in R$ 
  shows  $(x, y) \in \text{alpha-Tree-rel } O R$ 
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  $(x', y) \in R$ 
  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)

```

15.1 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)  $\longleftrightarrow$  ( $\forall t \in \text{set-bset } tset. \text{FL-valid-Tree } P t$ )
| 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)  $\longleftrightarrow$  ( $\exists \alpha' t' P'. \text{tAct } f \alpha t =_{\alpha} \text{tAct } f \alpha' t' \wedge \langle f \rangle P$ 
 $\rightarrow \langle \alpha', P' \rangle \wedge \text{FL-valid-Tree } P' t'$ )
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-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)
  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-eqt 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-eqt' permute-minus-cancel(2))
satisfies-eqt)
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-eqt')
  moreover from * have p · tAct f α t =α tAct (p · f) (p · α') (p · t')
  by (metis alpha-Tree-eqt 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-eqt permute-Tree.simps(4) permute-minus-cancel(2))
  moreover from * have ⟨f⟩P → ⟨-p · α', -p · P'⟩
  by (metis permute-minus-cancel(2) transition-eqt')
  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-eqt* [*eqt*]:

assumes *FL-valid-Tree* P t **shows** *FL-valid-Tree* (p · P) (p · t)
using *assms* **by** (*metis FL-valid-Tree-eqt'*)

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

```

15.2 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 $FL\text{-valid-Tree}_\alpha\text{-eqvt}$ [*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 $FL\text{-valid-Tree}_\alpha\text{-Conj}_\alpha$ [*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 $_\alpha$.rep-abs* *alpha-Tree-FL-valid-Tree*)

then show *?thesis*

by (*simp* *add: FL-valid-Tree $_\alpha$ -def* *Conj $_\alpha$ -def* *map-bset.rep-eq*)

qed

lemma *FL-valid-Tree $_{\alpha}$ -Not $_{\alpha}$* [simp]: $FL\text{-valid-Tree}_{\alpha} P (Not_{\alpha} t_{\alpha}) \longleftrightarrow \neg FL\text{-valid-Tree}_{\alpha} P t_{\alpha}$
 by transfer simp

lemma *FL-valid-Tree $_{\alpha}$ -Pred $_{\alpha}$* [simp]: $FL\text{-valid-Tree}_{\alpha} P (Pred_{\alpha} f \varphi) \longleftrightarrow \langle f \rangle P \vdash \varphi$
 by transfer simp

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

proof

assume $FL\text{-valid-Tree}_{\alpha} P (Act_{\alpha} f \alpha t_{\alpha})$

moreover have $Act_{\alpha} f \alpha t_{\alpha} = abs\text{-Tree}_{\alpha} (tAct f \alpha (rep\text{-Tree}_{\alpha} t_{\alpha}))$

by (metis *Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-rep*)

ultimately show $\exists \alpha' t_{\alpha}' P'. Act_{\alpha} f \alpha t_{\alpha} = Act_{\alpha} f \alpha' t_{\alpha}' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge FL\text{-valid-Tree}_{\alpha} P' t_{\alpha}'$

by (metis *Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-eq-iff FL-valid-Tree.simps(4) FL-valid-Tree $_{\alpha}$.abs-eq*)

next

assume $\exists \alpha' t_{\alpha}' P'. Act_{\alpha} f \alpha t_{\alpha} = Act_{\alpha} f \alpha' t_{\alpha}' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge FL\text{-valid-Tree}_{\alpha} P' t_{\alpha}'$

moreover have $\bigwedge \alpha' t_{\alpha}'. Act_{\alpha} f \alpha' t_{\alpha}' = abs\text{-Tree}_{\alpha} (tAct f \alpha' (rep\text{-Tree}_{\alpha} t_{\alpha}'))$

by (metis *Act $_{\alpha}$.abs-eq Tree $_{\alpha}$.abs-rep*)

ultimately show $FL\text{-valid-Tree}_{\alpha} P (Act_{\alpha} f \alpha t_{\alpha})$

by (metis *Tree $_{\alpha}$.abs-eq-iff FL-valid-Tree.simps(4) FL-valid-Tree $_{\alpha}$.abs-eq FL-valid-Tree $_{\alpha}$.rep-eq*)

qed

15.3 Validity for infinitary formulas

lift-definition *FL-valid* :: 'state \Rightarrow ('idx,'pred,'act,'effect) formula \Rightarrow bool (infix \models 70) is
 $FL\text{-valid-Tree}_{\alpha}$
 .

lemma *FL-valid-eqvt* [eqvt]:

assumes $P \models x$ shows $(p \cdot P) \models (p \cdot x)$

using *assms* by transfer (metis *FL-valid-Tree $_{\alpha}$ -eqvt*)

lemma *FL-valid-Conj* [simp]:

assumes *finite* (*supp* *xset*)

shows $P \models Conj\ xset \longleftrightarrow (\forall x \in set\text{-bset}\ xset. P \models x)$

using *assms* by (simp add: *FL-valid-def Conj-def map-bset.rep-eq*)

lemma *FL-valid-Not* [simp]: $P \models Not\ x \longleftrightarrow \neg P \models x$

by transfer simp

lemma *FL-valid-Pred* [simp]: $P \models Pred\ f\ \varphi \longleftrightarrow \langle f \rangle P \vdash \varphi$

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* ($\text{Act}_\alpha f \alpha$ (*Rep-formula* x))) = $\text{Act}_\alpha f \alpha$ (*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 $\langle \text{finite} (\text{supp } X) \rangle$

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 $\text{bn } \alpha' \#* (\text{supp } x' - \text{bn } \alpha', \text{supp } \alpha' - \text{bn } \alpha', \langle \alpha', P' \rangle)$

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*: $\text{supp} (\text{supp } x' - \text{bn } \alpha', \text{supp } \alpha' - \text{bn } \alpha', \langle \alpha', P' \rangle) \#* p$

by (*metis at-set-avoiding2*)

from *fresh-p* **have** $\text{supp} (\text{supp } x' - \text{bn } \alpha') \#* p$ **and** $\text{supp} (\text{supp } \alpha' - \text{bn } \alpha') \#* p$ **and** *1*: $\text{supp } \langle \alpha', P' \rangle \#* p$

by (*meson fresh-Pair fresh-def fresh-star-def*)**+**

then have *2*: $(\text{supp } x' - \text{bn } \alpha') \#* p$ **and** *3*: $(\text{supp } \alpha' - \text{bn } \alpha') \#* p$

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-eq 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, opaque-lifting) Diff-eqvt atom-set-perm-eq 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-eq-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-eq*)

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

using *eq and trans and valid and fresh-X* by (*metis bn-eqvt FL-valid-eqvt*)

next

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

then show $P \models \text{Act } f \alpha x$ by (*metis FL-valid-Act*)

qed

lemma *FL-valid-Act-fresh*:

assumes $\text{bn } \alpha \#^* \langle f \rangle P$

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*: $\text{bn } \alpha' \#^* \langle f \rangle P$

by (*metis FL-valid-Act-strong*)

from eq obtain p **where** $p \cdot \alpha: \alpha' = p \cdot \alpha$ **and** $p \cdot x: x' = p \cdot x$ **and** *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 $(\text{bn } \alpha \cup p \cdot \text{bn } \alpha) \#^* \langle f \rangle P$

using $p \cdot \alpha$ by (*metis bn-eqvt fresh-star-Un*)

then have $\text{supp } p \#^* \langle f \rangle P$

using *supp-p* by (*metis fresh-star-def subset-eq*)

then have $p \cdot P: -p \cdot \langle f \rangle P = \langle f \rangle P$

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

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

```

16 (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  $\vdash$   $\gamma$ 0)
  and transition :: 'state  $\Rightarrow$  ('act::bn, 'state) residual  $\Rightarrow$  bool (infix  $\rightarrow$   $\gamma$ 0)
  and effect-apply :: 'effect::fs  $\Rightarrow$  'state  $\Rightarrow$  'state (( $\cdot$ )- [0,101] 100) +
  assumes card-idx-perm: |UNIV::perm set| < o |UNIV::'idx set|
    and card-idx-state: |UNIV::'state set| < o |UNIV::'idx set|
begin

  definition FL-logically-equivalent :: 'effect first  $\Rightarrow$  'state  $\Rightarrow$  'state  $\Rightarrow$  bool where
    FL-logically-equivalent F P Q  $\equiv$ 
       $\forall x::('idx, 'pred, 'act, 'effect)$  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

```

17 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  $\bigwedge P Q. P \sim \cdot [L (\alpha, F, f)] Q \Longrightarrow P \models x \longleftrightarrow Q \models x$ 

```

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

from $\langle P \sim \cdot [F] Q \rangle$ **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'*: $P' \sim \cdot [L(\alpha', F, f)] Q'$
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-p*: $(\text{supp } x - \text{bn } \alpha) \#* p \wedge (\text{supp } \alpha - \text{bn } \alpha) \#* p$
and *supp-p*: $\text{supp } p \subseteq \text{bn } \alpha \cup p \cdot \text{bn } \alpha$
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-eqvt*)

moreover from *fresh* **and** $p\text{-}\alpha$ **have** $(p \cdot \text{bn } \alpha) \#* (F, f)$
by (*simp add: bn-eqvt fresh-star-Pair*)
with $\langle \text{bn } \alpha \#* (F, f) \rangle$ **and** *supp-p* **have** $\text{supp } (F, f) \#* p$
by (*meson UnE fresh-def fresh-star-def subsetCE*)
then have $\text{supp } F \#* p$ **and** $\text{supp } f \#* p$
by (*simp add: fresh-star-Un supp-Pair*)+

with *bisim'* **and** $p\text{-}\alpha$ **have** $(-p \cdot P') \sim \cdot [L(\alpha, F, f)] (-p \cdot Q')$
by (*metis FL-bisimilar-eqvt L-eqvt' permute-minus-cancel(2) supp-perm-eq*)

ultimately have $-p \cdot Q' \models x$
using $\langle \bigwedge P Q. P \sim \cdot [L(\alpha, F, f)] Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **by** *metis*

with $p\text{-}x$ **have** $Q' \models x'$
by (*metis permute-minus-cancel(1) FL-valid-eqvt*)

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

theorem *FL-bisimilarity-implies-equivalence*: **assumes** $P \sim \cdot [F] Q$ **shows** *FL-logically-equivalent*
 $F P Q$
unfolding *FL-logically-equivalent-def* **proof**
fix $x :: ('idx, 'pred, 'act, 'effect) \text{ formula}$
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

```

18 Logical Equivalence Implies F/L -Bisimilarity

```

context indexed-effect-nominal-ts
begin

```

definition *is-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 *is-distinguishing-formula-eqvt* :

assumes *x distinguishes P from Q* **shows** (*p* \cdot *x*) *distinguishes* (*p* \cdot *P*) *from* (*p* \cdot *Q*)

using *assms* **unfolding** *is-distinguishing-formula-def*

by (*metis permute-minus-cancel(2) FL-valid-eqvt*)

lemma *FL-equivalent-iff-not-distinguished*:

FL-logically-equivalent F P Q $\longleftrightarrow \neg(\exists x. x \in \mathcal{A}[F] \wedge x \text{ distinguishes } P \text{ from } Q)$

by (*meson FL-logically-equivalent-def Not is-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*: *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 $|UNIV :: \text{perm set}| < o |UNIV :: \text{'idx set}|$
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 $?y = Conj (Abs-bset ?B) :: ('idx, 'pred, 'act, 'effect)\ formula$

from *finite-supp-B* and *card-B* and *supp-B-subset-supp-P* have $supp\ ?y \subseteq$
 $supp\ (F,P)$
 by *simp*
 moreover have $?y \in \mathcal{A}[F]$
 proof
 show $finite\ (supp\ (Abs-bset\ ?B :: (\cdot, \cdot, \cdot, \cdot)\ formula\ set['idx]))$
 using *finite-supp-B card-B* by *simp*
 next
 fix x'
 assume $x' \in set-bset\ (Abs-bset\ ?B :: (\cdot, \cdot, \cdot, \cdot)\ formula\ set['idx])$
 then obtain p where $p-x: x' = p \cdot x$ and $fresh-p: supp\ (F,P) \#* p$
 using *card-B* by *auto*
 from *fresh-p* have $p \cdot F = F$
 using *fresh-star-Pair fresh-star-supp-conv perm-supp-eq* by *blast*
 with $\langle x \in \mathcal{A}[F] \rangle$ show $x' \in \mathcal{A}[F]$
 using $p-x$ by (metis *is-FL-formula-eqvt*)
 qed
 moreover have $?y$ distinguishes P from Q
 unfolding *is-distinguishing-formula-def* proof
 from $\langle x$ distinguishes P from $Q \rangle$ show $P \models ?y$
 by (auto *simp add: card-B finite-supp-B*) (metis *is-distinguishing-formula-def*
fresh-star-Un supp-Pair supp-perm-eq FL-valid-eqvt)
 next
 from $\langle x$ distinguishes P from $Q \rangle$ show $\neg Q \models ?y$
 by (auto *simp add: card-B finite-supp-B*) (metis *is-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 $F P Q f \varphi$
 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  $FL\text{-logically-equivalent } F P Q$  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  $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge FL\text{-logically-equivalent } (L (\alpha, F, f)) P' Q'$ 
  proof –
  {
    let  $?Q' = \{Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle\}$ 
    assume  $\forall Q' \in ?Q'. \neg FL\text{-logically-equivalent } (L (\alpha, F, f)) P' Q'$ 
    then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act, 'effect) \text{ formula. } x \in \mathcal{A}[L$ 
 $(\alpha, F, f)] \wedge x \text{ distinguishes } P' \text{ from } Q'$ 
    by (metis FL-equivalent-iff-not-distinguished)
    then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act, 'effect) \text{ formula. } x \in \mathcal{A}[L$ 
 $(\alpha, F, f)] \wedge \text{supp } x \subseteq \text{supp } (L (\alpha, F, f), P') \wedge x \text{ distinguishes } P' \text{ from } Q'$ 
    by (metis FL-distinguished-bounded-support)
    then obtain  $g :: 'state \Rightarrow ('idx, 'pred, 'act, 'effect) \text{ formula}$  where
     $*: \forall Q' \in ?Q'. g Q' \in \mathcal{A}[L (\alpha, F, f)] \wedge \text{supp } (g Q') \subseteq \text{supp } (L (\alpha, F, f), P')$ 
 $\wedge (g Q') \text{ distinguishes } P' \text{ from } Q'$ 
    by metis
    have  $\text{supp } (g \text{ ' } ?Q') \subseteq \text{supp } (L (\alpha, F, f), P')$ 
    by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)
    then have finite-supp-image: finite (supp (g ' ?Q'))
    using finite-supp rev-finite-subset by blast
    have  $|g \text{ ' } ?Q'| \leq o \mid UNIV :: 'state \text{ set}$ 
    by (metis card-of-UNIV card-of-image ordLeq-transitive)
    also have  $\mid UNIV :: 'state \text{ set} \mid < o \mid UNIV :: 'idx \text{ set} \mid$ 
    by (metis card-idx-state)
    also have  $\mid UNIV :: 'idx \text{ set} \mid \leq o \text{ natLeq } + c \mid UNIV :: 'idx \text{ set} \mid$ 
    by (metis Cnotzero-UNIV ordLeq-csum2)
    finally have card-image:  $|g \text{ ' } ?Q'| < o \text{ natLeq } + c \mid UNIV :: 'idx \text{ set} \mid$ .
    let  $?y = \text{Conj } (Abs\text{-bset } (g \text{ ' } ?Q')) :: ('idx, 'pred, 'act, 'effect) \text{ formula}$ 
    have  $\text{Act } f \alpha ?y \in \mathcal{A}[F]$ 
    proof
    from  $\langle f \in_{fs} F \rangle$  show  $f \in_{fs} F$  .
    next
    from  $\langle bn \alpha \#* (\langle f \rangle Q, F, f) \rangle$  show  $bn \alpha \#* (F, f)$ 
    using fresh-star-Pair by blast
    next
    show  $\text{Conj } (Abs\text{-bset } (g \text{ ' } ?Q')) \in \mathcal{A}[L (\alpha, F, f)]$ 
    proof
    show finite (supp (Abs-bset (g ' ?Q') :: (-,-,-) formula set['idx]))
    using finite-supp-image card-image by simp
    next
    fix  $x'$ 
    assume  $x' \in \text{set-bset } (Abs\text{-bset } (g \text{ ' } ?Q') :: (-,-,-) \text{ formula set['idx]})$ 
    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 is-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  $\langle \text{bn } \alpha \ \dagger * (\langle f \rangle Q, F, f) \rangle$  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 is-distinguishing-formula-def by blast
qed
ultimately have False
  by (metis \langle 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

19 L -Transform

19.1 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  $\Rightarrow$  ('a,'b)  $L$ -state  $\Rightarrow$  ('a,'b)  $L$ -state where  
     $p \cdot (AC\ x) = AC\ (p \cdot x)$   
  |  $p \cdot (EF\ x) = EF\ (p \cdot x)$ 
```

```
  instance
```

```
  proof
```

```
    fix  $x :: ('a,'b)\ L$ -state
```

```
    show  $0 \cdot x = x$  by (cases  $x$ , simp-all)
```

```
  next
```

```
    fix  $p\ q$  and  $x :: ('a,'b)\ L$ -state
```

```
    show  $(p + q) \cdot x = p \cdot q \cdot x$  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

19.2 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  $\Rightarrow$  ('a,'b) L-action  $\Rightarrow$  ('a,'b) L-action where  
    p  $\cdot$  (Act  $\alpha$ ) = Act (p  $\cdot$   $\alpha$ )  
  | p  $\cdot$  (Eff f) = Eff (p  $\cdot$  f)
```

```
instance
```

```
proof
```

```
  fix x :: ('a,'b) L-action
```

```
  show 0  $\cdot$  x = x by (cases x, simp-all)
```

```
next
```

```
  fix p q and x :: ('a,'b) L-action
```

```
  show (p + q)  $\cdot$  x = p  $\cdot$  q  $\cdot$  x by (cases x, simp-all)
```

```
qed
```

end

```
declare permute-L-action.simps [eqvt]
```

```
lemma supp-Act [simp]: supp (Act  $\alpha$ ) = supp  $\alpha$   
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  $\Rightarrow$  atom set where
  bn-L-action (Act  $\alpha$ ) = bn  $\alpha$ 
| bn-L-action (Eff -) = {}

```

```

instance

```

```

proof

```

```

  fix p and  $\alpha$  :: ('a,'b) L-action
  show p  $\cdot$  bn  $\alpha$  = bn (p  $\cdot$   $\alpha$ )
  by (cases  $\alpha$ ) (simp add: bn-eqvt, simp)

```

```

next

```

```

  fix  $\alpha$  :: ('a,'b) L-action
  show finite (bn  $\alpha$ )
  by (cases  $\alpha$ ) (simp add: bn-finite, simp)

```

```

qed

```

```

end

```

19.3 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

```

19.4 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-eqt*: **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-eqt'*)
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-eqt'*)
moreover from *fresh* **have** $bn (p \cdot \alpha) \#* (p \cdot F, p \cdot f)$
by (*metis bn-eqt 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-eqt 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 $\exists \alpha P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge \langle \alpha_L, P_L^\wedge \rangle = \langle \text{Act } \alpha, \text{EF } (L (\alpha, F, f), P') \rangle \wedge \text{bn } \alpha \#^* X$
using *assms proof* –
from $\langle \text{AC } (f, F, P) \rightarrow_L \langle \alpha_L, P_L^\wedge \rangle \rangle$ **obtain** $\alpha P'$ **where transition:** $P \rightarrow \langle \alpha, P^\wedge \rangle$
and alpha: $\langle \alpha_L, P_L^\wedge \rangle = \langle \text{Act } \alpha, \text{EF } (L (\alpha, F, f), P') \rangle$ **and fresh:** $\text{bn } \alpha \#^* (F, f)$
by (*metis L-transition.simps(1)*)
let $?Act = \text{Act } \alpha :: ('act, 'effect) \text{ L-action}$ — the type annotation prevents a type that is too polymorphic and doesn't fix 'effect
have *finite* ($\text{bn } \alpha$)
by (*fact bn-finite*)
moreover note (*finite (supp X)*)
moreover have *finite* ($\text{supp } (\langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle, \langle \alpha, P^\wedge \rangle, F, f)$)
by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)
moreover from fresh have $\text{bn } \alpha \#^* (\langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle, \langle \alpha, P^\wedge \rangle, F, f)$
by (*auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
ultimately obtain p **where fresh-X:** $(p \cdot \text{bn } \alpha) \#^* X$ **and** $\text{supp } (\langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle, \langle \alpha, P^\wedge \rangle, F, f) \#^* p$
by (*metis at-set-avoiding2*)
then have $\text{supp } \langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle \#^* p$ **and** $\text{supp } \langle \alpha, P^\wedge \rangle \#^* p$ **and** $\text{supp } (F, f) \#^* p$
by (*metis fresh-star-Un supp-Pair*)+
then have $p \cdot \langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle = \langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle$ **and** $p \cdot \langle \alpha, P^\wedge \rangle = \langle \alpha, P^\wedge \rangle$ **and** $p \cdot (F, f) = (F, f)$
by (*metis supp-perm-eq*)+
then have $\langle \text{Act } (p \cdot \alpha), \text{EF } (L (p \cdot \alpha, F, f), p \cdot P') \rangle = \langle ?Act, \text{EF } (L (\alpha, F, f), P') \rangle$ **and** $\langle p \cdot \alpha, p \cdot P' \rangle = \langle \alpha, P^\wedge \rangle$
using *permute-L-action.simps(1) permute-L-state.simps(2) abs-residual-pair-eqt L-eqt' Pair-eqt* **by** *auto*
then show $\exists \alpha P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge \langle \alpha_L, P_L^\wedge \rangle = \langle \text{Act } \alpha, \text{EF } (L (\alpha, F, f), P') \rangle \wedge \text{bn } \alpha \#^* X$
using *transition and alpha and fresh-X* **by** (*metis bn-eqt*)
qed

lemma *L-transition-AC-fresh:*

assumes $\text{bn } \alpha \#^* (F, f, P)$

shows $\text{AC } (f, F, P) \rightarrow_L \langle \text{Act } \alpha, P_L^\wedge \rangle \longleftrightarrow (\exists P'. P_L' = \text{EF } (L (\alpha, F, f), P') \wedge P \rightarrow \langle \alpha, P^\wedge \rangle)$

proof

assume $\text{AC } (f, F, P) \rightarrow_L \langle \text{Act } \alpha, P_L^\wedge \rangle$

moreover have *finite* ($\text{supp } (F, f, P)$)

by (*fact finite-supp*)

ultimately obtain $\alpha' P'$ **where trans:** $P \rightarrow \langle \alpha', P^\wedge \rangle$ **and eq:** $\langle \text{Act } \alpha :: ('act, 'effect) \text{ L-action}, P_L^\wedge \rangle = \langle \text{Act } \alpha', \text{EF } (L (\alpha', F, f), P') \rangle$ **and fresh:** $\text{bn } \alpha' \#^* (F, f, P)$

using *L-transition-AC-strong* **by** *blast*

from eq obtain p **where** $p \cdot (\text{Act } \alpha :: ('act, 'effect) \text{ L-action}, P_L^\wedge) = (\text{Act } \alpha', \text{EF } (L (\alpha', F, f), P'))$ **and** $\text{supp-}p: \text{supp } p \subseteq \text{bn } (\text{Act } \alpha :: ('act, 'effect) \text{ L-action})$

$\cup p \cdot \text{bn } (\text{Act } \alpha :: ('act, 'effect) \text{ L-action})$
using *residual-eq-iff-perm-renaming* **by** *metis*

from p **have** $p\text{-}\alpha: p \cdot \alpha = \alpha'$ **and** $p\text{-}P_{L'}: p \cdot P_{L'} = EF (L (\alpha', F, f), P')$
by *simp-all*

from *supp-p* **and** $p\text{-}\alpha$ **and** *assms* **and** *fresh* **have** $\text{supp } p \#* (F, f, P)$
by (*simp add: bn-eqvt fresh-star-def*) *blast*

then **have** $p\text{-}F: p \cdot F = F$ **and** $p\text{-}f: p \cdot f = f$ **and** $p\text{-}P: p \cdot P = P$
by (*simp-all add: fresh-star-Pair perm-supp-eq*)

from $p\text{-}P_{L'}$ **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-eqvt'*)

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** $\text{bn } \alpha \#* (F, f)$
by (*simp add: fresh-star-Pair*)

ultimately **show** $AC (f, F, P) \rightarrow_L \langle \text{Act } \alpha, P_{L'} \rangle$
using *L-transition.simps(1)* **by** *blast*

qed

end

19.5 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 $L\text{-transform-Tree} :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) \text{ Tree} \Rightarrow ('idx, 'pred, ('act, 'eff) \text{ L-action}) \text{ Formula.Tree}_\alpha$ **where**
 $L\text{-transform-Tree } (t\text{Conj } tset) = \text{Formula.Conj}_\alpha (\text{map-bset } L\text{-transform-Tree } tset)$

| $L\text{-transform-Tree } (t\text{Not } t) = \text{Formula.Not}_\alpha (L\text{-transform-Tree } t)$
| $L\text{-transform-Tree } (t\text{Pred } f \ \varphi) = \text{Formula.Act}_\alpha (\text{Eff } f) (\text{Formula.Pred}_\alpha \ \varphi)$
| $L\text{-transform-Tree } (t\text{Act } f \ \alpha \ t) = \text{Formula.Act}_\alpha (\text{Eff } f) (\text{Formula.Act}_\alpha (\text{Act } \alpha) (L\text{-transform-Tree } t))$

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 ($t\text{Conj } tset$)

then show ?case

by simp (metis (no-types, opaque-lifting) bset.map-cong0 map-bset-eqt permute-fun-def permute-minus-cancel(1))

qed simp-all

$L\text{-transform-Tree}$ respects α -equivalence.

lemma $\alpha\text{-Tree-}L\text{-transform-Tree}$:

assumes $\alpha\text{-Tree } t1 \ t2$

shows $L\text{-transform-Tree } t1 = L\text{-transform-Tree } t2$

using *assms* **proof** (induction $t1 \ t2$ rule: $\alpha\text{-Tree-induct}'$)

case ($\alpha\text{-tConj } tset1 \ tset2$)

then have $\text{rel-bset } (=) (\text{map-bset } L\text{-transform-Tree } tset1) (\text{map-bset } L\text{-transform-Tree } tset2)$

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 ($\alpha\text{-tAct } f1 \ \alpha1 \ t1 \ f2 \ \alpha2 \ t2$)

from $\langle \alpha\text{-Tree } (FL\text{-Formula.Tree.tAct } f1 \ \alpha1 \ t1) (FL\text{-Formula.Tree.tAct } f2 \ \alpha2 \ t2) \rangle$

obtain p **where** $*$: $(bn \ \alpha1, \ t1) \approx_{\text{set}} \alpha\text{-Tree } (\text{supp-rel } \alpha\text{-Tree}) \ p \ (bn \ \alpha2, \ t2)$

and $**$: $(bn \ \alpha1, \ \alpha1) \approx_{\text{set}} (=) \text{supp } p \ (bn \ \alpha2, \ \alpha2)$ **and** $f1 = f2$

by auto

from $*$ **have** fresh : $(\text{supp-rel } \alpha\text{-Tree } t1 - bn \ \alpha1) \ \#\ast \ p$ **and** α : $\alpha\text{-Tree } (p \cdot t1) \ t2$ **and** eq : $p \cdot bn \ \alpha1 = bn \ \alpha2$

by (auto simp add: $\alpha\text{-set}$)

from $\alpha\text{-tAct.IH}(2)$ **have** $\text{supp-rel } \text{Formula.alpha-Tree } (\text{Formula.rep-Tree}_\alpha (L\text{-transform-Tree } t1)) \subseteq \text{supp-rel } \alpha\text{-Tree } t1$

by (metis (no-types, lifting) infinite-mono $\text{Formula.alpha-Tree-permute-rep-commute}$ $L\text{-transform-Tree-eqt}$ mem-Collect-eq subsetI supp-rel-def)

with fresh **have** fresh' : $(\text{supp-rel } \text{Formula.alpha-Tree } (\text{Formula.rep-Tree}_\alpha (L\text{-transform-Tree } t1)) - bn \ \alpha1) \ \#\ast \ p$

by (meson DiffD1 DiffD2 DiffI fresh-star-def subsetCE)

moreover from α **have** α' : $\text{Formula.alpha-Tree } (p \cdot \text{Formula.rep-Tree}_\alpha (L\text{-transform-Tree } t1)) (\text{Formula.rep-Tree}_\alpha (L\text{-transform-Tree } t2))$

using $\alpha\text{-tAct.IH}(1)$ **by** (metis $\text{Formula.alpha-Tree-permute-rep-commute}$ $L\text{-transform-Tree-eqt}$)

moreover from fresh' α' eq **have** $\text{supp-rel } \text{Formula.alpha-Tree } (\text{Formula.rep-Tree}_\alpha (L\text{-transform-Tree } t1)) - bn \ \alpha1 = \text{supp-rel } \text{Formula.alpha-Tree } (\text{Formula.rep-Tree}_\alpha$

$(L\text{-transform-Tree } t2)) - bn \ \alpha2$
by (*metis (mono-tags) Diff-eqvt Formula.alpha-Tree-eqvt' Formula.alpha-Tree-eqvt-ax Formula.alpha-Tree-suppl-rel atom-set-perm-eq*)
ultimately have $(bn \ \alpha1, Formula.rep\text{-Tree}_\alpha (L\text{-transform-Tree } t1)) \approx_{set} Formula.alpha\text{-Tree} (suppl\text{-rel } Formula.alpha\text{-Tree}) p (bn \ \alpha2, Formula.rep\text{-Tree}_\alpha (L\text{-transform-Tree } t2))$
using eq by (*simp add: alpha-set*)
moreover from ** have $(bn \ \alpha1, Act \ \alpha1) \approx_{set} (=) suppl \ p (bn \ \alpha2, Act \ \alpha2)$
by (*metis (mono-tags, lifting) L-Transform.suppl-Act alpha-set permute-L-action.simps(1)*)
ultimately have $Formula.Act_\alpha (Act \ \alpha1) (L\text{-transform-Tree } t1) = Formula.Act_\alpha (Act \ \alpha2) (L\text{-transform-Tree } t2)$
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 $L\text{-transform-Tree}_\alpha :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) Tree_\alpha \Rightarrow ('idx, 'pred, ('act, 'eff) L\text{-action}) Formula.Tree_\alpha$ **is**
 $L\text{-transform-Tree}$
by (*fact alpha-Tree-L-transform-Tree*)

lemma $L\text{-transform-Tree}_\alpha\text{-eqvt [eqvt]: } p \cdot L\text{-transform-Tree}_\alpha \ t_\alpha = L\text{-transform-Tree}_\alpha (p \cdot t_\alpha)$
by *transfer (simp)*

lemma $L\text{-transform-Tree}_\alpha\text{-Conj}_\alpha [simp]: L\text{-transform-Tree}_\alpha (Conj_\alpha \ tset_\alpha) = Formula.Conj_\alpha (map\text{-bset } L\text{-transform-Tree}_\alpha \ tset_\alpha)$
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 $L\text{-transform-Tree}_\alpha\text{-Not}_\alpha [simp]: L\text{-transform-Tree}_\alpha (Not_\alpha \ t_\alpha) = Formula.Not_\alpha (L\text{-transform-Tree}_\alpha \ t_\alpha)$
by *transfer simp*

lemma $L\text{-transform-Tree}_\alpha\text{-Pred}_\alpha [simp]: L\text{-transform-Tree}_\alpha (Pred_\alpha \ f \ \varphi) = Formula.Act_\alpha (Eff \ f) (Formula.Pred_\alpha \ \varphi)$
by *transfer simp*

lemma $L\text{-transform-Tree}_\alpha\text{-Act}_\alpha [simp]: L\text{-transform-Tree}_\alpha (Act_\alpha \ f \ \alpha \ t_\alpha) = Formula.Act_\alpha (Eff \ f) (Formula.Act_\alpha (Act \ \alpha) (L\text{-transform-Tree}_\alpha \ t_\alpha))$
by *transfer simp*

lemma $finite\text{-suppl-map-bset-}L\text{-transform-Tree}_\alpha [simp]:$
assumes *finite (suppl tset_\alpha)*
shows *finite (suppl (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 $\text{supp} (\text{map-bset } L\text{-transform-Tree}_\alpha) = \{\}$
using $\text{supp-fun-eqvt } \text{supp-fun-app-eqvt}$ **by** blast
then have $\text{supp} (\text{map-bset } L\text{-transform-Tree}_\alpha \text{ tset}_\alpha) \subseteq \text{supp } \text{tset}_\alpha$
using supp-fun-app **by** blast
with assms **show** $\text{finite} (\text{supp} (\text{map-bset } L\text{-transform-Tree}_\alpha \text{ tset}_\alpha))$
by $(\text{metis } \text{finite-subset})$
qed

lemma $L\text{-transform-Tree}_\alpha\text{-preserves-hereditarily-fs}$:
assumes $\text{hereditarily-fs } t_\alpha$
shows $\text{Formula.hereditarilly-fs} (L\text{-transform-Tree}_\alpha \text{ t}_\alpha)$
using assms **proof** $(\text{induct rule: hereditarily-fs.induct})$
case $(\text{Conj}_\alpha \text{ tset}_\alpha)$
then show $?case$
by $(\text{auto intro!: Formula.hereditarilly-fs.Conj}_\alpha) (\text{metis } \text{imageE } \text{map-bset.rep-eq})$
next
case $(\text{Not}_\alpha \text{ t}_\alpha)$
then show $?case$
by $(\text{simp add: Formula.hereditarilly-fs.Not}_\alpha)$
next
case $(\text{Pred}_\alpha \text{ f } \varphi)$
then show $?case$
by $(\text{simp add: Formula.hereditarilly-fs.Act}_\alpha \text{ Formula.hereditarilly-fs.Pred}_\alpha)$
next
case $(\text{Act}_\alpha \text{ t}_\alpha \text{ f } \alpha)$
then show $?case$
by $(\text{simp add: Formula.hereditarilly-fs.Act}_\alpha)$
qed

L -transform for F/L -formulas.

lift-definition $L\text{-transform-formula} :: ('idx, 'pred::fs, 'act::bn, 'eff::fs) \text{ formula} \Rightarrow$
 $('idx, 'pred, ('act, 'eff) L\text{-action}) \text{ Formula.Tree}_\alpha$ **is**
 $L\text{-transform-Tree}_\alpha$
 $.$

lemma $L\text{-transform-formula-eqvt} [\text{eqvt}]: p \cdot L\text{-transform-formula } x = L\text{-transform-formula}$
 $(p \cdot x)$
by $\text{transfer } (\text{simp})$

lemma $L\text{-transform-formula-Conj} [\text{simp}]$:
assumes $\text{finite} (\text{supp } \text{xset})$
shows $L\text{-transform-formula} (\text{Conj } \text{xset}) = \text{Formula.Conj}_\alpha (\text{map-bset } L\text{-transform-formula}$
 $\text{xset})$
using assms **by** $(\text{simp add: Conj-def } L\text{-transform-formula-def } \text{bset.map-comp}$
 $\text{map-fun-def})$

lemma $L\text{-transform-formula-Not} [\text{simp}]: L\text{-transform-formula} (\text{Not } x) = \text{Formula.Not}_\alpha$
 $(L\text{-transform-formula } x)$
by $\text{transfer } \text{simp}$

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 α)

lemma *L-transform-Act* [*simp*]: *L-transform* (*FL-Formula.Act* $f \alpha x$) = *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 α*)

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 ($x :: ('idx, 'pred, 'act, 'effect) formula$) $\in \mathcal{A}[F]$
shows *FL-valid* $P x \longleftrightarrow$ *L-transform.valid* (*EF* (F, P)) (*L-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-valid* P (*Pred* $f \varphi$)

then have *L-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 $\langle \text{Eff } f, AC(f, F, \langle f \rangle P) \rangle$

by (*metis* *L-transition.simps(2)*)

ultimately show *L-transform.valid* (*EF* (F, P)) (*L-transform* (*Pred* $f \varphi$))

using *L-transform.valid-Act* **by** *fastforce*

next

assume *L-transform.valid* (*EF* (F, P)) (*L-transform* (*Pred* $f \varphi$))

then obtain P' **where** *trans*: *EF* (F, P) \rightarrow_L $\langle \text{Eff } f, P' \rangle$ **and** *valid*:

L-transform.valid $P' ? \varphi$

by *simp* (*metis* *bn-L-action.simps(2)* *empty-iff* *fresh-star-def* *L-transform.valid-Act-fresh*

L-transform.valid-Pred *L-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-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** $fresh: bn \alpha' \#^* (F, f)$
by (*metis FL-valid-Act-strong finite-supp*)
from eq **obtain** p **where** $p\text{-}x: p \cdot x = x'$ **and** $p\text{-}\alpha: p \cdot \alpha = \alpha'$ **and** $supp\text{-}p: supp \ p \subseteq bn \ \alpha \cup bn \ \alpha'$
by (*metis bn-eqvt FL-Formula.Act-eq-iff-perm-renaming*)
from $\langle bn \ \alpha \#^* (F, f) \rangle$ **and** $fresh$ **have** $supp \ (F, f) \#^* p$
using $supp\text{-}p$ **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 $supp\text{-}perm\text{-}eq$ **by** *fastforce+*

from $valid$ **have** $FL\text{-valid } (-p \cdot P')$ x
using $p\text{-}x$ **by** (*metis FL-valid-eqvt permute-minus-cancel(2)*)
then have $L\text{-transform.valid } (EF \ (L \ (\alpha, F, f), -p \cdot P'))$ ($L\text{-transform } x$)
using $Act.hyps(4)$ **by** *metis*
then have $L\text{-transform.valid } (p \cdot EF \ (L \ (\alpha, F, f), -p \cdot P'))$ ($p \cdot L\text{-transform } x$)
by (*fact L-transform.valid-eqvt*)
then have $L\text{-transform.valid } (EF \ (L \ (\alpha', F, f), P'))$ ($L\text{-transform } x'$)
using $p\text{-}x$ **and** $p\text{-}\alpha$ **and** $\langle p \cdot F = F \rangle$ **and** $\langle p \cdot f = f \rangle$ **by** *simp*

then have $L\text{-transform.valid } (AC \ (f, F, \langle f \rangle P))$ ($Formula.Act \ (Act \ \alpha')$) ($L\text{-transform } x'$)
using $trans$ $fresh$ $L\text{-transform.valid-Act}$ **by** *fastforce*
with $\langle f \in_{fs} F \rangle$ **and** eq **show** $L\text{-transform.valid } (EF \ (F, P))$ ($L\text{-transform } (FL\text{-Formula.Act } f \alpha x)$)
using $L\text{-transform.valid-Act}$ **by** *fastforce*
next
assume $*$: $L\text{-transform.valid } (EF \ (F, P))$ ($L\text{-transform } (FL\text{-Formula.Act } f \alpha x)$)

— rename $bn \ \alpha$ to avoid (F, f, P) , without touching F or $FL\text{-Formula.Act } f \alpha x$
obtain p **where** $1: (p \cdot bn \ \alpha) \#^* (F, f, P)$ **and** $2: supp \ (F, FL\text{-Formula.Act } f \alpha x) \#^* p$
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 $finite \ (supp \ (F, FL\text{-Formula.Act } f \alpha x))$ **by** (*simp add: finite-supp*)
next
from $\langle bn \ \alpha \#^* (F, f) \rangle$ **show** $bn \ \alpha \#^* (F, FL\text{-Formula.Act } f \alpha x)$
by (*simp add: fresh-star-Pair fresh-star-def fresh-def supp-Pair*)

```

qed metis
from 2 have supp F  $\sharp^*$  p and Act-fresh: supp (FL-Formula.Act f  $\alpha$  x)  $\sharp^*$  p
  by (simp add: fresh-star-Pair fresh-star-def supp-Pair)+
from  $\langle$ supp F  $\sharp^*$  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  $\alpha$  x = FL-Formula.Act f (p  $\cdot$   $\alpha$ )
(p  $\cdot$  x)
  by (metis FL-Formula.Act-eq-iff-perm FL-Formula.Act-eqt supp-perm-eq)

  with * obtain P' where trans: EF (F, P)  $\rightarrow_L$   $\langle$ Eff f,P $\wedge$  $\rangle$  and valid:
L-transform.valid P' (Formula.Act (Act (p  $\cdot$   $\alpha$ )) (L-transform (p  $\cdot$  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-eqt supp-fun-app supp-fun-app-eqt by fastforce
with 1 have bn (Act (p  $\cdot$   $\alpha$ ))  $\sharp^*$  AC (f, F,  $\langle$ f $\rangle$ P)
  by (auto simp add: bn-eqt 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-transform.valid P'' (L-transform (p  $\cdot$  x))
  using P' by (metis L-transform.valid-Act-fresh)

  from supp-f-P and 1 have bn (p  $\cdot$   $\alpha$ )  $\sharp^*$  (F, f,  $\langle$ f $\rangle$ P)
  by (auto simp add: bn-eqt fresh-star-def fresh-def supp-Pair)
  with trans' obtain P' where P'': P'' = EF (L (p  $\cdot$   $\alpha$ , F, f), P') and trans'':
 $\langle$ f $\rangle$ P  $\rightarrow$   $\langle$ p  $\cdot$   $\alpha$ ,P $\wedge$  $\rangle$ 
  by (metis L-transition-AC-fresh)

  from valid' have L-transform.valid ( $-$ p  $\cdot$  P'') (L-transform x)
  by (metis (mono-tags) L-transform.valid-eqt L-transform-eqt permute-minus-cancel(2))
  with P''  $\langle$ p  $\cdot$  F = F $\rangle$   $\langle$ p  $\cdot$  f = f $\rangle$  have L-transform.valid (EF (L ( $\alpha$ , F, f),
 $-$  p  $\cdot$  P')) (L-transform x)
  by simp (metis permute-minus-self permute-minus-cancel(1))
  then have FL-valid P' (p  $\cdot$  x)
  using Act.hyps(4) by (metis FL-valid-eqt permute-minus-cancel(1))

  with trans'' and eq show FL-valid P (FL-Formula.Act f  $\alpha$  x)
  by (metis FL-valid-Act)
qed
qed
end

```

19.6 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

$P \sim_L [F] Q \Longrightarrow L\text{-bisimilar } (EF (F,P)) (EF (F,Q))$
 $| P \sim_L [F] Q \Longrightarrow f \in_{fs} F \Longrightarrow L\text{-bisimilar } (AC (f, F, \langle f \rangle P)) (AC (f, F, \langle f \rangle Q))$

lemma *L-bisimilar-is-L-transform-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:* $P \sim_L [F] Q$

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 $\langle \wedge Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \Longrightarrow L\text{-bisimilar } P_L' Q_L' \Longrightarrow \textit{thesis} \rangle$

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** $bn \alpha \#* (F, f)$
by (*meson fresh-PairD(2) fresh-star-def*)
with Q_L **and** $trans-Q$ **have** $trans-Q_L: Q_L \rightarrow_L \langle Act \alpha, EF (L (\alpha, F, f),$
 $Q' \rangle$
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** $supp-p: supp p \subseteq bn \alpha \cup bn \alpha_L$
by (*metis (no-types, lifting) bn-L-action.simps(1) residual-eq-iff-perm-renaming*)
from $supp-p$ **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, opaque-lifting) Un-iff*
fresh-Pair fresh-def subsetCE supp-AC)
then **have** $p \cdot fQ: p \cdot \langle f \rangle Q = \langle f \rangle Q$ **and** $p \cdot Ff: p \cdot (F, f) = (F, f)$
by (*simp add: fresh-star-def perm-supp-eq+*)
from p **and** $p \cdot Ff$ **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** $p \cdot fQ$ **and** $p \cdot Ff$ **have** $p \cdot Q_L = Q_L$
by *simp*
with $trans-Q_L$ **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 $p \cdot Ff$ **by** *simp*

moreover from $p \cdot Ff$ **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 -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 $\langle \wedge Q_L'. Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \implies L$ -bisimilar $P_L' Q_L' \implies thesis$
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-FL-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 (invL-FL-bisimilar F) (is ?R)*
by (*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 *bisim: invL-FL-bisimilar F P Q* **and** *effect: f \in_{fs} F* **and** *satisfies:*
 $\langle f \rangle P \vdash \varphi$
from *bisim* **have** $EF (F, P) \sim_L EF (F, Q)$
by (*metis invL-FL-bisimilar-def*)
moreover have $bn (Eff f) \#* EF (F, Q)$
by (*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 (*metis L-transition.simps(2)*)
ultimately obtain Q_L' **where** *trans: EF (F, Q) \rightarrow_L $\langle Eff f, Q_L \rangle$* **and**
L-bisim: AC (f, F, $\langle f \rangle P$) \sim_L Q_L'
by (*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 (*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 *satisfies* **have** $AC (f, F, \langle f \rangle P) \vdash_L \varphi$
by (*metis L-satisfies.simps(1)*)
with *L-bisim* **and** Q_L' **have** $AC (f, F, \langle f \rangle Q) \vdash_L \varphi$
by (*metis L-transform.bisimilar-is-bisimulation L-transform.is-bisimulation-def*)
then show $\langle f \rangle Q \vdash \varphi$
by (*metis L-satisfies.simps(1)*)
qed
moreover have $\forall P Q. \text{invL-FL-bisimilar } F P Q \longrightarrow (\forall f. f \in_{fs} F \longrightarrow (\forall \alpha. P'. bn \alpha \#* (\langle f \rangle Q, F, f) \longrightarrow \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge \text{invL-FL-bisimilar } (L (\alpha, F, f)) P' Q'))$ **(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 (*metis invL-FL-bisimilar-def*)
moreover **have** $\text{bn} (Eff f) \#^* EF (F, Q)$
by (*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 (*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 (*metis L-transform.bisimilar-simulation-step*)
from trans_L **obtain** f' **where** $\langle Eff f :: ('act, 'effect) L\text{-action}, Q_L \rangle = \langle Eff$
 $f', AC (f', F, \langle f \rangle Q) \rangle$
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 $L\text{-bisim}$ **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** $\text{bn} (Act \alpha) \#^* AC (f, F, \langle f \rangle Q)$
by (*simp add: fresh-def fresh-star-def supp-Pair*)
moreover **from** fresh **have** $\text{bn } \alpha \#^* (F, f)$
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** $\text{trans}_L': AC (f, F, \langle f \rangle Q) \rightarrow_L \langle Act \alpha, Q_L'' \rangle$
and $L\text{-bisim}': EF (L (\alpha, F, f), P') \sim_L Q_L''$
by (*metis L-transform.bisimilar-simulation-step*)

have $\text{finite} (\text{supp} (\langle f \rangle Q, F, f))$
by (*fact finite-supp*)
with trans_L' **obtain** $\alpha' Q'$ **where** $\text{trans}': \langle f \rangle Q \rightarrow \langle \alpha', Q' \rangle$ **and** $\text{alpha}: \langle Act$
 $\alpha :: ('act, 'effect) L\text{-action}, Q_L'' \rangle = \langle Act \alpha', EF (L (\alpha', F, f), Q') \rangle$ **and** $\text{fresh}': \text{bn}$
 $\alpha' \#^* (\langle f \rangle Q, F, f)$
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** $\text{supp-}p: \text{supp } p \subseteq \text{bn } \alpha \cup \text{bn } \alpha'$
by (*metis Un-commute bn-L-action.simps(1) residual-eq-iff-perm-renaming*)
from $\text{supp-}p$ **and** fresh **and** fresh' **have** $\text{supp } p \#^* (\langle f \rangle Q, F, f)$
unfolding fresh-star-def **by** (*metis (no-types, opaque-lifting) Un-iff subsetCE*)
then **have** $p\text{-}fQ: p \cdot \langle f \rangle Q = \langle f \rangle Q$ **and** $p\text{-}F: p \cdot F = F$ **and** $p\text{-}f: p \cdot f = f$
by (*simp add: fresh-star-def perm-supp-eq*)
from p **and** $p\text{-}F$ **and** $p\text{-}f$ **have** $p\text{-}\alpha': p \cdot \alpha' = \alpha$ **and** $Q_L'': Q_L'' = EF (L$
 $(p \cdot \alpha', F, f), p \cdot Q')$
by *auto*

from $trans'$ **and** $p\text{-}fQ$ **and** $p\text{-}\alpha'$ **have** $\langle f \rangle Q \rightarrow \langle \alpha, p \cdot Q \rangle$
by (*metis transition-eqv'*)
moreover from $L\text{-}bisim'$ **and** Q_L'' **and** $p\text{-}\alpha'$ **have** $invL\text{-}FL\text{-}bisimilar (L$
 $(\alpha, F, f)) P' (p \cdot Q')$
by (*metis invL-FL-bisimilar-def*)
ultimately show $\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \wedge invL\text{-}FL\text{-}bisimilar (L (\alpha, F, f)) P'$
 Q'
by *metis*
qed
ultimately show $?R \wedge ?S \wedge ?T$
by *metis*
qed

theorem $P \sim_{\cdot[F]} Q \longleftrightarrow EF (F, P) \sim_{\cdot L} EF(F, Q)$

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 $EF (F, P) \sim_{\cdot L} EF(F, Q)$

by (*metis L-bisimilar-is-L-transform-bisimulation L-transform.bisimilar-def*)

next

assume $EF (F, P) \sim_{\cdot L} EF (F, Q)$

then have $invL\text{-}FL\text{-}bisimilar F P Q$

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 $\llbracket indexed\text{-}nominal\text{-}ts\ TYPE(?'idx) ?satisfies ?transition; nominal\text{-}ts.bisimilar ?satisfies ?transition ?P ?Q \rrbracket \implies indexed\text{-}nominal\text{-}ts.logically\text{-}equivalent\ TYPE(?'idx) ?satisfies ?transition ?P ?Q$ does not actually make use of the cardinality assumptions provided by indexed nominal transition systems.

locale $L\text{-}transform\text{-}indexed\text{-}effect\text{-}nominal\text{-}ts = indexed\text{-}effect\text{-}nominal\text{-}ts L\ satisfies\ transition\ effect\ apply$

for $L :: ('act::bn) \times ('effect::fs)\ fs\ set \times 'effect \Rightarrow 'effect\ fs\ set$

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-apply :: 'effect  $\Rightarrow$  'state  $\Rightarrow$  'state ( $\langle \cdot \rangle$ - [0,101] 100) +
assumes card-idx-L-transform-state: |UNIV::('state, 'effect) L-state set| < o |UNIV::'idx
set|
begin

  interpretation L-transform: indexed-nominal-ts ( $\vdash_L$ ) ( $\rightarrow_L$ )
  by unfold-locales (fact L-satisfies-eqvt, fact L-transition-eqvt, fact card-idx-perm,
fact card-idx-L-transform-state)

  notation L-transform.bisimilar (infix  $\sim_L$  100)

  theorem EF (F,P)  $\sim_L$  EF(F,Q)  $\longrightarrow$  P  $\sim$  [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  $\sim$  [F] Q
      by (fact FL-equivalence-implies-bisimilarity)
  qed

end

end
theory Weak-Transition-System
imports
  Transition-System
begin

```

20 Nominal Transition Systems and Bisimulations with Unobservable Transitions

20.1 Nominal transition systems with unobservable transitions

```

locale weak-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) +
  fixes  $\tau$  :: 'act
  assumes tau-eqvt [eqvt]:  $p \cdot \tau = \tau$ 
begin

  lemma bn-tau-empty [simp]:  $bn \ \tau = \{\}$ 
  using bn-eqvt bn-finite tau-eqvt by (metis eqvt-def supp-finite-atom-set supp-fun-eqvt)

  lemma bn-tau-fresh [simp]:  $bn \ \tau \ \#^* P$ 
  by (simp add: fresh-star-def)

```

inductive *tau-transition* :: 'state ⇒ 'state ⇒ bool (**infix** ⇒ 70) **where**

tau-refl [*simp*]: $P \Rightarrow P$

| *tau-step*: $\llbracket P \rightarrow \langle \tau, P^\wedge \rangle; P' \Rightarrow P'' \rrbracket \Longrightarrow P \Rightarrow P''$

definition *observable-transition* :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/ ⇒{-}/ - [70, 70, 71] 71) **where**

$P \Rightarrow \{\alpha\} P' \equiv \exists Q Q'. P \Rightarrow Q \wedge Q \rightarrow \langle \alpha, Q^\wedge \rangle \wedge Q' \Rightarrow P'$

definition *weak-transition* :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/ ⇒<)/ - [70, 70, 71] 71) **where**

$P \Rightarrow \langle \alpha \rangle P' \equiv \text{if } \alpha = \tau \text{ then } P \Rightarrow P' \text{ else } P \Rightarrow \{\alpha\} P'$

The transition relations defined above are equivariant.

lemma *tau-transition-eqt* :

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 $\langle P \rightarrow \langle \tau, P^\wedge \rangle \rangle$ **have** $p \cdot P \rightarrow \langle \tau, p \cdot P^\wedge \rangle$

using *tau-eqt transition-eqt'* **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-eqt* :

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-eqt' tau-transition-eqt*)

lemma *weak-transition-eqt* :

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-eqt permute-minus-cancel(2) tau-eqt tau-transition-eqt*)

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** $Q \rightarrow \langle \alpha, Q^\wedge \rangle$ **and** $Q' \Rightarrow P'$

shows $P \Rightarrow \{\alpha\} P'$

using *assms* *observable-transition-def* **by** *blast*

lemma *observable-transition-stepI* [*simp*]:

assumes $P \rightarrow \langle \alpha, P^\wedge \rangle$

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*]:
 $P \Rightarrow\langle \tau \rangle P' \longleftrightarrow P \Rightarrow P'$
by (*simp add: weak-transition-def*)

lemma *weak-transition-not-tau-iff* [*simp*]:
assumes $\alpha \neq \tau$
shows $P \Rightarrow\langle \alpha \rangle P' \longleftrightarrow P \Rightarrow\{\alpha\} P'$
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

20.2 Weak bisimulations

context *weak-nominal-ts*
begin

definition *is-weak-bisimulation* :: $('state \Rightarrow 'state \Rightarrow bool) \Rightarrow bool$ **where**
is-weak-bisimulation $R \equiv$

$\text{symp } R \wedge$
 — 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 $bn (-p \cdot \alpha) \#* (-p \cdot Q)$
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 \exists **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-eqvt* :
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-eqvt*)
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)

by (*fact weakly-bisimilar-symp*)

next

show ($\forall P Q \varphi. P \approx Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge P \approx Q' \wedge Q' \vdash \varphi) \wedge$

$(\forall P Q. P \approx Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle$

$Q' \wedge P' \approx Q'))$)
by (*auto simp add: is-weak-bisimulation-def weakly-bisimilar-def*) *blast+*
qed

lemma *weakly-bisimilar-tau-simulation-step*:

assumes $P \approx Q$ **and** $P \Rightarrow P'$

obtains Q' **where** $Q \Rightarrow Q'$ **and** $P' \approx Q'$

using $\langle P \Rightarrow P' \rangle$ $\langle P \approx Q \rangle$ **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** $\langle P \approx Q \rangle$ **obtain** Q'' **where** $Q \Rightarrow Q''$ **and** $P'' \approx Q''$

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 $P \approx Q$ **and** $\text{bn } \alpha \#* Q$ **and** $P \Rightarrow \langle \alpha \rangle P'$

obtains Q' **where** $Q \Rightarrow \langle \alpha \rangle Q'$ **and** $P' \approx Q'$

proof (*cases* $\alpha = \tau$)

case *True* **with** $\langle P \approx Q \rangle$ **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** $\text{tauP}: P \Rightarrow P1$ **and** $\text{trans}: P1 \rightarrow \langle \alpha, P2 \rangle$ **and**
tauP2: P2 \Rightarrow P'

using *observable-transition-def* **by blast**

from $\langle P \approx Q \rangle$ **and** tauP **obtain** $Q1$ **where** $\text{tauQ}: Q \Rightarrow Q1$ **and** $P1Q1: P1 \approx Q1$

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 bn α Q1 ($\langle \alpha, P2 \rangle, Q$), THEN exE]*)

show *finite* ($\text{bn } \alpha$) **by** (*fact bn-finite*)

next


```

  show finite (supp Q1) by (fact finite-supp)
next
  show finite (supp (( $\alpha, P2$ ), Q)) by (simp add: finite-supp supp-Pair)
next
  show  $bn \ \alpha \ \#* \ (\langle \alpha, P2 \rangle, Q)$  using  $\langle bn \ \alpha \ \#* \ Q \rangle$  by (simp add: fresh-star-Pair)
qed metis
from 2 have 3:  $supp \ \langle \alpha, P2 \rangle \ \#* \ p$  and 4:  $supp \ Q \ \#* \ p$ 
  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-eqvt)
with P2Q2 obtain Q' where  $tauQ2: Q2 \Rightarrow Q'$  and  $P'Q': (p \cdot P') \approx \cdot \ Q'$ 
  by (metis weakly-bisimilar-tau-simulation-step)

```

```

from tauQ and  $trans'$  and tauQ2 have  $Q \Rightarrow \langle p \cdot \alpha \rangle \ Q'$ 
  by (rule weak-transition-weakI)
with 4 have  $Q \Rightarrow \langle \alpha \rangle \ (-p \cdot Q')$ 
  by (metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqvt)
moreover from P'Q' have  $P' \approx \cdot \ (-p \cdot Q')$ 
  by (metis permute-minus-cancel(2) weakly-bisimilar-eqvt)
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 φ R

assume 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 $\forall P Q. ?bisim P Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle$
 $\longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge ?bisim P' Q')$
proof (*clarify*)
fix $P Q R \alpha P'$
assume PR : $P \approx \cdot R$ and RQ : $R \approx \cdot Q$ and *fresh*: $\text{bn } \alpha \#* Q$ 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* ($\text{supp } R$) by (*fact finite-supp*)
next
show *finite* ($\text{supp } (\langle \alpha, P' \rangle, Q)$) by (*simp add: finite-supp supp-Pair*)
next
show $\text{bn } \alpha \#* (\langle \alpha, P' \rangle, Q)$ by (*simp add: fresh fresh-star-Pair*)
qed *metis*
from 2 have 3: $\text{supp } \langle \alpha, P' \rangle \#* p$ and 4: $\text{supp } Q \#* p$
by (*simp add: fresh-star-Un supp-Pair*) $+$
from 3 have $\langle p \cdot \alpha, p \cdot P' \rangle = \langle \alpha, P' \rangle$
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 $\text{bn } (p \cdot \alpha) \#* Q$
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, opaque-lifting*) *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  $P \approx R$ 
    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
  Disjunction
begin

```

21 Weak Formulas

21.1 Lemmas about α -equivalence involving τ

```

context weak-nominal-ts
begin

lemma Act-tau-eq-iff [simp]:
   $Act \tau x1 = Act \alpha x2 \longleftrightarrow \alpha = \tau \wedge x2 = x1$ 
  (is ?l  $\longleftrightarrow$  ?r)
proof
  assume ?l
  then obtain p where  $p\alpha: p \cdot \tau = \alpha$  and  $p\alpha: p \cdot x1 = x2$  and fresh: (supp
x1 - bn  $\tau$ )  $\#* p$ 
    by (metis Act-eq-iff-perm)
  from  $p\alpha$  have  $\alpha = \tau$ 
    by (metis tau-eqvt)
  moreover from fresh and  $p\alpha$  have  $x2 = x1$ 
    by (simp add: supp-perm-eq)
  ultimately show ?r ..
next
  assume ?r then show ?l
    by simp
  qed
end

```

21.2 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::perm set| <o |UNIV::'idx set|
    and card-idx-state: |UNIV::'state set| <o |UNIV::'idx set|
    and card-idx-nat: |UNIV::nat set| <o |UNIV::'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::nat set| <o natLeq +c |UNIV::'idx set|
proof -
  note card-idx-nat
  also have |UNIV :: 'idx set|  $\leq_o$  natLeq +c |UNIV :: '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
    tau-steps x 0 = x
  | tau-steps x (Suc n) = Act  $\tau$  (tau-steps x n)

```

```

lemma tau-steps-eqvt [simp]:
  p  $\cdot$  tau-steps x n = tau-steps (p  $\cdot$  x) (p  $\cdot$  n)
  by (induct n) (simp-all add: permute-nat-def tau-eqvt)

```

```

lemma tau-steps-eqvt' [simp]:
  p  $\cdot$  tau-steps x = tau-steps (p  $\cdot$  x)
  by (simp add: permute-fun-def)

```

```

lemma tau-steps-eqvt-raw [simp]:
  p  $\cdot$  tau-steps = tau-steps
  by (simp add: permute-fun-def)

```

```

lemma tau-steps-add [simp]:
  tau-steps (tau-steps x m) n = tau-steps x (m + n)
  by (induct n) auto

```

```

lemma tau-steps-not-self:
  x = 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  $\alpha$  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*]:

$\text{finite } (\text{supp } (\text{map-bset } (\text{tau-steps } x) \ (\text{Abs-bset } \text{UNIV} :: \text{nat set}['\text{idx}])))$

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 $\text{supp } (\text{Abs-bset } \text{UNIV} :: \text{nat set}['\text{idx}]) = \{\}$

by (*simp add: eqvtI supp-fun-eqvt*)

ultimately have $\text{supp } (\text{map-bset } (\text{tau-steps } x) \ (\text{Abs-bset } \text{UNIV} :: \text{nat set}['\text{idx}])))$

$\subseteq \text{supp } x$

using *supp-fun-app* **by** *blast*

then show ?*thesis*

by (*metis finite-subset finite-supp*)

qed

lemma *weak-tau-modality-eqvt* [*simp*]:

$p \cdot \text{weak-tau-modality } x = \text{weak-tau-modality } (p \cdot x)$

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 *weak-tau-modality* $x = \text{weak-tau-modality } y$
then have $\text{map-bset } (\text{tau-steps } x) (\text{Abs-bset UNIV} :: - \text{set}['idx]) = \text{map-bset } (\text{tau-steps } y) (\text{Abs-bset UNIV})$
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** *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* :: $'act \Rightarrow ('idx, 'pred :: fs, 'act :: bn) \text{ formula} \Rightarrow$
 $('idx, 'pred, 'act) \text{ formula } (\langle\langle - \rangle\rangle -)$
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* [*simp*]:
 $p \cdot (\langle\langle \alpha \rangle\rangle x) = \langle\langle p \cdot \alpha \rangle\rangle (p \cdot x)$
using *tau-eqvt weak-action-modality-def* **by** *fastforce*

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 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  $1$ : supp  $x 1 - \text{bn } \alpha 1 = \text{supp } x 2 - \text{bn } \alpha 2$  and  $2$ :
    (supp  $x 1 - \text{bn } \alpha 1$ )  $\#^* p$ 
    and  $3$ :  $p \cdot x 1 = x 2$  and  $4$ : supp  $\alpha 1 - \text{bn } \alpha 1 = \text{supp } \alpha 2 - \text{bn } \alpha 2$  and  $5$ :
    (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 supp (weak-tau-modality  $x 1$ ) - bn  $\alpha 1 = \text{supp}$  (weak-tau-modality
     $x 2$ ) - bn  $\alpha 2$ 
    by (metis supp-weak-tau-modality)
    moreover from  $2$  have (supp (weak-tau-modality  $x 1$ ) - 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-eqt)
    ultimately have Act  $\alpha 1$  (weak-tau-modality  $x 1$ ) = Act  $\alpha 2$  (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

```

21.3 Weak formulas

```

inductive weak-formula :: ('idx,'pred::fs,'act::bn) formula  $\Rightarrow$  bool
  where
    wf-Conj: finite (supp  $xset$ )  $\Longrightarrow$  ( $\bigwedge x. x \in \text{set-bset } xset \Longrightarrow \text{weak-formula } x$ )
 $\Longrightarrow$  weak-formula (Conj  $xset$ )
    | wf-Not: weak-formula  $x \Longrightarrow \text{weak-formula}$  (Not  $x$ )
    | wf-Act: weak-formula  $x \Longrightarrow \text{weak-formula}$  ( $\langle\langle\alpha\rangle\rangle x$ )
    | wf-Pred: weak-formula  $x \Longrightarrow \text{weak-formula}$  ( $\langle\langle\tau\rangle\rangle(\text{Conj } (\text{binsert } (\text{Pred } \varphi)
    (\text{bsingleton } x))))$ 

```

```

lemma finite-supp-wf-Pred [simp]: finite (supp (binsert (Pred  $\varphi$ ) (bsingleton  $x$ )))
proof -

```

```

have supp (bsingleton x) ⊆ supp x
  by (simp add: eqvtI supp-fun-app-eqvt)
moreover have eqvt binsert
  by (simp add: eqvtI)
ultimately have supp (binsert (Pred φ) (bsingleton x)) ⊆ supp φ ∪ supp x
  using supp-fun-app supp-fun-app-eqvt by fastforce
then show ?thesis
  by (metis finite-UnI finite-supp rev-finite-subset)
qed

```

weak-formula is equivariant.

```

lemma weak-formula-eqvt [simp]: weak-formula x ⇒ weak-formula (p · 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-formula.wf-Pred)
qed

```

end

end

```

theory Weak-Validity
imports
  Weak-Formula
  Validity
begin

```

22 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 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 $P' \models \text{tau-steps } x \ n$

by *meson*

with $\langle P \rightarrow \langle \tau, P' \rangle \rangle$ have $P \models \text{tau-steps } x \ (\text{Suc } n)$

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  $trans: P \Rightarrow P'$  and  $valid: P' \models x$ 
    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  $eq: Act \alpha x = Act \alpha' x'$  and  $trans: P \Rightarrow \langle \alpha' \rangle$ 
 $P'$  and  $valid: P' \models x'$ 
    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  $trans: P \Rightarrow Q$  and  $valid: Q \models Act \alpha$ 
(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 $Q' \models \text{weak-tau-modality } x'$
 by (*metis valid-weak-tau-modality*)
 with *trans''* have $Q \models \text{Act } \alpha' \text{ (weak-tau-modality } x')$
 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 $P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists \alpha' x' P'. \text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \Rightarrow \langle \alpha' \rangle P' \wedge P' \models x' \wedge \text{bn } \alpha' \#* X)$
proof
 assume $P \models \langle \langle \alpha \rangle \rangle x$
 then obtain $\alpha' x' P'$ where *eq*: $\text{Act } \alpha x = \text{Act } \alpha' x'$ and *trans*: $P \Rightarrow \langle \alpha' \rangle P'$
 and valid: $P' \models x'$
 by (*metis valid-weak-action-modality*)
 show $\exists \alpha' x' P'. \text{Act } \alpha x = \text{Act } \alpha' x' \wedge P \Rightarrow \langle \alpha' \rangle P' \wedge P' \models x' \wedge \text{bn } \alpha' \#* X$
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* α')
 by (*fact bn-finite*)
 moreover note $\langle \text{finite } (\text{supp } X) \rangle$
 moreover have *finite* (*supp* ($\text{Act } \alpha' x', \langle \alpha', Q' \rangle$))
 by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)
 moreover have $\text{bn } \alpha' \#* (\text{Act } \alpha' x', \langle \alpha', Q' \rangle)$
 by (*auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
 ultimately obtain p where *fresh-X*: $(p \cdot \text{bn } \alpha') \#* X$ and *supp* ($\text{Act } \alpha'$

$x', \langle \alpha', Q' \rangle \#* p$
by (*metis at-set-avoiding2*)
then have $\text{supp } (Act \alpha' x') \#* p$ **and** $\text{supp } \langle \alpha', Q' \rangle \#* p$
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* ($\text{supp } P$)
by (*fact finite-supp*)
ultimately obtain $\alpha' x' P'$ **where**
 $eq: Act \alpha x = Act \alpha' x'$ **and** $\text{trans}: P \Rightarrow \langle \alpha' \rangle P'$ **and** $\text{valid}: P' \models x'$ **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** $\text{supp-}p$:
 $\text{supp } p \subseteq bn \alpha \cup p \cdot bn \alpha$
by (*metis Act-eq-iff-perm-renaming*)

from *assms* **and** *fresh* **have** $(bn \alpha \cup p \cdot bn \alpha) \#* P$
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** $P \Rightarrow \langle \alpha \rangle (-p \cdot P')$
using $p\text{-}P$ $p\text{-}\alpha$ **by** (*metis permute-minus-cancel(1) weak-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 $\exists P'. P \Rightarrow \langle \alpha \rangle P' \wedge P' \models x$
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

```

23 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  $P Q \equiv (\forall x::('idx, 'pred, 'act) \text{ formula. weak-formula } x \rightarrow P \models x \leftrightarrow Q \models x)$ 

```

```

notation weakly-logically-equivalent (infix  $\equiv$  50)

```

```

lemma weakly-logically-equivalent-eqvt:
  assumes  $P \equiv Q$  shows  $p \cdot P \equiv p \cdot Q$ 
unfolding weakly-logically-equivalent-def proof (clarify)
  fix  $x :: ('idx, 'pred, 'act) \text{ formula}$ 
  assume weak-formula  $x$ 
  then have weak-formula  $(-p \cdot x)$ 
  by simp
  then show  $p \cdot P \models x \leftrightarrow p \cdot Q \models x$ 
  using assms by (metis (no-types, lifting) weakly-logically-equivalent-def per-
mute-minus-cancel(2) valid-eqvt)
qed

```

```

end

```

```

end
theory Weak-Bisimilarity-Implies-Equivalence
imports
  Weak-Logical-Equivalence
begin

```

24 Weak Bisimilarity Implies Weak Logical Equivalence

context *indexed-weak-nominal-ts*
begin

lemma *weak-bisimilarity-implies-weak-equivalence-Act:*

assumes $\bigwedge P Q. P \approx \cdot Q \implies P \models x \longleftrightarrow Q \models x$

and $P \approx \cdot Q$

— 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*: $\text{Act } \alpha x = \text{Act } \alpha' x'$ **and** *trans*:

$P \Rightarrow \langle\alpha'\rangle P'$ **and** *valid*: $P' \models x'$ **and** *fresh*: $\text{bn } \alpha' \#* Q$

by (*metis valid-weak-action-modality-strong*)

from $\langle P \approx \cdot Q \rangle$ **and** *fresh* **and** *trans* **obtain** Q' **where** *trans'*: $Q \Rightarrow \langle\alpha'\rangle Q'$

and *bisim'*: $P' \approx \cdot Q'$

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-eqvt*)

moreover from *bisim'* **have** $(-p \cdot P') \approx \cdot (-p \cdot Q')$

by (*metis weakly-bisimilar-eqvt*)

ultimately have $-p \cdot Q' \models x$

using $\langle \bigwedge P Q. P \approx \cdot Q \implies P \models x \longleftrightarrow Q \models x \rangle$ **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 $\bigwedge P Q. P \approx \cdot Q \implies P \models x \longleftrightarrow Q \models x$

and $P \approx \cdot Q$

— 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** *trans*: $P \Rightarrow P'$ **and** *valid*: $P' \models ?c$

```

    using valid-weak-action-modality by auto

  have bn  $\tau \#* Q$ 
    by (simp add: fresh-star-def)
  with  $\langle P \approx \cdot Q \rangle$  and trans obtain  $Q'$  where trans':  $Q \Rightarrow Q'$  and bisim':  $P' \approx \cdot Q'$ 
    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 trans'':  $Q' \Rightarrow Q''$  and bisim'':  $P' \approx \cdot Q''$ 
  and  $***$ :  $Q'' \vdash \varphi$ 
    by (metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation)

  from bisim'' and  $**$  have  $Q'' \models x$ 
    using  $\langle \bigwedge P Q. P \approx \cdot Q \implies P \models x \longleftrightarrow Q \models x \rangle$  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  $P \approx \cdot Q$  shows  $P \equiv \cdot Q$ 
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

```

25 Weak Logical Equivalence Implies Weak Bisimilarity

```

context indexed-weak-nominal-ts
begin

```

```

  definition is-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 is-distinguishing-formula-eqvt [simp]:
    assumes x distinguishes P from Q shows  $(p \cdot x)$  distinguishes  $(p \cdot P)$  from  $(p \cdot Q)$ 
  using assms unfolding is-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 is-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  $\text{supp } P$  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 == b) · ?B ⊆ ?B
proof
  fix x'
  assume x' ∈ (a == b) · ?B
  then obtain p where 1: x' = (a == b) · p · x and 2: supp P #* p
    by (auto simp add: permute-set-def)
  let ?q = (a == b) + p
  from 1 have x' = ?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' ∈ ?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
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  $x' = p \cdot x$ 
  using Abs-bset-inverse mem-Collect-eq by auto
then show weak-formula x'
  using  $\langle \text{weak-formula } x \rangle$  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 is-distinguishing-formula-def proof
    from assms show  $P \models ?y$ 
    by (auto simp add: card-B finite-supp-B) (metis is-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 is-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  $P \equiv \cdot Q$  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  $\forall Q' \in ?Q'. \neg P \equiv \cdot Q'$ 
        then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } \text{weak-formula } x \wedge$ 
x distinguishes P from Q'
          by (metis weakly-equivalent-iff-not-distinguished)
          then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. } \text{weak-formula } x \wedge$ 
supp x  $\subseteq$  supp P  $\wedge$  x distinguishes P 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 from Q'
                by metis
                have supp (f ' ?Q')  $\subseteq$  supp P
                  by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)
                  then have finite-supp-image: finite (supp (f ' ?Q'))
                    using finite-supp rev-finite-subset by blast
                    have  $|f \text{ ' ?Q}'| \leq o \mid UNIV :: 'state \text{ set}$ 
                      using card-of-UNIV card-of-image ordLeq-transitive by blast

```

```

also have |UNIV :: 'state set| <o |UNIV :: 'idx set|
  by (metis card-idx-state)
also have |UNIV :: 'idx set| ≤o natLeq + c |UNIV :: 'idx set|
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-image: |f ' ?Q'| <o natLeq + c |UNIV :: 'idx set| .

let ?y = Conj (Abs-bset (f ' ?Q')) :: ('idx, 'pred, 'act) formula
have weak-formula ?y
proof (standard+)
  show finite (supp (Abs-bset (f ' ?Q')) :: - set['idx])
    using finite-supp-image card-image by simp
next
  fix x assume x ∈ set-bset (Abs-bset (f ' ?Q')) :: - set['idx]
  with card-image obtain Q' where Q' ∈ ?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 = ⟨⟨τ⟩⟩(Conj (binsert (Pred φ) (bsingleton ?y)))
have weak-formula ?z
  by standard (fact ⟨weak-formula ?y⟩)
moreover have P ⊨ ?z
proof -
  have P ⇒⟨τ⟩ P
    by simp
  moreover
  {
    fix Q'
    assume Q ⇒ Q' ∧ Q' ⊢ φ
    with * have P ⊨ f Q'
      by (metis is-distinguishing-formula-def mem-Collect-eq)
  }
  with ⟨P ⊢ φ⟩ have P ⊨ Conj (binsert (Pred φ) (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 ¬ Q ⊨ ?z
proof
  assume Q ⊨ ?z
  then obtain Q' where 1: Q ⇒ Q' and Q' ⊨ Conj (binsert (Pred
φ) (bsingleton ?y))
    using valid-weak-action-modality by auto
  then have 2: Q' ⊢ φ and 3: Q' ⊨ ?y
    by (simp add: binsert.rep-eq finite-supp-image card-image)+
  from 3 have ∧Q''. Q ⇒ Q'' ∧ Q'' ⊢ φ ⟶ Q' ⊨ f Q''
    by (simp add: finite-supp-image card-image)
  with 1 and 2 and * show False

```

```

    using is-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  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. weak-formula } x \wedge$ 
x distinguishes  $P'$  from  $Q'$ 
        by (metis weakly-equivalent-iff-not-distinguished)
      then have  $\forall Q' \in ?Q'. \exists x :: ('idx, 'pred, 'act) \text{ formula. weak-formula } x \wedge$ 
supp  $x \subseteq \text{supp } P' \wedge x$  distinguishes  $P'$  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  $P'$  from  $Q'$ 
        by metis
      have supp  $P'$  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  $f \text{ ' } ?Q' \subseteq (a \Rightarrow b) \cdot (f \text{ ' } ?Q')$ 
      proof
      fix  $x$ 
      assume  $x \in f \text{ ' } ?Q'$ 

```

```

then obtain Q' where 1: x = f Q' and 2: Q ⇒⟨α⟩ Q'
  by auto
with * and a and b have a ∉ supp (f Q') and b ∉ supp (f Q')
  by auto
with 1 have x = (a ⇒ b) · f Q'
  by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
with 2 show x ∈ (a ⇒ b) · (f ' ?Q')
  using mem-permute-iff by blast
qed
ultimately show (a ⇒ b) · (f ' ?Q') = f ' ?Q' ..
qed
then have supp-image-subset-supp-P': supp (f ' ?Q') ⊆ supp P'
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have finite-supp-image: finite (supp (f ' ?Q'))
  using finite-supp rev-finite-subset by blast
have |f ' ?Q'| ≤ 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| ≤ o natLeq + c |UNIV :: 'idx set|
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-image: |f ' ?Q'| < o natLeq + c |UNIV :: 'idx set| .

let ?y = Conj (Abs-bset (f ' ?Q')) :: ('idx, 'pred, 'act) formula
have weak-formula (⟨⟨α⟩⟩?y)
  proof (standard+)
    show finite (supp (Abs-bset (f ' ?Q') :: - set['idx]))
      using finite-supp-image card-image by simp
    next
      fix x assume x ∈ set-bset (Abs-bset (f ' ?Q') :: - set['idx'])
      with card-image obtain Q' where Q' ∈ ?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 ⊨ ⟨⟨α⟩⟩?y
  unfolding valid-weak-action-modality proof (standard+)
    from ⟨P → ⟨α, P'⟩⟩ show P ⇒⟨α⟩ P'
      by simp
  next
  {
    fix Q'
    assume Q ⇒⟨α⟩ Q'
    with * have P' ⊨ f Q'
      by (metis is-distinguishing-formula-def mem-Collect-eq)
  }
  then show P' ⊨ ?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 is-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
theory Weak-Expressive-Completeness
imports
  Weak-Bisimilarity-Implies-Equivalence
  Weak-Equivalence-Implies-Bisimilarity
  Disjunction
begin

```

26 Weak Expressive Completeness

context *indexed-weak-nominal-ts*

begin

26.1 Distinguishing weak formulas

Lemma *distinguished_bounded_support* only shows the existence of a distinguishing weak formula, without stating what this formula looks like. We now define an explicit function that returns a distinguishing weak formula, in a way that this function is equivariant (on pairs of non-weakly-equivalent states).

Note that this definition uses Hilbert's choice operator ε , which is not necessarily equivariant. This is immediately remedied by a hull construction.

definition *distinguishing-weak-formula* :: 'state \Rightarrow 'state \Rightarrow ('idx, 'pred, 'act) formula **where**

distinguishing-weak-formula P Q \equiv Conj (Abs-bset {-p \cdot (ϵ x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes (p \cdot P) from (p \cdot Q))|p. True})

— just an auxiliary lemma that will be useful further below

lemma *distinguishing-weak-formula-card-aux*:

{|-p \cdot (ϵ x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes (p \cdot P) from (p \cdot Q))|p. True} |<o natLeq +c |UNIV :: 'idx set|

proof —

let ?some = λ p. (ϵ x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes (p \cdot P) from (p \cdot Q))

let ?B = {-p \cdot ?some p|p. True}

have ?B \subseteq (λ p. -p \cdot ?some p) ' 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 show ?thesis .

qed

— just an auxiliary lemma that will be useful further below

lemma *distinguishing-weak-formula-supp-aux*:

assumes \neg (P \equiv Q)

shows supp (Abs-bset {-p \cdot (ϵ x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes (p \cdot P) from (p \cdot Q))|p. True} :: - set['idx]) \subseteq supp P

proof —

let ?some = λ p. (ϵ x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes (p \cdot P) from (p \cdot Q))

let ?B = {-p \cdot ?some p|p. True}

{

fix p

from assms **have** \neg (p \cdot P \equiv p \cdot Q)

by (metis weakly-logically-equivalent-eqvt permute-minus-cancel(2))

then have supp (?some p) \subseteq supp (p \cdot P)

using distinguished-bounded-support **by** (metis (mono-tags, lifting) weakly-equivalent-iff-not-distinguished someI-ex)

}

note supp-some = this

{

fix x

assume x \in ?B

then obtain p **where** x = -p \cdot ?some p

```

    by blast
  with supp-some have  $\text{supp } (p \cdot x) \subseteq \text{supp } (p \cdot P)$ 
    by simp
  then have  $\text{supp } x \subseteq \text{supp } P$ 
    by (metis (full-types) permute-boolE subset-eqvt supp-eqvt)
}
note * = this
have supp-B:  $\text{supp } ?B \subseteq \text{supp } P$ 
  by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)

from supp-B and distinguishing-weak-formula-card-aux show ?thesis
  using supp-Abs-bset by blast
qed

lemma distinguishing-weak-formula-eqvt [simp]:
  assumes  $\neg (P \equiv Q)$ 
  shows  $p \cdot \text{distinguishing-weak-formula } P \ Q = \text{distinguishing-weak-formula } (p \cdot P) \ (p \cdot Q)$ 
  proof -
    let ?some =  $\lambda p. (\epsilon x. \text{weak-formula } x \wedge \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))$ 
    let ?B =  $\{-p \cdot ?some \ p|p. \text{True}\}$ 

    from assms have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}[idx]) \subseteq \text{supp } P$ 
      by (rule distinguishing-weak-formula-supp-aux)
    then have finite ( $\text{supp } (\text{Abs-bset } ?B :: - \text{set}[idx])$ )
      using finite-supp rev-finite-subset by blast
    with distinguishing-weak-formula-card-aux have *:  $p \cdot \text{Conj } (\text{Abs-bset } ?B) = \text{Conj } (\text{Abs-bset } (p \cdot ?B))$ 
      by simp

    let ?some' =  $\lambda p'. (\epsilon x. \text{weak-formula } x \wedge \text{supp } x \subseteq \text{supp } (p' \cdot p \cdot P) \wedge x \text{ distinguishes } (p' \cdot p \cdot P) \text{ from } (p' \cdot p \cdot Q))$ 
    let ?B' =  $\{-p' \cdot ?some' \ p'|p'. \text{True}\}$ 

    have  $p \cdot ?B = ?B'$ 
    proof
      {
        fix px
        assume  $px \in p \cdot ?B$ 
        then obtain x where 1:  $px = p \cdot x$  and 2:  $x \in ?B$ 
          by (metis (no-types, lifting) image-iff permute-set-eq-image)
        from 2 obtain p' where 3:  $x = -p' \cdot ?some \ p'$ 
          by blast
        from 1 and 3 have  $px = -(p' - p) \cdot ?some' (p' - p)$ 
          by simp
        then have  $px \in ?B'$ 
          by blast
      }
    }
  
```



```

then show  $p \cdot ?B \subseteq ?B'$ 
  by blast
next
  {
    fix  $x$ 
    assume  $x \in ?B'$ 
    then obtain  $p'$  where  $x = -p' \cdot ?some' p'$ 
      by blast
    then have  $x = p \cdot -(p' + p) \cdot ?some (p' + p)$ 
      by (simp add: add.inverse-distrib-swap)
    then have  $x \in p \cdot ?B$ 
      using mem-permute-iff by blast
  }
then show  $?B' \subseteq p \cdot ?B$ 
  by blast
qed

```

```

with * show ?thesis
  unfolding distinguishing-weak-formula-def by simp
qed

```

```

lemma supp-distinguishing-weak-formula:
  assumes  $\neg (P \equiv Q)$ 
  shows  $supp (distinguishing-weak-formula P Q) \subseteq supp P$ 
proof -
  let  $?some = \lambda p. (\epsilon x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes$ 
  ( $p \cdot P$ ) from ( $p \cdot Q$ ))
  let  $?B = \{- p \cdot ?some p | p. True\}$ 

  from assms have  $supp (Abs-bset ?B :: - set['idx]) \subseteq supp P$ 
    by (rule distinguishing-weak-formula-supp-aux)
  moreover from this have  $finite (supp (Abs-bset ?B :: - set['idx]))$ 
    using finite-supp rev-finite-subset by blast
  ultimately show ?thesis
  unfolding distinguishing-weak-formula-def by simp
qed

```

```

lemma distinguishing-weak-formula-distinguishes:
  assumes  $\neg (P \equiv Q)$ 
  shows (distinguishing-weak-formula P Q) distinguishes P from Q
proof -
  let  $?some = \lambda p. (\epsilon x. weak-formula x \wedge supp x \subseteq supp (p \cdot P) \wedge x distinguishes$ 
  ( $p \cdot P$ ) from ( $p \cdot Q$ ))
  let  $?B = \{- p \cdot ?some p | p. True\}$ 

```

```

{
  fix  $p$ 
  from assms have  $\neg (p \cdot P) \equiv (p \cdot Q)$ 
    by (metis permute-minus-cancel(2) weakly-logically-equivalent-eqt)

```

```

    then have (?some p) distinguishes (p · P) from (p · Q)
    by (metis (mono-tags, lifting) distinguished-bounded-support weakly-equivalent-iff-not-distinguished
someI-ex)
  }
  note some-distinguishes = this

  {
    fix P'
    from assms have supp (Abs-bset ?B :: - set['idx]) ⊆ supp P
      by (rule distinguishing-weak-formula-supp-aux)
    then have finite (supp (Abs-bset ?B :: - set['idx']))
      using finite-supp rev-finite-subset by blast
    with distinguishing-weak-formula-card-aux have P' ⊨ distinguishing-weak-formula
P Q ↔ (∀ x ∈ ?B. P' ⊨ x)
      unfolding distinguishing-weak-formula-def by simp
  }
  note valid-distinguishing-formula = this

  {
    fix p
    have P ⊨ ¬p · ?some p
      by (metis (mono-tags) is-distinguishing-formula-def permute-minus-cancel(2)
some-distinguishes valid-eqt)
  }
  then have P ⊨ distinguishing-weak-formula P Q
    using valid-distinguishing-formula by blast

  moreover have ¬ Q ⊨ distinguishing-weak-formula P Q
    using valid-distinguishing-formula by (metis (mono-tags, lifting) is-distinguishing-formula-def
mem-Collect-eq permute-minus-cancel(1) some-distinguishes valid-eqt)

  ultimately show (distinguishing-weak-formula P Q) distinguishes P from Q
    using is-distinguishing-formula-def by blast
qed

lemma distinguishing-weak-formula-is-weak:
  assumes ¬ (P ≡ Q)
  shows weak-formula (distinguishing-weak-formula P Q)
proof -
  let ?some = λp. (ε x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes
(p · P) from (p · Q))
  let ?B = {¬ p · ?some p | p. True}

  from assms have supp (Abs-bset ?B :: - set['idx']) ⊆ supp P
    by (rule distinguishing-weak-formula-supp-aux)
  then have finite (supp (Abs-bset ?B :: - set['idx']))
    using finite-supp rev-finite-subset by blast

  moreover have set-bset (Abs-bset ?B :: - set['idx']) = ?B

```

```

using distinguishing-weak-formula-card-aux Abs-bset-inverse' by simp

moreover
{
  fix x
  assume x ∈ ?B
  then obtain p where x = -p · ?some p
  by blast
  moreover from assms have ¬ (p · P) ≡ (p · Q)
  by (metis permute-minus-cancel(2) weakly-logically-equivalent-eqt)
  then have weak-formula (?some p)
  by (metis (mono-tags, lifting) distinguished-bounded-support weakly-equivalent-iff-not-distinguished
someI-ex)
  ultimately have weak-formula x
  by simp
}

ultimately show ?thesis
unfolding distinguishing-weak-formula-def using wf-Conj by blast
qed

```

26.2 Characteristic weak formulas

A *characteristic weak formula* for a state P is valid for (exactly) those states that are weakly bisimilar to P .

definition *characteristic-weak-formula* :: 'state \Rightarrow ('idx, 'pred, 'act) formula where

characteristic-weak-formula $P \equiv \text{Conj} (\text{Abs-bset} \{ \text{distinguishing-weak-formula } P \ Q \mid Q. \neg (P \equiv \cdot Q) \})$

— just an auxiliary lemma that will be useful further below

lemma *characteristic-weak-formula-card-aux*:

$\{ \text{distinguishing-weak-formula } P \ Q \mid Q. \neg (P \equiv \cdot Q) \} <_o \text{ natLeq } +c \mid \text{UNIV} :: \text{'idx set}$

proof —

let $?B = \{ \text{distinguishing-weak-formula } P \ Q \mid Q. \neg (P \equiv \cdot Q) \}$

have $?B \subseteq (\text{distinguishing-weak-formula } P) \text{ 'UNIV}$

by auto

then have $|?B| \leq_o \mid \text{UNIV} :: \text{'state set}$

by (rule surj-imp-ordLeq)

also have $\mid \text{UNIV} :: \text{'state set} <_o \mid \text{UNIV} :: \text{'idx set}$

by (metis card-idx-state)

also have $\mid \text{UNIV} :: \text{'idx set} \leq_o \text{ natLeq } +c \mid \text{UNIV} :: \text{'idx set}$

by (metis Cnotzero-UNIV ordLeq-csum2)

finally show ?thesis .

qed

— just an auxiliary lemma that will be useful further below

lemma *characteristic-weak-formula-supp-aux*:
shows $\text{supp } (\text{Abs-bset } \{ \text{distinguishing-weak-formula } P \ Q | Q. \neg (P \equiv Q) \}) \subseteq \text{supp } P$
proof –
let $?B = \{ \text{distinguishing-weak-formula } P \ Q | Q. \neg (P \equiv Q) \}$

{
fix x
assume $x \in ?B$
then obtain Q **where** $x = \text{distinguishing-weak-formula } P \ Q$ **and** $\neg (P \equiv Q)$
by *blast*
with *supp-distinguishing-weak-formula* **have** $\text{supp } x \subseteq \text{supp } P$
by *metis*
}

note $*$ = *this*
have *supp-B*: $\text{supp } ?B \subseteq \text{supp } P$
by (*rule set-bounded-supp, fact finite-supp, cut-tac *, blast*)

from *supp-B* **and** *characteristic-weak-formula-card-aux* **show** *?thesis*
using *supp-Abs-bset* **by** *blast*
qed

lemma *characteristic-weak-formula-eqvt* [*simp*]:
 $p \cdot \text{characteristic-weak-formula } P = \text{characteristic-weak-formula } (p \cdot P)$
proof –
let $?B = \{ \text{distinguishing-weak-formula } P \ Q | Q. \neg (P \equiv Q) \}$

have $\text{supp } (\text{Abs-bset } ?B \subseteq \text{supp } P$
by (*fact characteristic-weak-formula-supp-aux*)
then have *finite* ($\text{supp } (\text{Abs-bset } ?B \subseteq \text{supp } P)$)
using *finite-supp rev-finite-subset* **by** *blast*
with *characteristic-weak-formula-card-aux* **have** $*$: $p \cdot \text{Conj } (\text{Abs-bset } ?B) = \text{Conj } (\text{Abs-bset } (p \cdot ?B))$
by *simp*

let $?B' = \{ \text{distinguishing-weak-formula } (p \cdot P) \ Q | Q. \neg ((p \cdot P) \equiv Q) \}$

have $p \cdot ?B = ?B'$
proof
{
fix px
assume $px \in p \cdot ?B$
then obtain x **where** $1: px = p \cdot x$ **and** $2: x \in ?B$
by (*metis (no-types, lifting) image-iff permute-set-eq-image*)
from 2 **obtain** Q **where** $3: x = \text{distinguishing-weak-formula } P \ Q$ **and** $4: \neg (P \equiv Q)$
by *blast*
with 1 **have** $px = \text{distinguishing-weak-formula } (p \cdot P) \ (p \cdot Q)$

```

    by simp
  moreover from 4 have  $\neg (p \cdot P) \equiv (p \cdot Q)$ 
    by (metis weakly-logically-equivalent-eqv permute-minus-cancel(2))
  ultimately have  $px \in ?B'$ 
    by blast
}
then show  $p \cdot ?B \subseteq ?B'$ 
  by blast
next
{
  fix x
  assume  $x \in ?B'$ 
  then obtain Q where 1:  $x = \text{distinguishing-weak-formula } (p \cdot P) Q$  and
2:  $\neg (p \cdot P) \equiv Q$ 
    by blast
  from 2 have  $\neg P \equiv (-p \cdot Q)$ 
    by (metis weakly-logically-equivalent-eqv permute-minus-cancel(1))
  moreover from this and 1 have  $x = p \cdot \text{distinguishing-weak-formula } P$ 
(-p \cdot Q)
    by simp
  ultimately have  $x \in p \cdot ?B$ 
    using mem-permute-iff by blast
}
then show  $?B' \subseteq p \cdot ?B$ 
  by blast
qed

with * show ?thesis
  unfolding characteristic-weak-formula-def by simp
qed

lemma characteristic-weak-formula-eqv-raw [simp]:
   $p \cdot \text{characteristic-weak-formula} = \text{characteristic-weak-formula}$ 
  by (simp add: permute-fun-def)

lemma characteristic-weak-formula-is-weak:
  weak-formula (characteristic-weak-formula P)
proof -
  let  $?B = \{\text{distinguishing-weak-formula } P Q \mid Q. \neg (P \equiv Q)\}$ 

  have  $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]) \subseteq \text{supp } P$ 
    by (fact characteristic-weak-formula-supp-aux)
  then have finite (supp (Abs-bset ?B :: - set['idx']))
    using finite-supp rev-finite-subset by blast

  moreover have  $\text{set-bset } (\text{Abs-bset } ?B :: - \text{set}['idx']) = ?B$ 
    using characteristic-weak-formula-card-aux Abs-bset-inverse' by simp

  moreover

```

```

{
  fix x
  assume x ∈ ?B
  then have weak-formula x
    using distinguishing-weak-formula-is-weak by blast
}

ultimately show ?thesis
  unfolding characteristic-weak-formula-def using wf-Conj by presburger
qed

lemma characteristic-weak-formula-is-characteristic':
  Q ⊨ characteristic-weak-formula P ↔ P ≡ Q
proof -
  let ?B = {distinguishing-weak-formula P Q | Q. ¬ (P ≡ Q)}

  {
    fix P'
    have supp (Abs-bset ?B :: - set['idx]) ⊆ supp P
      by (fact characteristic-weak-formula-supp-aux)
    then have finite (supp (Abs-bset ?B :: - set['idx']))
      using finite-supp rev-finite-subset by blast
    with characteristic-weak-formula-card-aux have P' ⊨ characteristic-weak-formula
      P ↔ (∀ x ∈ ?B. P' ⊨ x)
      unfolding characteristic-weak-formula-def by simp
    }
  note valid-characteristic-formula = this

  show ?thesis
  proof
    assume *: Q ⊨ characteristic-weak-formula P
    show P ≡ Q
    proof (rule ccontr)
      assume ¬ (P ≡ Q)
      with * show False
      using distinguishing-weak-formula-distinguishes is-distinguishing-formula-def
      valid-characteristic-formula by auto
    qed
  next
    assume P ≡ Q
    moreover have P ⊨ characteristic-weak-formula P
      using distinguishing-weak-formula-distinguishes is-distinguishing-formula-def
      valid-characteristic-formula by auto
    ultimately show Q ⊨ characteristic-weak-formula P
      using weakly-logically-equivalent-def characteristic-weak-formula-is-weak by
      blast
    qed
  qed

```

lemma *characteristic-weak-formula-is-characteristic*:
 $Q \models \text{characteristic-weak-formula } P \iff P \approx \cdot Q$
using *characteristic-weak-formula-is-characteristic'* **by** (*meson weak-bisimilarity-implies-weak-equivalence*
weak-equivalence-implies-weak-bisimilarity)

26.3 Weak expressive completeness

Every finitely supported set of states that is closed under weak bisimulation can be described by a weak formula; namely, by a disjunction of characteristic weak formulas.

theorem *weak-expressive-completeness*:
assumes *finite* (*supp S*)
and $\bigwedge P Q. P \in S \implies P \approx \cdot Q \implies Q \in S$
shows $P \models \text{Disj } (\text{Abs-bset } (\text{characteristic-weak-formula } 'S)) \iff P \in S$
and *weak-formula* (*Disj* (*Abs-bset* (*characteristic-weak-formula* 'S)))
proof –
let $?B = \text{characteristic-weak-formula } 'S$

have $?B \subseteq \text{characteristic-weak-formula } 'UNIV$
by *auto*
then have $|?B| \leq o |UNIV :: 'state\ set|$
by (*rule surj-imp-ordLeq*)
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-B*: $|?B| < o \text{natLeq} + c |UNIV :: 'idx\ set|$.

have *eqvt image and eqvt characteristic-weak-formula*
by (*simp add: eqvtI*)
then have *supp-B*: $\text{supp } ?B \subseteq \text{supp } S$
using *supp-fun-eqvt supp-fun-app supp-fun-app-eqvt* **by** *blast*
with *card-B* **have** $\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]) \subseteq \text{supp } S$
using *supp-Abs-bset* **by** *blast*
with $\langle \text{finite } (\text{supp } S) \rangle$ **have** $\text{finite } (\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx]))$
using *finite-supp rev-finite-subset* **by** *blast*

with *card-B* **have** $P \models \text{Disj } (\text{Abs-bset } (\text{characteristic-weak-formula } 'S)) \iff$
 $(\exists x \in ?B. P \models x)$
by *simp*

with $\langle \bigwedge P Q. P \in S \implies P \approx \cdot Q \implies Q \in S \rangle$ **show** $P \models \text{Disj } (\text{Abs-bset}$
 $(\text{characteristic-weak-formula } 'S)) \iff P \in S$
using *characteristic-weak-formula-is-characteristic* *characteristic-weak-formula-is-characteristic'*
weakly-logically-equivalent-def **by** *fastforce*

— it remains to show that the disjunction is a weak formula

have *eqvt Formula.Not*

```

    by (simp add: eqvtI)
  with supp-B and ⟨eqvt image⟩ have supp-Not-B: supp (Formula.Not ‘ ?B) ⊆
supp S
    using supp-fun-eqvt supp-fun-app supp-fun-app-eqvt by blast

  have |Formula.Not ‘ ?B| ≤o |?B|
    by simp
  also note card-B
  finally have card-not-B: |Formula.Not ‘ ?B| <o natLeq + c |UNIV :: ‘idx set| .

  with supp-Not-B have supp (Abs-bset (Formula.Not ‘ ?B) :: - set[‘idx]) ⊆ supp
S
    using supp-Abs-bset by blast
  with ⟨finite (supp S)⟩ have finite (supp (Abs-bset (Formula.Not ‘ ?B) :: -
set[‘idx]))
    using finite-supp rev-finite-subset by blast

  moreover have ∧x. x ∈ Formula.Not ‘ ?B ⇒ weak-formula x
    using characteristic-weak-formula-is-weak wf-Not by auto

  moreover from card-B have *: map-bset Formula.Not (Abs-bset ?B :: -
set[‘idx]) = (Abs-bset (Formula.Not ‘ ?B) :: - set[‘idx])
    using map-bset.abs-eq[unfolded eq-onp-def] by blast

  moreover from card-not-B have set-bset (Abs-bset (Formula.Not ‘ ?B) :: -
set[‘idx]) = Formula.Not ‘ ?B
    by simp

  ultimately show weak-formula (Disj (Abs-bset (characteristic-weak-formula ‘
S)))
    unfolding Disj-def by (metis wf-Conj wf-Not)
  qed

end

end
theory S-Transform
imports
  Bisimilarity-Implies-Equivalence
  Equivalence-Implies-Bisimilarity
  Weak-Bisimilarity-Implies-Equivalence
  Weak-Equivalence-Implies-Bisimilarity
  Weak-Expressive-Completeness
begin

```


27 S-Transform: State Predicates as Actions

27.1 Actions and binding names

```
datatype ('act,'pred) S-action =  
  Act 'act  
  | Pred 'pred
```

```
instantiation S-action :: (pt,pt) pt  
begin
```

```
  fun permute-S-action :: perm  $\Rightarrow$  ('a,'b) S-action  $\Rightarrow$  ('a,'b) S-action where  
    p  $\cdot$  (Act  $\alpha$ ) = Act (p  $\cdot$   $\alpha$ )  
  | p  $\cdot$  (Pred  $\varphi$ ) = Pred (p  $\cdot$   $\varphi$ )
```

```
  instance
```

```
  proof
```

```
    fix x :: ('a,'b) S-action
```

```
    show 0  $\cdot$  x = x by (cases x, simp-all)
```

```
  next
```

```
    fix p q and x :: ('a,'b) S-action
```

```
    show (p + q)  $\cdot$  x = p  $\cdot$  q  $\cdot$  x by (cases x, simp-all)
```

```
  qed
```

```
end
```

```
declare permute-S-action.simps [eqvt]
```

```
lemma supp-Act [simp]: supp (Act  $\alpha$ ) = supp  $\alpha$   
unfolding supp-def by simp
```

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

```
instantiation S-action :: (fs,fs) fs  
begin
```

```
  instance
```

```
  proof
```

```
    fix x :: ('a,'b) S-action
```

```
    show finite (supp x)
```

```
    by (cases x) (simp add: finite-supp)+
```

```
  qed
```

```
end
```

```
instantiation S-action :: (bn,fs) bn  
begin
```

```
  fun bn-S-action :: ('a,'b) S-action  $\Rightarrow$  atom set where
```

```

  bn-S-action (Act  $\alpha$ ) = bn  $\alpha$ 
| bn-S-action (Pred  $-$ ) = {}

```

instance

proof

```

  fix p and  $\alpha$  :: ('a,'b) S-action
  show p · bn  $\alpha$  = bn (p ·  $\alpha$ )
  by (cases  $\alpha$ ) (simp add: bn-eqvt, simp)

```

next

```

  fix  $\alpha$  :: ('a,'b) S-action
  show finite (bn  $\alpha$ )
  by (cases  $\alpha$ ) (simp add: bn-finite, simp)

```

qed

end

27.2 Satisfaction

context *nominal-ts*

begin

Here our formalization differs from the informal presentation, where the S -transform does not have any predicates. In Isabelle/HOL, there are no empty types; we use type *unit* instead. However, it is clear from the following definition of the satisfaction relation that the single element of this type is not actually used in any meaningful way.

definition *S-satisfies* :: 'state \Rightarrow unit \Rightarrow bool (infix \vdash_S 70) **where**
 $P \vdash_S \varphi \iff \text{False}$

lemma *S-satisfies-eqvt*: **assumes** $P \vdash_S \varphi$ **shows** $(p \cdot P) \vdash_S (p \cdot \varphi)$
using *assms* **by** (simp add: *S-satisfies-def*)

end

27.3 Transitions

context *nominal-ts*

begin

inductive *S-transition* :: 'state \Rightarrow (('act,'pred) S-action, 'state) residual \Rightarrow bool
(infix \rightarrow_S 70) **where**

```

  Act:  $P \rightarrow \langle \alpha, P' \rangle \implies P \rightarrow_S \langle \text{Act } \alpha, P' \rangle$ 
| Pred:  $P \vdash \varphi \implies P \rightarrow_S \langle \text{Pred } \varphi, P \rangle$ 

```

lemma *S-transition-eqvt*: **assumes** $P \rightarrow_S \alpha_S P'$ **shows** $(p \cdot P) \rightarrow_S (p \cdot \alpha_S P')$
using *assms* **by** cases (simp add: *S-transition.Act transition-eqvt'*, simp add: *S-transition.Pred satisfies-eqvt*)

If there is an S -transition, there is an ordinary transition with the same

residual—it is not necessary to consider alpha-variants.

lemma *S-transition-cases* [case-names Act Pred, consumes 1]: **assumes** $P \rightarrow_S \langle \alpha_S, P' \rangle$
and $\bigwedge \alpha. \alpha_S = \text{Act } \alpha \implies P \rightarrow \langle \alpha, P' \rangle \implies R$
and $\bigwedge \varphi. \alpha_S = \text{Pred } \varphi \implies P' = P \implies P \vdash \varphi \implies R$
shows R
using *assms proof* (cases rule: *S-transition.cases*)
case (*Act* $\alpha' P''$)
let $?Act = \text{Act} :: 'act \Rightarrow ('act, 'pred) \text{ S-action}$
from $\langle \alpha_S, P' \rangle = \langle \text{Act } \alpha', P'' \rangle$ **obtain** α **where** $\alpha_S = \text{Act } \alpha$
by (*meson bn-S-action.elims residual-empty-bn-eq-iff*)
with $\langle \alpha_S, P' \rangle = \langle \text{Act } \alpha', P'' \rangle$ **obtain** p **where** $\text{supp } (?Act \alpha, P') - \text{bn } (?Act \alpha) = \text{supp } (?Act \alpha', P'') - \text{bn } (?Act \alpha')$
and $(\text{supp } (?Act \alpha, P') - \text{bn } (?Act \alpha)) \#* p$ **and** $p \cdot (?Act \alpha, P') = (?Act \alpha', P'')$ **and** $p \cdot \text{bn } (?Act \alpha) = \text{bn } (?Act \alpha')$
by (*auto simp add: residual-eq-iff-perm*)
then have $\text{supp } (\alpha, P') - \text{bn } \alpha = \text{supp } (\alpha', P'') - \text{bn } \alpha'$ **and** $(\text{supp } (\alpha, P') - \text{bn } \alpha) \#* p$
and $p \cdot (\alpha, P') = (\alpha', P'')$ **and** $p \cdot \text{bn } \alpha = \text{bn } \alpha'$
by (*simp-all add: supp-Pair*)
then have $\langle \alpha, P' \rangle = \langle \alpha', P'' \rangle$
by (*metis residual-eq-iff-perm*)
with $\langle \alpha_S = \text{Act } \alpha \rangle$ **and** $\langle P \rightarrow \langle \alpha', P'' \rangle \rangle$ **show** R
using $\langle \bigwedge \alpha. \alpha_S = \text{Act } \alpha \implies P \rightarrow \langle \alpha, P' \rangle \implies R \rangle$ **by** *metis*
next
case (*Pred* φ)
from $\langle \alpha_S, P' \rangle = \langle \text{Pred } \varphi, P \rangle$ **have** $\alpha_S = \text{Pred } \varphi$ **and** $P' = P$
by (*metis bn-S-action.simps(2) residual-empty-bn-eq-iff*)
with $\langle P \vdash \varphi \rangle$ **show** R
using $\langle \bigwedge \varphi. \alpha_S = \text{Pred } \varphi \implies P' = P \implies P \vdash \varphi \implies R \rangle$ **by** *metis*
qed

lemma *S-transition-Act-iff*: $P \rightarrow_S \langle \text{Act } \alpha, P' \rangle \longleftrightarrow P \rightarrow \langle \alpha, P' \rangle$
using *S-transition.Act S-transition-cases* **by** *fastforce*

lemma *S-transition-Pred-iff*: $P \rightarrow_S \langle \text{Pred } \varphi, P' \rangle \longleftrightarrow P' = P \wedge P \vdash \varphi$
using *S-transition.Pred S-transition-cases* **by** *fastforce*

end

27.4 Strong Bisimilarity in the S-transform

context *nominal-ts*

begin

interpretation *S-transform*: *nominal-ts* (\vdash_S) (\rightarrow_S)
by *unfold-locales (fact S-satisfies-eqvt, fact S-transition-eqvt)*

no-notation *S-satisfies* (**infix** \vdash_S 70) — denotes (\vdash_S) instead

notation $S\text{-transform.bisimilar}$ (**infix** \sim_S 100)

Bisimilarity is equivalent to bisimilarity in the S -transform.

lemma $\text{bisimilar-is-}S\text{-transform-bisimulation}$: $S\text{-transform.is-bisimulation bisimilar}$

unfolding $S\text{-transform.is-bisimulation-def}$

proof

show symp bisimilar

by ($\text{fact bisimilar-symp}$)

next

have $\forall P Q. P \sim \cdot Q \longrightarrow (\forall \varphi. P \vdash_S \varphi \longrightarrow Q \vdash_S \varphi)$ (**is** $?S$)

by ($\text{simp add: } S\text{-transform.S-satisfies-def}$)

moreover have $\forall P Q. P \sim \cdot Q \longrightarrow (\forall \alpha_S P'. \text{bn } \alpha_S \#* Q \longrightarrow P \rightarrow_S \langle \alpha_S, P' \rangle \longrightarrow (\exists Q'. Q \rightarrow_S \langle \alpha_S, Q' \rangle \wedge P' \sim \cdot Q'))$ (**is** $?T$)

proof (clarify)

fix $P Q \alpha_S P'$

assume $\text{bisim: } P \sim \cdot Q$ **and** $\text{fresh}_S: \text{bn } \alpha_S \#* Q$ **and** $\text{trans}_S: P \rightarrow_S \langle \alpha_S, P' \rangle$

obtain Q' **where** $Q \rightarrow_S \langle \alpha_S, Q' \rangle$ **and** $P' \sim \cdot Q'$

using trans_S **proof** ($\text{cases rule: } S\text{-transition-cases}$)

case ($\text{Act } \alpha$)

from $\langle \alpha_S = \text{Act } \alpha \rangle$ **and** fresh_S **have** $\text{bn } \alpha \#* Q$

by simp

with bisim **and** $\langle P \rightarrow \langle \alpha, P' \rangle \rangle$ **obtain** Q' **where** $\text{trans}Q: Q \rightarrow \langle \alpha, Q' \rangle$

and $\text{bisim}' : P' \sim \cdot Q'$

by ($\text{metis bisimilar-simulation-step}$)

from $\langle \alpha_S = \text{Act } \alpha \rangle$ **and** $\text{trans}Q$ **have** $Q \rightarrow_S \langle \alpha_S, Q' \rangle$

by ($\text{simp add: } S\text{-transition.Act}$)

with bisim' **show** thesis

using $\langle \wedge Q'. Q \rightarrow_S \langle \alpha_S, Q' \rangle \Longrightarrow P' \sim \cdot Q' \Longrightarrow \text{thesis} \rangle$ **by** blast

next

case ($\text{Pred } \varphi$)

from bisim **and** $\langle P \vdash \varphi \rangle$ **have** $Q \vdash \varphi$

by ($\text{metis is-bisimulation-def bisimilar-is-bisimulation}$)

with $\langle \alpha_S = \text{Pred } \varphi \rangle$ **have** $Q \rightarrow_S \langle \alpha_S, Q \rangle$

by ($\text{simp add: } S\text{-transition.Pred}$)

with bisim **and** $\langle P' = P \rangle$ **show** thesis

using $\langle \wedge Q'. Q \rightarrow_S \langle \alpha_S, Q' \rangle \Longrightarrow P' \sim \cdot Q' \Longrightarrow \text{thesis} \rangle$ **by** blast

qed

then show $\exists Q'. Q \rightarrow_S \langle \alpha_S, Q' \rangle \wedge P' \sim \cdot Q'$

by auto

qed

ultimately show $?S \wedge ?T$

by metis

qed

lemma $S\text{-transform-bisimilar-is-bisimulation}$: $\text{is-bisimulation } S\text{-transform.bisimilar}$

unfolding $\text{is-bisimulation-def}$

proof

```

show symp S-transform.bisimilar
  by (fact S-transform.bisimilar-symp)
next
have  $\forall P Q. P \sim_S Q \longrightarrow (\forall \varphi. P \vdash \varphi \longrightarrow Q \vdash \varphi)$  (is ?S)
proof (clarify)
  fix  $P Q \varphi$ 
  assume bisim:  $P \sim_S Q$  and valid:  $P \vdash \varphi$ 
  from valid have  $P \rightarrow_S \langle \text{Pred } \varphi, P \rangle$ 
    by (fact S-transition.Pred)
  moreover have  $bn (\text{Pred } \varphi) \#^* Q$ 
    by (simp add: fresh-star-def)
  ultimately obtain  $Q'$  where trans':  $Q \rightarrow_S \langle \text{Pred } \varphi, Q' \rangle$ 
    using bisim by (metis S-transform.bisimilar-simulation-step)
  from trans' show  $Q' \vdash \varphi$ 
    using S-transition-Pred-iff by blast
qed
moreover have  $\forall P Q. P \sim_S Q \longrightarrow (\forall \alpha P'. bn \alpha \#^* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow$ 
 $(\exists Q'. Q \rightarrow \langle \alpha, Q' \rangle \wedge P' \sim_S Q'))$  (is ?T)
proof (clarify)
  fix  $P Q \alpha P'$ 
  assume bisim:  $P \sim_S Q$  and fresh:  $bn \alpha \#^* Q$  and trans:  $P \rightarrow \langle \alpha, P' \rangle$ 
  from trans have  $P \rightarrow_S \langle \text{Act } \alpha, P' \rangle$ 
    by (fact S-transition.Act)
  with bisim and fresh obtain  $Q'$  where trans':  $Q \rightarrow_S \langle \text{Act } \alpha, Q' \rangle$  and
bisim':  $P' \sim_S Q'$ 
    by (metis S-transform.bisimilar-simulation-step bn-S-action.simps(1))
  from trans' have  $Q \rightarrow \langle \alpha, Q' \rangle$ 
    by (metis S-transition-Act-iff)
  with bisim' show  $\exists Q'. Q \rightarrow \langle \alpha, Q' \rangle \wedge P' \sim_S Q'$ 
    by metis
qed
ultimately show ?S  $\wedge$  ?T
  by metis
qed

theorem S-transform-bisimilar-iff:  $P \sim_S Q \longleftrightarrow P \sim Q$ 
proof
  assume  $P \sim_S Q$ 
  then show  $P \sim Q$ 
    by (metis S-transform-bisimilar-is-bisimulation bisimilar-def)
next
  assume  $P \sim Q$ 
  then show  $P \sim_S Q$ 
    by (metis S-transform.bisimilar-def bisimilar-is-S-transform-bisimulation)
qed

end

```

27.5 Weak Bisimilarity in the S -transform

context *weak-nominal-ts*

begin

lemma *weakly-bisimilar-tau-transition-weakly-bisimilar*:

assumes $P \approx \cdot R$ and $P \Rightarrow Q$ and $Q \Rightarrow R$

shows $Q \approx \cdot R$

proof –

let $?bisim = \lambda S T. S \approx \cdot T \vee \{S, T\} = \{Q, R\}$

have *is-weak-bisimulation* $?bisim$

unfolding *is-weak-bisimulation-def*

proof

show *symp* $?bisim$

using *weakly-bisimilar-symp* by (*simp add: insert-commute symp-def*)

next

have $\forall S T \varphi. ?bisim S T \wedge S \vdash \varphi \longrightarrow (\exists T'. T \Rightarrow T' \wedge ?bisim S T' \wedge T' \vdash \varphi)$ (is $?S$)

proof (*clarify*)

fix $S T \varphi$

assume *bisim*: $?bisim S T$ and *valid*: $S \vdash \varphi$

from *bisim* show $\exists T'. T \Rightarrow T' \wedge ?bisim S T' \wedge T' \vdash \varphi$

proof

assume $S \approx \cdot T$

with *valid* show *?thesis*

by (*metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation*)

next

assume $\{S, T\} = \{Q, R\}$

then have $S = Q \wedge T = R \vee T = Q \wedge S = R$

by (*metis doubleton-eq-iff*)

then show *?thesis*

proof

assume $S = Q \wedge T = R$

with $\langle P \Rightarrow Q \rangle$ and $\langle P \approx \cdot R \rangle$ and *valid* show *?thesis*

by (*metis is-weak-bisimulation-def tau-transition-trans weakly-bisimilar-is-weak-bisimulation weakly-bisimilar-tau-simulation-step*)

next

assume $T = Q \wedge S = R$

with $\langle Q \Rightarrow R \rangle$ and *valid* show *?thesis*

by (*meson reflpE weakly-bisimilar-reflp*)

qed

qed

qed

moreover have $\forall S T. ?bisim S T \longrightarrow (\forall \alpha S'. \text{bn } \alpha \#* T \longrightarrow S \rightarrow \langle \alpha, S' \rangle \longrightarrow (\exists T'. T \Rightarrow \langle \alpha \rangle T' \wedge ?bisim S' T'))$ (is $?T$)

proof (*clarify*)

fix $S T \alpha S'$

assume *bisim*: $?bisim S T$ and *fresh*: $\text{bn } \alpha \#* T$ and *trans*: $S \rightarrow \langle \alpha, S' \rangle$

from *bisim* show $\exists T'. T \Rightarrow \langle \alpha \rangle T' \wedge ?bisim S' T'$

proof

```

assume  $S \approx \cdot T$ 
with fresh and trans show ?thesis
  by (metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation)
next
assume  $\{S, T\} = \{Q, R\}$ 
then have  $S = Q \wedge T = R \vee T = Q \wedge S = R$ 
  by (metis doubleton-eq-iff)
then show ?thesis
proof
  assume  $S = Q \wedge T = R$ 
  with  $\langle P \Rightarrow Q \rangle$  and  $\langle P \approx \cdot R \rangle$  and fresh and trans show ?thesis
  using observable-transition-stepI tau-refl weak-transition-stepI weak-transition-weakI
weakly-bisimilar-weak-simulation-step by blast
next
  assume  $T = Q \wedge S = R$ 
  with  $\langle Q \Rightarrow R \rangle$  and trans show ?thesis
  by (metis observable-transition-stepI reflpE tau-refl weak-transition-stepI
weak-transition-weakI weakly-bisimilar-reflp)
  qed
qed
qed
ultimately show  $?S \wedge ?T$ 
  by metis
qed
then show ?thesis
  using weakly-bisimilar-def by blast
qed

```

notation *S-satisfies* (**infix** \vdash_S 70)

interpretation *S-transform: weak-nominal-ts* (\vdash_S) (\rightarrow_S) *Act* τ
by *unfold-locales* (*fact S-satisfies-eqvt*, *fact S-transition-eqvt*, *simp add: tau-eqvt*)

no-notation *S-satisfies* (**infix** \vdash_S 70) — denotes (\vdash_S) instead

notation *S-transform.tau-transition* (**infix** \Rightarrow_S 70)

notation *S-transform.observable-transition* ($- / \Rightarrow \{-\}_S / -$ [70, 70, 71] 71)

notation *S-transform.weak-transition* ($- / \Rightarrow \langle - \rangle_S / -$ [70, 70, 71] 71)

notation *S-transform.weakly-bisimilar* (**infix** $\approx \cdot_S$ 100)

lemma *S-transform-tau-transition-iff*: $P \Rightarrow_S P' \iff P \Rightarrow P'$

proof

assume $P \Rightarrow_S P'$

then show $P \Rightarrow P'$

by *induct* (*simp*, *metis S-transition-Act-iff tau-step*)

next

assume $P \Rightarrow P'$

then show $P \Rightarrow_S P'$

by *induct* (*simp*, *metis S-transform.tau-transition.simps S-transition.Act*)

qed

lemma *S-transform-observable-transition-iff*: $P \Rightarrow \langle \text{Act } \alpha \rangle_S P' \longleftrightarrow P \Rightarrow \langle \alpha \rangle P'$
unfolding *S-transform.observable-transition-def observable-transition-def*
by (*metis S-transform-tau-transition-iff S-transition-Act-iff*)

lemma *S-transform-weak-transition-iff*: $P \Rightarrow \langle \text{Act } \alpha \rangle_S P' \longleftrightarrow P \Rightarrow \langle \alpha \rangle P'$
by (*simp add: S-transform-observable-transition-iff S-transform-tau-transition-iff weak-transition-def*)

Weak bisimilarity is equivalent to weak bisimilarity in the *S*-transform.

lemma *weakly-bisimilar-is-S-transform-weak-bisimulation*: *S-transform.is-weak-bisimulation weakly-bisimilar*

unfolding *S-transform.is-weak-bisimulation-def*

proof

show *symp weakly-bisimilar*

by (*fact weakly-bisimilar-symp*)

next

have $\forall P Q \varphi. P \approx \cdot Q \wedge P \vdash_S \varphi \longrightarrow (\exists Q'. Q \Rightarrow_S Q' \wedge P \approx \cdot Q' \wedge Q' \vdash_S \varphi)$

(*is ?S*)

by (*simp add: S-transform.S-satisfies-def*)

moreover have $\forall P Q. P \approx \cdot Q \longrightarrow (\forall \alpha_S P'. \text{bn } \alpha_S \#^* Q \longrightarrow P \rightarrow_S \langle \alpha_S, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha_S \rangle_S Q' \wedge P' \approx \cdot Q'))$ (*is ?T*)

proof (*clarify*)

fix $P Q \alpha_S P'$

assume *bisim*: $P \approx \cdot Q$ **and** *fresh_S*: $\text{bn } \alpha_S \#^* Q$ **and** *trans_S*: $P \rightarrow_S \langle \alpha_S, P' \rangle$

obtain Q' **where** $Q \Rightarrow \langle \alpha_S \rangle_S Q'$ **and** $P' \approx \cdot Q'$

using *trans_S* **proof** (*cases rule: S-transition-cases*)

case (*Act* α)

from $\langle \alpha_S = \text{Act } \alpha \rangle$ **and** *fresh_S* **have** $\text{bn } \alpha \#^* Q$

by *simp*

with *bisim* **and** $\langle P \rightarrow \langle \alpha, P' \rangle \rangle$ **obtain** Q' **where** *trans_Q*: $Q \Rightarrow \langle \alpha \rangle Q'$

and *bisim'*: $P' \approx \cdot Q'$

by (*metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation*)

from $\langle \alpha_S = \text{Act } \alpha \rangle$ **and** *trans_Q* **have** $Q \Rightarrow \langle \alpha_S \rangle_S Q'$

by (*metis S-transform-weak-transition-iff*)

with *bisim'* **show** *thesis*

using $\langle \wedge Q'. Q \Rightarrow \langle \alpha_S \rangle_S Q' \Longrightarrow P' \approx \cdot Q' \Longrightarrow \text{thesis} \rangle$ **by** *blast*

next

case (*Pred* φ)

from *bisim* **and** $\langle P \vdash \varphi \rangle$ **obtain** Q' **where** $Q \Rightarrow Q'$ **and** $P \approx \cdot Q'$ **and**

$Q' \vdash \varphi$

by (*metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation*)

from $\langle Q \Rightarrow Q' \rangle$ **have** $Q \Rightarrow_S Q'$

by (*metis S-transform-tau-transition-iff*)

moreover from $\langle Q' \vdash \varphi \rangle$ **have** $Q' \rightarrow_S \langle \text{Pred } \varphi, Q' \rangle$

by (*simp add: S-transition.Pred*)

ultimately have $Q \Rightarrow \langle \alpha_S \rangle_S Q'$

using $\langle \alpha_S = \text{Pred } \varphi \rangle$ **by** (*metis S-transform.observable-transitionI*)

S-transform.tau-refl S-transform.weak-transition-stepI
with $\langle P' = P \rangle$ **and** $\langle P \approx \cdot Q' \rangle$ **show** *thesis*
using $\langle \wedge Q'. Q \Rightarrow \langle \alpha_S \rangle_S Q' \Longrightarrow P' \approx \cdot Q' \Longrightarrow \text{thesis} \rangle$ **by** *blast*
qed
then show $\exists Q'. Q \Rightarrow \langle \alpha_S \rangle_S Q' \wedge P' \approx \cdot Q'$
by *auto*
qed
ultimately show $?S \wedge ?T$
by *metis*
qed

lemma *S-transform-weakly-bisimilar-is-weak-bisimulation: is-weak-bisimulation*
S-transform.weakly-bisimilar
unfolding *is-weak-bisimulation-def*
proof
show *symp S-transform.weakly-bisimilar*
by (*fact S-transform.weakly-bisimilar-symp*)
next
have $\forall P Q \varphi. P \approx \cdot_S Q \wedge P \vdash \varphi \longrightarrow (\exists Q'. Q \Rightarrow Q' \wedge P \approx \cdot_S Q' \wedge Q' \vdash \varphi)$
(is $?S$ **)**
proof (*clarify*)
fix $P Q \varphi$
assume *bisim: $P \approx \cdot_S Q$ and valid: $P \vdash \varphi$*
from *valid* **have** $P \Rightarrow \langle \text{Pred } \varphi \rangle_S P$
by (*simp add: S-transition.Pred*)
moreover **have** $\text{bn } (\text{Pred } \varphi) \#^* Q$
by (*simp add: fresh-star-def*)
ultimately obtain Q'' **where** *trans': $Q \Rightarrow \langle \text{Pred } \varphi \rangle_S Q''$ and bisim': $P \approx \cdot_S Q''$*
using *bisim* **by** (*metis S-transform.weakly-bisimilar-weak-simulation-step*)

from *trans'* **obtain** $Q' Q_1$ **where** *trans₁: $Q \Rightarrow_S Q'$ and trans₂: $Q' \rightarrow_S \langle \text{Pred } \varphi, Q_1 \rangle$ and trans₃: $Q_1 \Rightarrow_S Q''$*
by (*auto simp add: S-transform.observable-transition-def*)
from *trans₂* **have** *eq: $Q_1 = Q'$ and $Q' \vdash \varphi$*
using *S-transition-Pred-iff* **by** *blast+*
moreover **from** *trans₁* **and** *trans₃* **and** *eq* **and** *bisim* **and** *bisim'* **have** $P \approx \cdot_S Q'$
by (*metis S-transform.weakly-bisimilar-equivp S-transform.weakly-bisimilar-tau-transition-weakly-bisimilar-equivp-def*)
moreover **from** *trans₁* **have** $Q \Rightarrow Q'$
by (*metis S-transform.tau-transition-iff*)
ultimately show $\exists Q'. Q \Rightarrow Q' \wedge P \approx \cdot_S Q' \wedge Q' \vdash \varphi$
by *metis*
qed
moreover **have** $\forall P Q. P \approx \cdot_S Q \longrightarrow (\forall \alpha P'. \text{bn } \alpha \#^* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'. Q \Rightarrow \langle \alpha \rangle Q' \wedge P' \approx \cdot_S Q'))$ **(is** $?T$ **)**
proof (*clarify*)
fix $P Q \alpha P'$

```

    assume bisim:  $P \approx_S Q$  and fresh:  $bn \ \alpha \ \#* \ Q$  and trans:  $P \rightarrow \langle \alpha, P \rangle$ 
    from trans have  $P \rightarrow_S \langle Act \ \alpha, P \rangle$ 
    by (fact S-transition.Act)
    with bisim and fresh obtain  $Q'$  where trans':  $Q \Rightarrow \langle Act \ \alpha \rangle_S \ Q'$  and bisim':
 $P' \approx_S Q'$ 
    by (metis S-transform.is-weak-bisimulation-def S-transform.weakly-bisimilar-is-weak-bisimulation
bn-S-action.simps(1))
    from trans' have  $Q \Rightarrow \langle \alpha \rangle \ Q'$ 
    by (metis S-transform-weak-transition-iff)
    with bisim' show  $\exists Q'. Q \Rightarrow \langle \alpha \rangle \ Q' \wedge P' \approx_S Q'$ 
    by metis
  qed
  ultimately show  $?S \wedge ?T$ 
  by metis
  qed

```

theorem *S-transform-weakly-bisimilar-iff*: $P \approx_S Q \longleftrightarrow P \approx \cdot Q$

proof

assume $P \approx_S Q$

then show $P \approx \cdot Q$

by (*metis S-transform-weakly-bisimilar-is-weak-bisimulation weakly-bisimilar-def*)

next

assume $P \approx \cdot Q$

then show $P \approx_S Q$

by (*metis S-transform.weakly-bisimilar-def weakly-bisimilar-is-S-transform-weak-bisimulation*)

qed

end

27.6 Translation of (strong) formulas into formulas without predicates

Since we defined formulas via a manual quotient construction, we also need to define the *S*-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 *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 (strong) formulas as arguments—we derive a version that returns formulas.

primrec *S-transform-Tree* :: $(\text{'idx}, \text{'pred}::\text{fs}, \text{'act}::\text{bn}) \ \text{Tree} \Rightarrow (\text{'idx}, \text{unit}, (\text{'act}, \text{'pred}) \ \text{S-action}) \ \text{Tree}_\alpha$ **where**

S-transform-Tree (*tConj tset*) = $\text{Conj}_\alpha \ (\text{map-bset } \text{S-transform-Tree } \text{tset})$

| *S-transform-Tree* (*tNot t*) = $\text{Not}_\alpha \ (\text{S-transform-Tree } \text{t})$

| *S-transform-Tree* (*tPred φ*) = $\text{Act}_\alpha \ (\text{S-action.Pred } \varphi) \ (\text{Conj}_\alpha \ \text{bempty})$

| $S\text{-transform-Tree } (tAct \alpha t) = Act_\alpha (S\text{-action.Act } \alpha) (S\text{-transform-Tree } t)$

lemma $S\text{-transform-Tree-eqt}$ [eqvt]: $p \cdot S\text{-transform-Tree } t = S\text{-transform-Tree } (p \cdot t)$

proof (induct t)

case (tConj tset)

then show ?case

by simp (metis (no-types, opaque-lifting) bset.map-cong0 map-bset-eqt permute-fun-def permute-minus-cancel(1))

qed simp-all

$S\text{-transform-Tree}$ respects α -equivalence.

lemma $alpha\text{-Tree-}S\text{-transform-Tree}$:

assumes $t1 =_\alpha t2$

shows $S\text{-transform-Tree } t1 = S\text{-transform-Tree } t2$

using *assms* **proof** (induction t1 t2 rule: $alpha\text{-Tree-induct}'$)

case ($alpha\text{-tConj } tset1 tset2$)

then have $rel\text{-bset } (=)$ ($map\text{-bset } S\text{-transform-Tree } tset1$) ($map\text{-bset } S\text{-transform-Tree } tset2$)

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 ($alpha\text{-tAct } \alpha1 t1 \alpha2 t2$)

from $\langle tAct \alpha1 t1 =_\alpha tAct \alpha2 t2 \rangle$

obtain p **where** $*$: ($bn \alpha1, t1$) $\approx_{set} alpha\text{-Tree } (supp\text{-rel } alpha\text{-Tree}) p$ ($bn \alpha2, t2$)

and $**$: ($bn \alpha1, \alpha1$) $\approx_{set} (=)$ $supp p$ ($bn \alpha2, \alpha2$)

by auto

from $*$ **have** $fresh$: ($supp\text{-rel } alpha\text{-Tree } t1 - bn \alpha1$) $\sharp^* p$ **and** $alpha$: $p \cdot t1 =_\alpha t2$ **and** eq : $p \cdot bn \alpha1 = bn \alpha2$

by (auto simp add: $alpha\text{-set}$)

from $alpha\text{-tAct.IH}(2)$ **have** $supp\text{-rel } alpha\text{-Tree } (rep\text{-Tree}_\alpha (S\text{-transform-Tree } t1)) \subseteq supp\text{-rel } alpha\text{-Tree } t1$

by (metis (no-types, lifting) infinite-mono $alpha\text{-Tree-permute-rep-commute } S\text{-transform-Tree-eqt } mem\text{-Collect-eq } subsetI supp\text{-rel-def}$)

with $fresh$ **have** $fresh'$: ($supp\text{-rel } alpha\text{-Tree } (rep\text{-Tree}_\alpha (S\text{-transform-Tree } t1)) - bn \alpha1$) $\sharp^* p$

by (meson DiffD1 DiffD2 DiffI $fresh\text{-star-def } subsetCE$)

moreover from $alpha$ **have** $alpha'$: $p \cdot rep\text{-Tree}_\alpha (S\text{-transform-Tree } t1) =_\alpha rep\text{-Tree}_\alpha (S\text{-transform-Tree } t2)$

using $alpha\text{-tAct.IH}(1)$ **by** (metis $alpha\text{-Tree-permute-rep-commute } S\text{-transform-Tree-eqt}$)

moreover from $fresh'$ $alpha'$ eq **have** $supp\text{-rel } alpha\text{-Tree } (rep\text{-Tree}_\alpha (S\text{-transform-Tree } t1)) - bn \alpha1 = supp\text{-rel } alpha\text{-Tree } (rep\text{-Tree}_\alpha (S\text{-transform-Tree } t2)) - bn \alpha2$

by (metis (mono-tags) Diff-eqt $alpha\text{-Tree-eqt}'$ $alpha\text{-Tree-eqt-aux } alpha\text{-Tree-supp-rel } atom\text{-set-perm-eq}$)

ultimately have ($bn \alpha1, rep\text{-Tree}_\alpha (S\text{-transform-Tree } t1)$) $\approx_{set} alpha\text{-Tree } (supp\text{-rel } alpha\text{-Tree}) p$ ($bn \alpha2, rep\text{-Tree}_\alpha (S\text{-transform-Tree } t2)$)

using eq **by** (simp add: $alpha\text{-set}$)

moreover from ** have $(bn\ \alpha 1, S\text{-action.Act}\ \alpha 1) \approx_{set} (=) \text{supp } p\ (bn\ \alpha 2,$
 $S\text{-action.Act}\ \alpha 2)$
by $(metis\ (mono\text{-tags},\ lifting)\ S\text{-Transform.supp-Act}\ \alpha\text{-set}\ \text{permute-}S\text{-action.simps}(1))$
ultimately have $Act_\alpha\ (S\text{-action.Act}\ \alpha 1)\ (S\text{-transform-Tree}\ t1) = Act_\alpha\ (S\text{-action.Act}$
 $\alpha 2)\ (S\text{-transform-Tree}\ t2)$
by $(auto\ simp\ add:\ Act_\alpha\text{-eq-iff})$
then show $?case$
by $simp$
qed $simp\text{-all}$

S -transform for trees modulo α -equivalence.

lift-definition $S\text{-transform-Tree}_\alpha :: ('idx, 'pred :: fs, 'act :: bn)\ Tree_\alpha \Rightarrow ('idx, unit,$
 $('act, 'pred)\ S\text{-action})\ Tree_\alpha$ **is**
 $S\text{-transform-Tree}$
by $(fact\ \alpha\text{-Tree-}S\text{-transform-Tree})$

lemma $S\text{-transform-Tree}_\alpha\text{-eqvt}\ [eqvt]: p \cdot S\text{-transform-Tree}_\alpha\ t_\alpha = S\text{-transform-Tree}_\alpha$
 $(p \cdot t_\alpha)$
by $transfer\ (simp)$

lemma $S\text{-transform-Tree}_\alpha\text{-Conj}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Conj_\alpha\ tset_\alpha) = Conj_\alpha$
 $(map\ bset\ S\text{-transform-Tree}_\alpha\ tset_\alpha)$
by $(simp\ add:\ Conj_\alpha\text{-def}'\ S\text{-transform-Tree}_\alpha.\text{abs-eq})\ (metis\ (no\text{-types},\ lifting)\ S\text{-transform-Tree}_\alpha.\text{rep-eq}\ bset.\text{map-comp}\ bset.\text{map-cong0}\ \text{comp-apply})$

lemma $S\text{-transform-Tree}_\alpha\text{-Not}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Not_\alpha\ t_\alpha) = Not_\alpha\ (S\text{-transform-Tree}_\alpha$
 $t_\alpha)$
by $transfer\ simp$

lemma $S\text{-transform-Tree}_\alpha\text{-Pred}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Pred_\alpha\ \varphi) = Act_\alpha$
 $(S\text{-action.Pred}\ \varphi)\ (Conj_\alpha\ \text{bempty})$
by $transfer\ simp$

lemma $S\text{-transform-Tree}_\alpha\text{-Act}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Act_\alpha\ \alpha\ t_\alpha) = Act_\alpha$
 $(S\text{-action.Act}\ \alpha)\ (S\text{-transform-Tree}_\alpha\ t_\alpha)$
by $transfer\ simp$

lemma $finite\text{-supp-map-bset-}S\text{-transform-Tree}_\alpha\ [simp]:$
assumes $finite\ (supp\ tset_\alpha)$
shows $finite\ (supp\ (map\ bset\ S\text{-transform-Tree}_\alpha\ tset_\alpha))$

proof –

have $eqvt\ map\text{-bset}\ \text{and}\ eqvt\ S\text{-transform-Tree}_\alpha$
by $(simp\ add:\ eqvtI)+$
then have $supp\ (map\ bset\ S\text{-transform-Tree}_\alpha) = \{\}$
using $supp\text{-fun-}eqvt\ supp\text{-fun-app-}eqvt$ **by** $blast$
then have $supp\ (map\ bset\ S\text{-transform-Tree}_\alpha\ tset_\alpha) \subseteq supp\ tset_\alpha$
using $supp\text{-fun-app}$ **by** $blast$
with $assms$ **show** $finite\ (supp\ (map\ bset\ S\text{-transform-Tree}_\alpha\ tset_\alpha))$
by $(metis\ finite\text{-subset})$

qed

lemma *S-transform-Tree $_{\alpha}$ -preserves-hereditarily-fs*:
 assumes *hereditarily-fs t $_{\alpha}$*
 shows *hereditarily-fs (S-transform-Tree $_{\alpha}$ t $_{\alpha}$)*
using *assms proof (induct rule: hereditarily-fs.induct)*
 case (*Conj $_{\alpha}$ tset $_{\alpha}$*)
 then show *?case*
 by (*auto intro!: hereditarily-fs.Conj $_{\alpha}$*) (*metis imageE map-bset.rep-eq*)
next
 case (*Not $_{\alpha}$ t $_{\alpha}$*)
 then show *?case*
 by (*simp add: hereditarily-fs.Not $_{\alpha}$*)
next
 case (*Pred $_{\alpha}$ φ*)
 have *finite (supp bempty)*
 by (*simp add: eqvtI supp-fun-eqvt*)
 then show *?case*
 using *hereditarily-fs.Act $_{\alpha}$ finite-supp-implies-hereditarily-fs-Conj $_{\alpha}$* **by** *fastforce*
next
 case (*Act $_{\alpha}$ t $_{\alpha}$ α*)
 then show *?case*
 by (*simp add: Formula.hereditarily-fs.Act $_{\alpha}$*)
qed

S-transform for (strong) formulas.

lift-definition *S-transform-formula* :: (*'idx, 'pred::fs, 'act::bn*) *formula* \Rightarrow (*'idx, unit,*
'act, 'pred) *S-action*) *Tree $_{\alpha}$ is*
 S-transform-Tree $_{\alpha}$
 .

lemma *S-transform-formula-eqvt [eqvt]: p \cdot S-transform-formula x = S-transform-formula*
(p \cdot x)
 by *transfer (simp)*

lemma *S-transform-formula-Conj [simp]:*
 assumes *finite (supp xset)*
 shows *S-transform-formula (Conj xset) = Conj $_{\alpha}$ (map-bset S-transform-formula*
xset)
 using *assms by (simp add: Conj-def S-transform-formula-def bset.map-comp*
map-fun-def)

lemma *S-transform-formula-Not [simp]: S-transform-formula (Not x) = Not $_{\alpha}$ (S-transform-formula*
x)
 by *transfer simp*

lemma *S-transform-formula-Pred [simp]: S-transform-formula (Formula.Pred φ)*
= Act $_{\alpha}$ (S-action.Pred φ) (Conj $_{\alpha}$ bempty)
 by *transfer simp*

lemma *S-transform-formula-Act* [simp]: *S-transform-formula* (*Formula.Act* α x)
 = *Formula.Act* _{α} (*S-action.Act* α) (*S-transform-formula* x)
by *transfer simp*

lemma *S-transform-formula-hereditarily-fs* [simp]: *hereditarily-fs* (*S-transform-formula* x)
by *transfer (fact S-transform-Tree $_{\alpha}$ -preserves-hereditarily-fs)*

Finally, we define the proper *S*-transform, which returns formulas instead of trees.

definition *S-transform* :: ('idx, 'pred::fs, 'act::bn) *formula* \Rightarrow ('idx, unit, ('act, 'pred) *S-action*) *formula* **where**
S-transform x = *Abs-formula* (*S-transform-formula* x)

lemma *S-transform-eqvt* [eqvt]: $p \cdot$ *S-transform* x = *S-transform* ($p \cdot x$)
unfolding *S-transform-def* **by** *simp*

lemma *finite-supp-map-bset-S-transform* [simp]:
assumes *finite* (*supp* $xset$)
shows *finite* (*supp* (*map-bset* *S-transform* $xset$))

proof –

have *eqvt map-bset and eqvt S-transform*
by (*simp add: eqvtI*)
then have *supp (map-bset S-transform) = {}*
using *supp-fun-eqvt supp-fun-app-eqvt* **by** *blast*
then have *supp (map-bset S-transform $xset$) \subseteq supp $xset$*
using *supp-fun-app* **by** *blast*
with *assms* **show** *finite (supp (map-bset S-transform $xset$))*
by (*metis finite-subset*)

qed

lemma *S-transform-Conj* [simp]:
assumes *finite* (*supp* $xset$)
shows *S-transform (Conj $xset$) = Conj (map-bset S-transform $xset$)*
using *assms* **unfolding** *S-transform-def* **by** (*simp, simp add: Conj-def bset.map-comp o-def*)

lemma *S-transform-Not* [simp]: *S-transform (Not x) = Not (S-transform x)*
unfolding *S-transform-def* **by** (*simp add: Not.abs-eq eq-onp-same-args*)

lemma *S-transform-Pred* [simp]: *S-transform (Formula.Pred φ) = Formula.Act*
(S-action.Pred φ) (Conj bempty)
unfolding *S-transform-def* **by** (*simp add: Formula.Act-def Conj-rep-eq eqvtI supp-fun-eqvt*)

lemma *S-transform-Act* [simp]: *S-transform (Formula.Act α x) = Formula.Act*
(S-action.Act α) (S-transform x)
unfolding *S-transform-def* **by** (*simp, simp add: Formula.Act-def*)

context *nominal-ts*

begin

lemma *valid-Conj-bempty* [*simp*]: $P \models \text{Conj } \text{bempty}$
by (*simp add: bempty.rep-eq eqvtI supp-fun-eqvt*)

notation *S-satisfies* (**infix** \vdash_S 70)

interpretation *S-transform*: *nominal-ts* $(\vdash_S) (\rightarrow_S)$
by *unfold-locales (fact S-satisfies-eqvt, fact S-transition-eqvt)*

notation *S-transform.valid* (**infix** \models_S 70)

The *S*-transform preserves satisfaction of formulas in the following sense:

theorem *valid-iff-valid-S-transform*: **shows** $P \models x \longleftrightarrow P \models_S \text{S-transform } x$
proof (*induct x arbitrary: P*)
 case (*Conj xset*)
 then show *?case*
 by *auto (metis imageE map-bset.rep-eq, simp add: map-bset.rep-eq)*
 next
 case (*Not x*)
 then show *?case by simp*
 next
 case (*Pred φ*)
 let *? $\varphi = \text{Formula.Pred } \varphi :: ('idx, 'pred, ('act, 'pred) \text{S-action}) \text{formula}$*
 have *bn (S-action.Pred $\varphi :: ('act, 'pred) \text{S-action}) \#* P$*
 by (*simp add: fresh-star-def*)
 then show *?case*
 by (*auto simp add: S-transform.valid-Act-fresh S-transition-Pred-iff*)
 next
 case (*Act αx*)
 show *?case*
 proof
 assume $P \models \text{Formula.Act } \alpha x$
 then obtain $\alpha' x' P'$ **where** *eq: Formula.Act $\alpha x = \text{Formula.Act } \alpha' x'$ and*
trans: $P \rightarrow \langle \alpha', P' \rangle$ and valid: $P' \models x'$
 by (*metis valid-Act*)
 from *eq obtain* p **where** *p-x: $p \cdot x = x'$ and p- α : $p \cdot \alpha = \alpha'$*
 by (*metis Act-eq-iff-perm*)

 from *valid have* $-p \cdot P' \models x$
 using *p-x by (metis valid-eqvt permute-minus-cancel(2))*
 then have $-p \cdot P' \models_S \text{S-transform } x$
 using *Act.hyps(1) by metis*
 then have $P' \models_S \text{S-transform } x'$
 by (*metis (no-types, lifting) p-x S-transform.valid-eqvt S-transform-eqvt permute-minus-cancel(1)*)

```

with eq and trans show  $P \models_S S\text{-transform } (Formula.Act \alpha x)$ 
  using S-transform.valid-Act S-transition.Act by fastforce
next
assume  $*$ :  $P \models_S S\text{-transform } (Formula.Act \alpha x)$ 

  — rename bn  $\alpha$  to avoid  $P$ , without touching Formula.Act  $\alpha x$ 
obtain  $p$  where  $1: (p \cdot bn \alpha) \#* P$  and  $2: supp (Formula.Act \alpha x) \#* p$ 
proof (rule at-set-avoiding2[of bn  $\alpha$  P Formula.Act  $\alpha x$ , THEN  $exE$ ])
  show finite (bn  $\alpha$ ) by (fact bn-finite)
next
  show finite (supp  $P$ ) by (fact finite-supp)
next
  show finite (supp (Formula.Act  $\alpha x$ )) by (fact finite-supp)
next
  show  $bn \alpha \#* Formula.Act \alpha x$  by simp
qed metis
from  $2$  have eq:  $Formula.Act \alpha x = Formula.Act (p \cdot \alpha) (p \cdot x)$ 
  using supp-perm-eq by fastforce

with  $*$  have  $P \models_S Formula.Act (S\text{-action.Act } (p \cdot \alpha)) (S\text{-transform } (p \cdot x))$ 
  by simp
with  $1$  obtain  $P'$  where trans:  $P \rightarrow_S \langle S\text{-action.Act } (p \cdot \alpha), P' \rangle$  and valid:
 $P' \models_S S\text{-transform } (p \cdot x)$ 
  by (metis S-transform.valid-Act-fresh bn-S-action.simps(1) bn-eqvt)

from valid have  $-p \cdot P' \models_S S\text{-transform } x$ 
  by (metis (no-types, opaque-lifting) S-transform.valid-eqvt S-transform-eqvt
permute-minus-cancel(1))
then have  $-p \cdot P' \models x$ 
  using Act.hyps(1) by metis
then have  $P' \models p \cdot x$ 
  by (metis permute-minus-cancel(1) valid-eqvt)

moreover from trans have  $P \rightarrow \langle p \cdot \alpha, P' \rangle$ 
  using S-transition-Act-iff by blast

ultimately show  $P \models Formula.Act \alpha x$ 
  using eq valid-Act by blast
qed
qed
end

context indexed-nominal-ts
begin

```

The following (alternative) proof of the “ \rightarrow ” direction of theorem *nominal-ts.bisimilar* (\vdash_S) $(\rightarrow_S) ?P ?Q = ?P \sim ?Q$, namely that bisimilarity in the S -transform implies bisimilarity in the original transition system,

uses the fact that the S -transform(ation) preserves satisfaction of formulas, together with the fact that bisimilarity (in the S -transform) implies logical equivalence, and equivalence (in the original transition system) implies bisimilarity. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system.

interpretation S -transform: *indexed-nominal-ts* (\vdash_S) (\rightarrow_S)

by *unfold-locales* (*fact S-satisfies-eqvt*, *fact S-transition-eqvt*, *fact card-idx-perm*, *fact card-idx-state*)

notation S -transform.bisimilar (**infix** \sim_S 100)

theorem $P \sim_S Q \longrightarrow P \sim \cdot Q$

proof

assume $P \sim_S Q$

then have S -transform.logically-equivalent $P Q$

by (*fact S-transform.bisimilarity-implies-equivalence*)

with *valid-iff-valid-S-transform* **have** logically-equivalent $P Q$

using *logically-equivalent-def S-transform.logically-equivalent-def* **by** *blast*

then show $P \sim \cdot Q$

by (*fact equivalence-implies-bisimilarity*)

qed

end

27.7 Translation of weak formulas into formulas without predicates

context *indexed-weak-nominal-ts*

begin

notation S -satisfies (**infix** \vdash_S 70)

interpretation S -transform: *indexed-weak-nominal-ts S-action.Act* τ (\vdash_S) (\rightarrow_S)

by *unfold-locales* (*fact S-satisfies-eqvt*, *fact S-transition-eqvt*, *simp add: tau-eqvt*, *fact card-idx-perm*, *fact card-idx-state*, *fact card-idx-nat*)

notation S -transform.valid (**infix** \models_S 70)

notation S -transform.weakly-bisimilar (**infix** \approx_S 100)

The S -transform of a weak formula is not necessarily a weak formula. However, the image of all weak formulas under the S -transform is adequate for weak bisimilarity.

corollary $P \approx_S Q \iff (\forall x. \text{weak-formula } x \longrightarrow P \models_S S\text{-transform } x \iff Q \models_S S\text{-transform } x)$

by (*meson valid-iff-valid-S-transform weak-bisimilarity-implies-weak-equivalence weak-equivalence-implies-weak-bisimilarity S-transform-weakly-bisimilar-iff weakly-logically-equivalent-def*)

For every weak formula, there is an equivalent weak formula over the S -transform.

```

corollary
  assumes weak-formula  $x$ 
  obtains  $y$  where  $S\text{-transform.weak-formula } y$  and  $\forall P. P \models x \longleftrightarrow P \models_S y$ 
proof –
  let  $?S = \{P. P \models x\}$ 

  –  $\{P. P \models x\}$  is finitely supported
  have  $\text{supp } x \text{ supports } ?S$ 
  unfolding supports-def proof (clarify)
  fix  $a\ b$ 
  assume  $a: a \notin \text{supp } x$  and  $b: b \notin \text{supp } x$ 
  {
    fix  $P$ 
    from  $a$  and  $b$  have  $(a \rightleftharpoons b) \cdot x = x$ 
      by (simp add: fresh-def swap-fresh-fresh)
    then have  $(a \rightleftharpoons b) \cdot P \models x \longleftrightarrow P \models x$ 
      by (metis permute-swap-cancel valid-evt)
  }
  note  $* = \text{this}$ 
  show  $(a \rightleftharpoons b) \cdot ?S = ?S$ 
  by auto (metis mem-Collect-eq mem-permute-iff permute-swap-cancel *, simp
add: Collect-evt permute-fun-def *)
  qed
  then have finite (supp ?S)
  using finite-supp supports-finite by blast

  –  $\{P. P \models x\}$  is closed under weak bisimilarity
  moreover {
    fix  $P\ Q$ 
    assume  $P \in ?S$  and  $P \approx_S Q$ 
    with  $\langle \text{weak-formula } x \rangle$  have  $Q \in ?S$ 
    using S-transform-weakly-bisimilar-iff weak-bisimilarity-implies-weak-equivalence
weakly-logically-equivalent-def by auto
  }

  ultimately show ?thesis
  using S-transform.weak-expressive-completeness that by (metis (no-types,
lifting) mem-Collect-eq)
  qed

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.