Modal Logics for Nominal Transition Systems

Tjark Weber et al.

July 20, 2018

Abstract

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

Contents

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

1 Bounded Sets Equipped With a Permutation Action

Additional lemmas about bounded sets.

interpretation *bset-lifting*: *bset-lifting* **.**

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

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

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

lift-definition *permute-bset* :: *perm* ⇒ *′a set*[*′b*] ⇒ *′a set*[*′b*] **is**
permute
proof −
fix *p* **and** *A* :: *′a set*
have $|p \cdot A| \leq o |A|$ **by** (*simp add*: *permute-set-eq-image*)
also assume $|A| <o natLeq +c |UNIV :: 'b\ set|$
finally show $|p \cdot A| <o natLeq +c |UNIV :: 'b\ set|$ **.**
qed

instance
by *standard* (*transfer*, *simp*)+

end

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

lemma *supp-Abs-bset* [*simp*]:
assumes $|A| <o natLeq +c |UNIV :: 'k\ set|$
shows *supp* (*Abs-bset A* :: *′a*::*pt set*[*′k*]) = *supp A*
proof −
from *assms* **have** $\bigwedge p.\ p \cdot$ (*Abs-bset A* :: *′a*::*pt set*[*′k*]) ≠ *Abs-bset A* ⟷ *p* · *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*: $p \cdot B = map\text{-}bset\ (permute\ p)\ B$
by *transfer* (*auto simp add*: *image-def permute-set-def*)

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

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

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

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

lemma *bsingleton-eqvt* [*eqvt*]: $p \cdot bsingleton\ x = bsingleton\ (p \cdot x)$
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**
less-bool-rel $\equiv \{(x,y).\ x<y\}$

lemma *less-bool-rel-iff* [*simp*]:
$(a,b) \in less\text{-}bool\text{-}rel \longleftrightarrow \neg\ a \wedge b$
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**

$(p \cdot x,\ x) \in$ *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 O hull-rel* $\subseteq$ *hull-rel O R*
proof
fix *x*
assume $x \in R\ O$ *hull-rel*
then obtain *x1 x2 y* **where** *x*: $x = (x1,x2)$ **and** *R*: $(x1,y) \in R$ **and** $(y,x2) \in$ *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 $(x1,\ -p \cdot x1) \in$ *hull-rel*
by (*metis hull-rel.intros permute-minus-cancel*(*2*))
ultimately show $x \in$ *hull-rel O R*
using *x* **by** *auto*
qed

lemma *wf-hull-rel-relcomp*:
assumes *wf R* **and** *eqvt R*
shows *wf* (*hull-rel O R*)
using *assms* **by** (*metis hull-rel-relcomp-subset wf-relcomp-compatible*)

lemma *hull-rel-relcompI* [*simp*]:
assumes $(x,\ y) \in R$
shows $(p \cdot x,\ y) \in$ *hull-rel O R*
using *assms* **by** (*metis hull-rel.intros relcomp.relcompI*)

lemma *hull-rel-relcomp-trivialI* [*simp*]:
assumes $(x,\ y) \in R$
shows $(x,\ y) \in$ *hull-rel O 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* ⇒ *atom set*
assumes *bn-eqvt*: *p* ∙ (*bn* α) = *bn* (*p* ∙ α)
and *bn-finite*: *finite* (*bn* α)

lemma *bn-subset-supp*: *bn* α ⊆ *supp* α
by (*metis* (*erased*, *hide-lams*) *bn-eqvt bn-finite eqvt-at-def finite-supp supp-eqvt-at supp-finite-atom-set*)

3.2 Raw residuals and α-equivalence

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

fun *alpha-residual* :: (*'act*::*bn* × *'state*::*pt*) ⇒ (*'act* × *'state*) ⇒ *bool* **where**
alpha-residual (α*1*,*P1*) (α*2*,*P2*) ⟷ [*bn* α*1*]*set*. (α*1*, *P1*) = [*bn* α*2*]*set*. (α*2*, *P2*)

α-equivalence is equivariant.

lemma *alpha-residual-eqvt* [*eqvt*]:
assumes *alpha-residual r1 r2*
shows *alpha-residual* (*p* ∙ *r1*) (*p* ∙ *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* ⇒ (*'a*,*'b*) *residual* ⇒ (*'a*,*'b*) *residual*
is *permute*
by (*fact alpha-residual-eqvt*)

instance
proof
fix *res* :: (-,-) *residual*
show *0* • *res* = *res*
by *transfer* (*metis alpha-residual-equivp equivp-reflp permute-zero*)
next
fix *p q* :: *perm* **and** *res* :: (-,-) *residual*
show (*p* + *q*) • *res* = *p* • *q* • *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* ⇒ *'state::pt* ⇒ (*'act*,*'state*) *residual*
(⟨-,-⟩ [*0*,*0*] *1000*)
where
⟨α,*P*⟩ == *abs-residual* (α,*P*)

lemma *abs-residual-pair-eqvt* [*simp*]: *p* ∙ ⟨α,*P*⟩ = ⟨*p* ∙ α, *p* ∙ *P*⟩
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*: *supp* ⟨α, *P::'state::fs*⟩ = *supp* (α,*P*) − *bn* α
proof −
have *supp* ⟨α,*P*⟩ = *supp* ([*bn* α]*set*. (α, *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*]: *bn* α ♯* ⟨α,*P::'state::fs*⟩
by (*simp add*: *fresh-star-def fresh-def supp-abs-residual-pair*)

lemma *finite-supp-abs-residual-pair* [*simp*]: *finite* (*supp* ⟨α, *P::'state::fs*⟩)
by (*metis finite-Diff finite-supp supp-abs-residual-pair*)

3.6 Equality between residuals

lemma *residual-eq-iff-perm*: ⟨α*1*,*P1*⟩ = ⟨α*2*,*P2*⟩ ⟷
(∃ *p*. *supp* (α*1*, *P1*) − *bn* α*1* = *supp* (α*2*, *P2*) − *bn* α*2* ∧ (*supp* (α*1*, *P1*) − *bn* α*1*) ♯* *p* ∧ *p* ∙ (α*1*, *P1*) = (α*2*, *P2*) ∧ *p* ∙ *bn* α*1* = *bn* α*2*)
(**is** *?l* ⟷ *?r*)
proof
assume ∗: *?l*
then have [*bn* α*1*]*set*. (α*1*, *P1*) = [*bn* α*2*]*set*. (α*2*, *P2*)
by (*simp add*: *residual.abs-eq-iff*)
then obtain *p* **where** (*bn* α*1*, (α*1*,*P1*)) ≈*set* ((=)) *supp p* (*bn* α*2*, (α*2*,*P2*))
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** (*bn* α*1*, (α*1*,*P1*)) ≈*set* ((=)) *supp p* (*bn* α*2*, (α*2*,*P2*))
using *alpha-set.simps* **by** *blast*
then have [*bn* α*1*]*set*. (α*1*, *P1*) = [*bn* α*2*]*set*. (α*2*, *P2*)
using *Abs-eq-iff*(*1*) **by** *blast*
then show *?l*
by (*simp add*: *residual.abs-eq-iff*)
qed

lemma *residual-eq-iff-perm-renaming*: $\langle \alpha 1, P1 \rangle = \langle \alpha 2, P2 \rangle \longleftrightarrow$
$(\exists p.\ supp\ (\alpha 1,\ P1) - bn\ \alpha 1 = supp\ (\alpha 2,\ P2) - bn\ \alpha 2 \wedge (supp\ (\alpha 1,\ P1) - bn\ \alpha 1) \sharp* p \wedge p \cdot (\alpha 1,\ P1) = (\alpha 2,\ P2) \wedge p \cdot bn\ \alpha 1 = bn\ \alpha 2 \wedge supp\ p \subseteq bn\ \alpha 1 \cup p \cdot bn\ \alpha 1)$
(**is** $?l \longleftrightarrow ?r$)
proof
assume *?l*
then obtain p **where** p: $supp\ (\alpha 1,\ P1) - bn\ \alpha 1 = supp\ (\alpha 2,\ P2) - bn\ \alpha 2 \wedge (supp\ (\alpha 1,\ P1) - bn\ \alpha 1) \sharp* p \wedge p \cdot (\alpha 1,\ P1) = (\alpha 2,\ P2) \wedge p \cdot bn\ \alpha 1 = bn\ \alpha 2$
by (*metis residual-eq-iff-perm*)
moreover obtain q **where** *q-p*: $\forall b \in bn\ \alpha 1.\ q \cdot b = p \cdot b$ **and** *supp-q*: $supp\ q \subseteq bn\ \alpha 1 \cup p \cdot bn\ \alpha 1$
by (*metis set-renaming-perm2*)
have $supp\ q \subseteq supp\ p$
proof
fix a **assume** $*$: $a \in supp\ q$ **then show** $a \in supp\ p$
proof (*cases* $a \in bn\ \alpha 1$)
case *True* **then show** *?thesis*
using $*$ *q-p* **by** (*metis mem-Collect-eq supp-perm*)
next
case *False* **then have** $a \in p \cdot bn\ \alpha 1$
using $*$ *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** $(supp\ (\alpha 1,\ P1) - bn\ \alpha 1) \sharp* q$
by (*meson fresh-def fresh-star-def subset-iff*)
moreover with p **and** *q-p* **have** $\bigwedge a.\ a \in supp\ \alpha 1 \Longrightarrow q \cdot a = p \cdot a$ **and** $\bigwedge a.\ a \in supp\ P1 \Longrightarrow q \cdot a = p \cdot a$
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 *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 $\bigwedge (act::'act::bn)\ (state::'state::fs)\ (c::'a::fs).\ bn\ act \sharp* c \Longrightarrow P\ c\ \langle act, state \rangle$
shows $P\ c\ residual$
proof (*rule residual.abs-induct*, *clarify*)

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

3.8 Other lemmas

lemma *residual-empty-bn-eq-iff*:
assumes *bn* $\alpha 1 = \{\}$
shows $\langle \alpha 1,P1\rangle = \langle \alpha 2,P2\rangle \longleftrightarrow \alpha 1 = \alpha 2 \wedge P1 = P2$
proof
assume $\langle \alpha 1,P1\rangle = \langle \alpha 2,P2\rangle$
with *assms* **have** $[\{\}]set.\ (\alpha 1,\ P1) = [bn\ \alpha 2]set.\ (\alpha 2,\ P2)$
by (*simp add*: *residual.abs-eq-iff*)
then obtain *p* **where** $(\{\},\ (\alpha 1,\ P1)) \approx set\ ((=))\ supp\ p\ (bn\ \alpha 2,\ (\alpha 2,\ P2))$
using *Abs-eq-iff*(*1*) **by** *blast*
then show $\alpha 1 = \alpha 2 \wedge P1 = P2$
unfolding *alpha-set* **using** *supp-perm-eq* **by** *fastforce*
next
assume $\alpha 1 = \alpha 2 \wedge P1 = P2$ **then show** $\langle \alpha 1,P1\rangle = \langle \alpha 2,P2\rangle$
by *simp*
qed

end
theory *Transition-System*
imports
Residual
begin

4 Nominal Transition Systems and Bisimulations

4.1 Basic Lemmas

lemma *symp-eqvt* [*eqvt*]:
assumes *symp R* **shows** *symp* $(p \cdot R)$
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* ⇒ *'pred::fs* ⇒ *bool* (**infix** ⊢ *70*)
    and *transition* :: *'state* ⇒ (*'act::bn*,*'state*) *residual* ⇒ *bool* (**infix** → *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* ⇒ *'state* ⇒ *bool*) ⇒ *bool* **where**
    is-bisimulation R ≡
      symp R ∧
      (∀ *P Q*. *R P Q* ⟶ (∀ φ. *P* ⊢ φ ⟶ *Q* ⊢ φ)) ∧
       (∀ *P Q*. *R P Q* ⟶ (∀ α *P'*. *bn* α ♯* *Q* ⟶ *P* → ⟨α,*P'*⟩ ⟶ (∃ *Q'*. *Q* → ⟨α,*Q'*⟩ ∧ *R P' Q'*)))

  definition *bisimilar* :: *'state* ⇒ *'state* ⇒ *bool* (**infix** ∼· *100*) **where**
    P ∼· *Q* ≡ ∃ *R*. *is-bisimulation R* ∧ *R P Q*

(∼·) is an equivariant equivalence relation.

  lemma *is-bisimulation-eqvt* :
    assumes *is-bisimulation R* **shows** *is-bisimulation* (*p* · *R*)
  using *assms* **unfolding** *is-bisimulation-def*
  proof (*clarify*)
    assume *1*: *symp R*
    assume *2*: ∀ *P Q*. *R P Q* ⟶ (∀ φ. *P* ⊢ φ ⟶ *Q* ⊢ φ)
    assume *3*: ∀ *P Q*. *R P Q* ⟶ (∀ α *P'*. *bn* α ♯* *Q* ⟶ *P* → ⟨α,*P'*⟩ ⟶ (∃ *Q'*. *Q* → ⟨α,*Q'*⟩ ∧ *R P' Q'*))
    have *symp* (*p* · *R*) (**is** *?S*)
      using *1* **by** (*simp add*: *symp-eqvt*)
    moreover have ∀ *P Q*. (*p* · *R*) *P Q* ⟶ (∀ φ. *P* ⊢ φ ⟶ *Q* ⊢ φ) (**is** *?T*)
      proof (*clarify*)
        fix *P Q* φ
        assume ∗: (*p* · *R*) *P Q* **and** ∗∗: *P* ⊢ φ
        from ∗ **have** *R* (−*p* · *P*) (−*p* · *Q*)
          by (*simp add*: *eqvt-lambda permute-bool-def unpermute-def*)
        then show *Q* ⊢ φ
          using *2* ∗∗ **by** (*metis permute-minus-cancel*(*1*) *satisfies-eqvt*)
      qed

```
  moreover have ∀ P Q. (p · R) P Q ⟶ (∀ α P'. bn α ♯* Q ⟶ P → ⟨α,P'⟩
⟶ (∃ Q'. Q → ⟨α,Q'⟩ ∧ (p · R) P' Q')) (is ?U)
    proof (clarify)
      fix P Q α P'
      assume ∗: (p · R) P Q and ∗∗: bn α ♯* Q and ∗∗∗: P → ⟨α,P'⟩
      from ∗ have R (−p · P) (−p · Q)
        by (simp add: eqvt-lambda permute-bool-def unpermute-def)
      moreover have bn (−p · α) ♯* (−p · Q)
        using ∗∗ by (metis bn-eqvt fresh-star-permute-iff)
      moreover have −p · P → ⟨−p · α, −p · P'⟩
        using ∗∗∗ by (metis transition-eqvt')
      ultimately obtain Q' where T: −p · Q → ⟨−p · α,Q'⟩ and R: R (−p ·
P') Q'
        using 3 by metis
      from T have Q → ⟨α, p · Q'⟩
        by (metis permute-minus-cancel(1) transition-eqvt')
      moreover from R have (p · R) P' (p · Q')
        by (metis eqvt-apply eqvt-lambda permute-bool-def unpermute-def)
      ultimately show ∃ Q'. Q → ⟨α,Q'⟩ ∧ (p · R) P' Q'
        by metis
    qed
  ultimately show ?S ∧ ?T ∧ ?U by simp
 qed

 lemma bisimilar-eqvt :
  assumes P ∼· Q shows (p · P) ∼· (p · Q)
 proof −
  from assms obtain R where ∗: is-bisimulation R ∧ R P Q
    unfolding bisimilar-def ..
  then have is-bisimulation (p · R)
    by (simp add: is-bisimulation-eqvt)
  moreover from ∗ have (p · R) (p · P) (p · Q)
    by (metis eqvt-apply permute-boolI)
  ultimately show (p · P) ∼· (p · 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 ∼· x
    unfolding bisimilar-def by auto
 qed

 lemma bisimilar-symp: symp bisimilar
 proof (rule sympI)
  fix P Q
```

```
    assume P ∼· Q
    then obtain R where ∗: is-bisimulation R ∧ R P Q
      unfolding bisimilar-def ..
    then have R Q P
      unfolding is-bisimulation-def by (simp add: symp-def)
    with ∗ show Q ∼· P
      unfolding bisimilar-def by auto
  qed

  lemma bisimilar-is-bisimulation: is-bisimulation bisimilar
  unfolding is-bisimulation-def proof
    show symp (∼·)
      by (fact bisimilar-symp)
  next
    show (∀ P Q. P ∼· Q ⟶ (∀ φ. P ⊢ φ ⟶ Q ⊢ φ)) ∧
      (∀ P Q. P ∼· Q ⟶ (∀ α P'. bn α ♯* Q ⟶ P → ⟨α,P'⟩ ⟶ (∃ Q'. Q →
⟨α,Q'⟩ ∧ P' ∼· 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 ∼· Q and QR: Q ∼· R
    let ?bisim = bisimilar OO bisimilar
    have symp ?bisim
      proof (rule sympI)
        fix P R
        assume ?bisim P R
        then obtain Q where P ∼· Q and Q ∼· R
          by blast
        then have R ∼· Q and Q ∼· P
          by (metis bisimilar-symp sympE)+
        then show ?bisim R P
          by blast
      qed
    moreover have ∀ P Q. ?bisim P Q ⟶ (∀ φ. P ⊢ φ ⟶ Q ⊢ φ)
      using bisimilar-is-bisimulation is-bisimulation-def by auto
    moreover have ∀ P Q. ?bisim P Q ⟶
          (∀ α P'. bn α ♯* Q ⟶ P → ⟨α,P'⟩ ⟶ (∃ Q'. Q → ⟨α,Q'⟩ ∧ ?bisim P'
Q'))
      proof (clarify)
        fix P R Q α P'
        assume PR: P ∼· R and RQ: R ∼· Q and fresh: bn α ♯* Q and trans: P
→ ⟨α,P'⟩
        — rename ⟨α,P'⟩ to avoid R, without touching Q
        obtain p where 1: (p · bn α) ♯* R and 2: supp (⟨α,P'⟩, Q) ♯* p
          proof (rule at-set-avoiding2[of bn α R (⟨α,P'⟩, Q), THEN exE])
            show finite (bn α) by (fact bn-finite)
```

```
        next
          show finite (supp R) by (fact finite-supp)
        next
          show finite (supp (⟨α,P'⟩, Q)) by (simp add: finite-supp supp-Pair)
        next
          show bn α ♯* (⟨α,P'⟩, Q) by (simp add: fresh fresh-star-Pair)
        qed metis
      from 2 have 3: supp ⟨α,P'⟩ ♯* p and 4: supp Q ♯* p
        by (simp add: fresh-star-Un supp-Pair)+
      from 3 have ⟨p • α, p • P'⟩ = ⟨α,P'⟩
        using supp-perm-eq by fastforce
      then obtain pR' where 5: R → ⟨p • α, pR'⟩ and 6: (p • P') ∼· 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 • α) ♯* Q
        by (metis bn-eqvt fresh-star-permute-iff supp-perm-eq)
      then obtain pQ' where 7: Q → ⟨p • α, pQ'⟩ and 8: pR' ∼· pQ'
     using RQ 5 by (metis (full-types) bisimilar-is-bisimulation is-bisimulation-def)
      from 7 have Q → ⟨α, −p • pQ'⟩
       using 4 by (metis permute-minus-cancel(2) supp-perm-eq transition-eqvt')
      moreover from 6 and 8 have ?bisim P' (−p • pQ')
          by (metis (no-types, hide-lams) bisimilar-eqvt permute-minus-cancel(2)
relcompp.simps)
      ultimately show ∃ Q'. Q → ⟨α,Q'⟩ ∧ ?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 ∼· 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 ∼· Q and bn α ♯* Q and P → ⟨α,P'⟩
  obtains Q' where Q → ⟨α,Q'⟩ and P' ∼· 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**
  $t \in$ *set-bset tset* $\Longrightarrow$ (*t*, *tConj tset*) $\in$ *Tree-wf*
| (*t*, *tNot t*) $\in$ *Tree-wf*
| (*t*, *tAct* α *t*) $\in$ *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* $\Rightarrow$ *bool* **and** *t*
  assume $\forall x.\ (\forall y.\ (y, x) \in$ *Tree-wf* $\longrightarrow$ *P y*$) \longrightarrow$ *P x*
  then show *P t*
    proof (*induction t*)
      case *tConj* **then show** *?case*
        by (*metis Tree.distinct*(*2*) *Tree.distinct*(*5*) *Tree.inject*(*1*) *Tree-wf.cases*)
    next
      case *tNot* **then show** *?case*
        by (*metis Tree.distinct*(*1*) *Tree.distinct*(*9*) *Tree.inject*(*2*) *Tree-wf.cases*)
    next
      case *tPred* **then show** *?case*
       by (*metis Tree.distinct*(*11*) *Tree.distinct*(*3*) *Tree.distinct*(*7*) *Tree-wf.cases*)
    next
      case *tAct* **then show** *?case*
        by (*metis Tree.distinct*(*10*) *Tree.distinct*(*6*) *Tree.inject*(*4*) *Tree-wf.cases*)
    qed
qed

We define a permutation operation on the type of trees.

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

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

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

end

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

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

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

The relation *Tree-wf* is equivariant.

lemma *Tree-wf-eqvt-aux*:
  assumes (*t1*, *t2*) ∈ *Tree-wf* **shows** ($p \cdot t1$, $p \cdot t2$) ∈ *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** ($p \cdot t$, $p \cdot tConj\ tset$) ∈ *Tree-wf*
    by (*metis Tree-wf.intros*(*1*) *mem-permute-iff permute-Tree-tConj set-bset-eqvt*)
next
  fix *t* :: ('*a*,'*b*,'*c*) *Tree*
  show ($p \cdot t$, $p \cdot tNot\ t$) ∈ *Tree-wf*
    by (*metis Tree-wf.intros*(*2*) *permute-Tree.simps*(*2*))
next

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

lemma *Tree-wf-eqvt* [*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-eqvt-aux*)
next
show *Tree-wf* $\subseteq p \cdot$ *Tree-wf*
by (*auto simp add*: *permute-set-def*) (*metis Tree-wf-eqvt-aux permute-minus-cancel*(*1*))
qed

lemma *Tree-wf-eqvt'*: *eqvt Tree-wf*
by (*metis Tree-wf-eqvt 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* φ) = *supp* φ
unfolding *supp-def* **by** *simp*

lemma *supp-tAct* [*simp*]: *supp* (*tAct* α *t*) = *supp* $\alpha \cup$ *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*]:
$p \cdot$ *supp-rel R x* = *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* (*tConj tset*) = *supp-rel alpha-Tree* (*tConj tset*).
We then realized that it is not necessary to define the concept of "free variables" at all, but the definition of α-equivalence becomes much simpler (in particular, it is no longer mutually recursive) if we directly use the support modulo α-equivalence.

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

lemma *supp-rel-cong* [*fundef-cong*]:
⟦ $x{=}x'$; $\bigwedge a\ b.\ R\ ((a \rightleftharpoons b) \cdot x')\ x' \longleftrightarrow R'\ ((a \rightleftharpoons b) \cdot x')\ x'$ ⟧ ⟹ *supp-rel R x* = *supp-rel R′ x′*
by (*simp add*: *supp-rel-def*)

lemma *rel-bset-cong* [*fundef-cong*]:
⟦ $x{=}x'$; $y{=}y'$; $\bigwedge a\ b.\ a \in$ *set-bset* $x' \Longrightarrow b \in$ *set-bset* $y' \Longrightarrow R\ a\ b \longleftrightarrow R'\ a\ b$ ⟧
⟹ *rel-bset R x y* ⟷ *rel-bset R′ x′ y′*
by (*simp add*: *rel-bset-def rel-set-def*)

lemma *alpha-set-cong* [*fundef-cong*]:
⟦ $bs{=}bs'$; $x{=}x'$; $R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y'$; $f\,x' = f'\,x'$; $f\,y' = f'\,y'$; $p{=}p'$; $cs{=}cs'$; $y{=}y'$ ⟧ ⟹
alpha-set (*bs*, *x*) *R f p* (*cs*, *y*) ⟷ *alpha-set* (*bs′*, *x′*) *R′ f′ p′* (*cs′*, *y′*)
by (*simp add*: *alpha-set*)

quotient-type
(*'idx*,*'pred*,*'act*) $Tree_p$ = (*'idx*,*'pred*::*pt*,*'act*::*bn*) *Tree* / *hull-relp*
by (*fact hull-relp-equivp*)

lemma *abs-Tree*$_p$*-eq* [*simp*]: *abs-Tree*$_p$ $(p \cdot t)$ = *abs-Tree*$_p$ *t*
by (*metis hull-relp.simps Tree*$_p$*.abs-eq-iff*)

lemma *permute-rep-abs-Tree*$_p$:
obtains *p* **where** *rep-Tree*$_p$ (*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 *Tree-wf*$_p$ :: (*'idx*,*'pred*::*pt*,*'act*::*bn*) $Tree_p$ *rel* **is**
Tree-wf .

lemma *Tree-wf*$_p$*I* [*simp*]:
assumes (*a*, *b*) ∈ *Tree-wf*
shows (*abs-Tree*$_p$ $(p \cdot a)$, *abs-Tree*$_p$ *b*) ∈ *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*) ∈ *Tree-wf*
shows (*abs-Tree*$_p$ *a*, *abs-Tree*$_p$ *b*) ∈ *Tree-wf*$_p$

using *assms* **by** (*metis Tree-wf*$_p$*I permute-zero*)

lemma *Tree-wf*$_p$*E*:
assumes (a_p, b_p) ∈ *Tree-wf*$_p$
obtains *a b* **where** a_p = *abs-Tree*$_p$ *a* **and** b_p = *abs-Tree*$_p$ *b* **and** (*a*,*b*) ∈ *Tree-wf*
using *assms* **by** (*metis Pair-inject Tree-wf*$_p$*.abs-eq prod-fun-imageE*)

lemma *wf-Tree-wf*$_p$: *wf Tree-wf*$_p$
proof (*rule wf-subset*[*of inv-image* (*hull-rel O Tree-wf*) *rep-Tree*$_p$])
show *wf* (*inv-image* (*hull-rel O Tree-wf*) *rep-Tree*$_p$)
by (*metis Tree-wf-eqvt*′ *wf-Tree-wf wf-hull-rel-relcomp wf-inv-image*)
next
show *Tree-wf*$_p$ ⊆ *inv-image* (*hull-rel O Tree-wf*) *rep-Tree*$_p$
proof (*standard*, *case-tac x*, *clarify*)
fix a_p b_p :: (′*d*, ′*e*, ′*f*) *Tree*$_p$
assume (a_p, b_p) ∈ *Tree-wf*$_p$
then obtain *a b* **where** *1*: a_p = *abs-Tree*$_p$ *a* **and** *2*: b_p = *abs-Tree*$_p$ *b* **and** *3*: (*a*,*b*) ∈ *Tree-wf*
by (*rule Tree-wf*$_p$*E*)
from *1* **obtain** *p* **where** *4*: *rep-Tree*$_p$ a_p = *p* ∙ *a*
by (*metis permute-rep-abs-Tree*$_p$)
from *2* **obtain** *q* **where** *5*: *rep-Tree*$_p$ b_p = *q* ∙ *b*
by (*metis permute-rep-abs-Tree*$_p$)
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 (a_p, b_p) ∈ *inv-image* (*hull-rel O Tree-wf*) *rep-Tree*$_p$
using *4 5* **by** *auto*
qed
qed

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

Here it comes ...

function (*sequential*)
alpha-Tree :: (′*idx*,′*pred*::*pt*,′*act*::*bn*) *Tree* ⇒ (′*idx*,′*pred*,′*act*) *Tree* ⇒ *bool* (**infix** =$_\alpha$ *50*) **where**
— (=$_\alpha$)
alpha-tConj: *tConj tset1* =$_\alpha$ *tConj tset2* ⟷ *rel-bset alpha-Tree tset1 tset2*
| *alpha-tNot*: *tNot t1* =$_\alpha$ *tNot t2* ⟷ *t1* =$_\alpha$ *t2*
| *alpha-tPred*: *tPred φ1* =$_\alpha$ *tPred φ2* ⟷ *φ1* = *φ2*
— the action may have binding names
| *alpha-tAct*: *tAct α1 t1* =$_\alpha$ *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*: - =$_\alpha$ - ⟷ *False*

— 254 subgoals (!)
by *pat-completeness auto*
termination
proof
let *?R* = *inv-image* (*max-ext Tree-wf*$_p$) *alpha-Tree-termination*
show *wf ?R*
by (*metis max-ext-wf wf-Tree-wf*$_p$ *wf-inv-image*)
qed (*auto simp add*: *max-ext.simps Tree-wf.simps simp del*: *permute-Tree-tConj*)

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

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

lemma *alpha-Tree-induct*[*case-names tConj tNot tPred tAct*, *consumes 1*]:
assumes *t1* $=_\alpha$ *t2*
and $\bigwedge$*tset1 tset2*. ($\bigwedge$*a b*. *a* $\in$ *set-bset tset1* $\Longrightarrow$ *b* $\in$ *set-bset tset2* $\Longrightarrow$ *a* $=_\alpha$ *b* $\Longrightarrow$ *P a b*) $\Longrightarrow$
rel-bset ($=_\alpha$) *tset1 tset2* $\Longrightarrow$ *P* (*tConj tset1*) (*tConj tset2*)
and $\bigwedge$*t1 t2*. *t1* $=_\alpha$ *t2* $\Longrightarrow$ *P t1 t2* $\Longrightarrow$ *P* (*tNot t1*) (*tNot t2*)
and $\bigwedge\varphi$. *P* (*tPred* φ) (*tPred* φ)
and $\bigwedge\alpha 1\ t1\ \alpha 2\ t2$. ($\bigwedge$*p*. *p* $\cdot$ *t1* $=_\alpha$ *t2* $\Longrightarrow$ *P* (*p* $\cdot$ *t1*) *t2*) $\Longrightarrow$
($\bigwedge$*a b*. ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t1*) $=_\alpha$ *t1* $\Longrightarrow$ *P* ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t1*) *t1*) $\Longrightarrow$ ($\bigwedge$*a b*. ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t2*) $=_\alpha$ *t2* $\Longrightarrow$ *P* ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t2*) *t2*) $\Longrightarrow$
($\exists$*p*. (*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* $\alpha 2$, $\alpha 2$)) $\Longrightarrow$
P (*tAct* $\alpha 1$ *t1*) (*tAct* $\alpha 2$ *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-rel* ($=_\alpha$) *t* = *supp-rel* ($=_\alpha$) (*p* $\cdot$ *t*)
proof −
{
fix *a*
let *?B* = {*b*. ¬ ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t*) $=_\alpha$ *t*}
let *?pB* = {*b*. ¬ ((*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 ⊆ ?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 ⟷ 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 =α t2 ⟷ p • t1 =α p • 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-eqvt)
   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-eqvt)
    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-eqvt)
    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 ∃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 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 α1 t1 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 α1 t1* $=_\alpha$ *tAct α2 t2*
then obtain *q* **where** *1*: (*bn α1*, *t1*) $\approx$*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) *q* (*bn α2*, *t2*) **and** *2*: (*bn α1*, *α1*) $\approx$*set* (=) *supp q* (*bn α2*, *α2*)
by *auto*
from *1* **and** *t1* **and** *t2* **have** *supp-rel* ($=_\alpha$) (*p* $\cdot$ *t1*) − *bn* (*p* $\cdot$ *α1*) = *supp-rel* ($=_\alpha$) (*p* $\cdot$ *t2*) − *bn* (*p* $\cdot$ *α2*)
by (*metis Diff-eqvt alpha-set bn-eqvt*)
moreover from *1* **and** *t1* **have** (*supp-rel* ($=_\alpha$) (*p* $\cdot$ *t1*) − *bn* (*p* $\cdot$ *α1*)) $\sharp*$ (*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* $\cdot$ *α1*) = *bn* (*p* $\cdot$ *α2*)
by (*simp add*: *alpha-set bn-eqvt*)
ultimately have (*bn* (*p* $\cdot$ *α1*), *p* $\cdot$ *t1*) $\approx$*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) (*p* + *q* − *p*) (*bn* (*p* $\cdot$ *α2*), *p* $\cdot$ *t2*)
by (*simp add*: *alpha-set*)
moreover from *2* **have** (*bn* (*p* $\cdot$ *α1*), *p* $\cdot$ *α1*) $\approx$*set* (=) *supp* (*p* + *q* − *p*) (*bn* (*p* $\cdot$ *α2*), *p* $\cdot$ *α2*)
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 α1 t1* $=_\alpha$ *p* $\cdot$ *tAct α2 t2*
by *auto*
next
assume *p* $\cdot$ *tAct α1 t1* $=_\alpha$ *p* $\cdot$ *tAct α2 t2*
then obtain *q* **where** *1*: (*bn* (*p* $\cdot$ *α1*), *p* $\cdot$ *t1*) $\approx$*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) *q* (*bn* (*p* $\cdot$ *α2*), *p* $\cdot$ *t2*) **and** *2*: (*bn* (*p* $\cdot$ *α1*), *p* $\cdot$ *α1*) $\approx$*set* (=) *supp q* (*bn* (*p* $\cdot$ *α2*), *p* $\cdot$ *α2*)
by *auto*
{
from *1* **and** *t1* **and** *t2* **have** *supp-rel* ($=_\alpha$) *t1* − *bn α1* = *supp-rel* ($=_\alpha$) *t2* − *bn α2*
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 α1*) $\sharp*$ (− *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 α1* = *bn α2*
by (*metis alpha-set bn-eqvt permute-minus-cancel*(*2*))
ultimately have (*bn α1*, *t1*) $\approx$*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) (−*p* + *q* + *p*) (*bn α2*, *t2*)
by (*simp add*: *alpha-set*)
}
moreover
{
from *2* **have** *supp α1* − *bn α1* = *supp α2* − *bn α2*

by (*metis* (*no-types*, *lifting*) *Diff-eqvt alpha-set bn-eqvt permute-eq-iff supp-eqvt*)
moreover with *2* **have** (*supp* $\alpha 1 - bn\ \alpha 1$) $\sharp*$ $(-p + q + p)$
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*]: $t1 =_\alpha t2 \Longrightarrow p \cdot t1 =_\alpha p \cdot t2$
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*) $\approx set\ (=_\alpha)\ (supp\text{-}rel\ (=_\alpha))\ p$ (*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*) $\approx set\ (=_\alpha)\ (supp\text{-}rel\ (=_\alpha))\ (-p)$ (*bn* $\alpha 1$, *t1*) $\wedge$ (*bn* $\alpha 2$,

```
α2) ≈set (=) supp (−p) (bn α1, α1)
      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) 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 ∀x'. x' ∈ set-bset tset-x ⟶ (∃z'. z' ∈ set-bset tset-z ∧ x' =α 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.prems z)
              from 1 2 3 5 have x' =α z'
                by (rule tConj.IH)
              with 4 show ∃z'. z' ∈ set-bset tset-z ∧ x' =α z'
                by auto
            qed
          next
            show ∀z'. z' ∈ set-bset tset-z ⟶ (∃x'. x' ∈ set-bset tset-x ∧ x' =α 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.prems z)
              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 ∃x'. x' ∈ set-bset tset-x ∧ x' =α z'
                by auto
            qed
          qed
        with z show tConj tset-x =α z
          by simp
```

```
    qed (insert tConj.prems, auto)
  next
    case tNot then show ?case
      by (cases z) simp-all
  next
    case tPred then show ?case
      by simp
  next
    case (tAct α1 t1 α2 t2) show ?case
    proof (cases z)
      fix α t
      assume z: z = tAct α t
      obtain p where 1: (bn α1, t1) ≈set (=α) (supp-rel (=α)) p (bn α2, t2) ∧
(bn α1, α1) ≈set (=) supp p (bn α2, α2)
        using tAct.hyps by auto
      obtain q where 2: (bn α2, t2) ≈set (=α) (supp-rel (=α)) q (bn α, t) ∧ (bn
α2, α2) ≈set (=) supp q (bn α, α)
        using tAct.prems z by auto
      have (bn α1, t1) ≈set (=α) (supp-rel (=α)) (q + p) (bn α, t)
        proof −
          have supp-rel (=α) t1 − bn α1 = supp-rel (=α) t − bn α
            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.prems, 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*: $\bigwedge x\ y.$ *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* $(=_\alpha)$ $((a \rightleftharpoons b) \cdot$ *tset1*$)$ $((a \rightleftharpoons b) \cdot$ *tset2*$)$
by (*metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj*)
have *rel-bset* $(=_\alpha)$ $((a \rightleftharpoons b) \cdot$ *tset1*$)$ *tset1* $\longleftrightarrow$ *rel-bset* $(=_\alpha)$ $((a \rightleftharpoons b) \cdot$ *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* $=_\alpha$ *tAct α2 t2*
using *tAct.hyps* **by** *simp*
then have $(a \rightleftharpoons b) \cdot$ *tAct α1 t1* $=_\alpha$ *tAct α1 t1* $\longleftrightarrow$ $(a \rightleftharpoons b) \cdot$ *tAct α2 t2* $=_\alpha$ *tAct α2 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 α t1* $=_\alpha$ *tAct α t2*
proof −
have (*bn α*, *t1*) $\approx$*set* $(=_\alpha)$ (*supp-rel* $(=_\alpha)$) *0* (*bn α*, *t2*)
using *assms* **by** (*simp add*: *alpha-Tree-supp-rel alpha-set fresh-star-zero*)
moreover have (*bn α*, *α*) $\approx$*set* $(=)$ *supp 0* (*bn α*, *α*)
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 φ*) = *supp φ*

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

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

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

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

```
      moreover
      {
        fix b
        assume b ∈ {b. ¬ (x ⇌ b) • t =α t} − supp-rel (=α) t
        then have b1: ¬ (x ⇌ b) • t =α t and b2: b ∉ supp-rel (=α) t − bn α
          by simp+
        from b1 have (x ⇌ b) • t ≠ t
          by (metis alpha-Tree-refl p reflpE)
        then have sort-of x = sort-of b
          using swap-different-sorts by fastforce
        then have (x ⇌ b) • (supp-rel (=α) t − bn α) ≠ supp-rel (=α) t − bn α
          using b2 x1 x2 by (simp add: swap-set-in)
       then have supp-rel (=α) ((x ⇌ b) • t) − bn ((x ⇌ b) • α) ≠ supp-rel (=α)
t − bn α
          by (simp add: Diff-eqvt bn-eqvt)
        then have b ∈ {b. ¬ (x ⇌ b) • tAct α t =α tAct α t}
          by (simp add: alpha-set)
      }
      ultimately have infinite {b. ¬ (x ⇌ b) • tAct α t =α tAct α t}
        by (rule infinite-mono)
      then have x ∈ supp-rel (=α) (tAct α t)
        unfolding supp-rel-def ..
    }
    ultimately show x ∈ supp-rel (=α) (tAct α t)
      by auto
  qed
next
  show supp-rel (=α) (tAct α t) ⊆ supp α ∪ supp-rel (=α) t − bn α
  proof
    fix x
    assume x ∈ supp-rel (=α) (tAct α t)
    then have ∗: infinite {b. ¬ (x ⇌ b) • tAct α t =α tAct α t}
      unfolding supp-rel-def ..
    moreover
    {
      fix b
      assume ¬ (x ⇌ b) • tAct α t =α tAct α t
      then have (x ⇌ b) • α ≠ α ∨ ¬ (x ⇌ b) • t =α t
        using alpha-Tree-tAct by force
    }
    ultimately have infinite {b. (x ⇌ b) • α ≠ α ∨ ¬ (x ⇌ b) • t =α t}
      by (metis (mono-tags, lifting) infinite-mono mem-Collect-eq)
    then have infinite {b. (x ⇌ b) • α ≠ α} ∨ infinite {b. ¬ (x ⇌ b) • t =α t}
      by (metis (mono-tags) finite-Collect-disjI)
    then have x ∈ supp α ∪ supp-rel (=α) t
      by (simp add: supp-def supp-rel-def)
    moreover
    {
      assume ∗∗: x ∈ bn α
```

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

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

quotient-type
$('idx,'pred,'act)\ Tree_\alpha = ('idx,'pred::pt,'act::bn)\ Tree\ /\ alpha\text{-}Tree$
by (*fact alpha-Tree-equivp*)

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

lemma $Tree_\alpha\text{-}rep\text{-}abs$ [*simp*]: $rep\text{-}Tree_\alpha\ (abs\text{-}Tree_\alpha\ t) =_\alpha t$
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 $permute\text{-}Tree_\alpha :: perm \Rightarrow ('a,'b,'c)\ Tree_\alpha \Rightarrow ('a,'b,'c)\ Tree_\alpha$

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

instance
proof
fix t_α :: (-,-,-) $Tree_\alpha$
show $0 \cdot t_\alpha = t_\alpha$
by *transfer* (*metis alpha-Tree-equivp equivp-reflp permute-zero*)
next
fix p q :: *perm* **and** t_α :: (-,-,-) $Tree_\alpha$
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 $permute\text{-}Tree_\alpha.abs\text{-}eq$ [*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* $Tree_\alpha.abs\text{-}eq\text{-}iff$ $Tree_\alpha\text{-}abs\text{-}rep$ $permute\text{-}Tree_\alpha.abs\text{-}eq$)

5.3 Constructors for trees modulo α-equivalence

The constructors are lifted from trees.

lift-definition $Conj_\alpha$ :: ('*idx*,'*pred*,'*act*) $Tree_\alpha$ *set*['*idx*] ⇒ ('*idx*,'*pred*::*pt*,'*act*::*bn*) $Tree_\alpha$ **is**
tConj
by *simp*

lemma $map\text{-}bset\text{-}abs\text{-}rep\text{-}Tree_\alpha$: *map-bset* $abs\text{-}Tree_\alpha$ (*map-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-bset* $rep\text{-}Tree_\alpha$ $tset_\alpha$))
by (*metis* $Conj_\alpha.abs\text{-}eq$ $map\text{-}bset\text{-}abs\text{-}rep\text{-}Tree_\alpha$)

lift-definition Not_α :: ('*idx*,'*pred*,'*act*) $Tree_\alpha$ ⇒ ('*idx*,'*pred*::*pt*,'*act*::*bn*) $Tree_\alpha$ **is**
tNot
by *simp*

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

lift-definition Act_α :: '*act* ⇒ ('*idx*,'*pred*,'*act*) $Tree_\alpha$ ⇒ ('*idx*,'*pred*::*pt*,'*act*::*bn*) $Tree_\alpha$ **is**

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

The lifted constructors are equivariant.

lemma $Conj_\alpha$*-eqvt* [*eqvt*, *simp*]: $p \cdot Conj_\alpha\ tset_\alpha = Conj_\alpha\ (p \cdot tset_\alpha)$
proof −
{
fix x
assume $x \in$ *set-bset* $(p \cdot$ *map-bset rep-Tree*$_\alpha\ tset_\alpha)$
then obtain y **where** $y \in$ *set-bset* $($*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*: $t_\alpha \in$ *set-bset* $tset_\alpha$ **and** *2*: $x = p \cdot$ *rep-Tree*$_\alpha\ t_\alpha$
by (*metis imageE map-bset.rep-eq*)
let *?x′* = *rep-Tree*$_\alpha\ (p \cdot t_\alpha)$
from *1* **have** $p \cdot t_\alpha \in$ *set-bset* $(p \cdot tset_\alpha)$
by (*metis mem-permute-iff permute-bset.rep-eq*)
then have *?x′* ∈ *set-bset* $($*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 $\exists\, x' \in$*set-bset* $(map\text{-}bset\ rep\text{-}Tree_\alpha\ (p \cdot tset_\alpha)).\ x =_\alpha x'$
..
}
moreover
{
fix y
assume $y \in$ *set-bset* $(map\text{-}bset\ rep\text{-}Tree_\alpha\ (p \cdot tset_\alpha))$
then obtain x **where** $x \in$ *set-bset* $(p \cdot tset_\alpha)$ **and** *rep-Tree*$_\alpha\ x = y$
by (*metis imageE map-bset.rep-eq*)
then obtain t_α **where** *1*: $t_\alpha \in$ *set-bset* $tset_\alpha$ **and** *2*: *rep-Tree*$_\alpha\ (p \cdot t_\alpha) = y$
by (*metis imageE permute-bset.rep-eq permute-set-eq-image*)
let *?y′* = $p \cdot$ *rep-Tree*$_\alpha\ t_\alpha$
from *1* **have** *rep-Tree*$_\alpha\ t_\alpha \in$ *set-bset* $(map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha)$
by (*simp add*: *bset.set-map*)
then have *?y′* ∈ *set-bset* $(p \cdot map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha)$
by (*metis mem-permute-iff permute-bset.rep-eq*)
moreover from *2* **have** *?y′* $=_\alpha y$
by (*metis alpha-Tree-permute-rep-commute*)
ultimately have $\exists\, y' \in$*set-bset* $(p \cdot map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha).\ y' =_\alpha y$
..
}
ultimately show *?thesis*
by (*simp add*: $Conj_\alpha$*-def′ map-bset-eqvt rel-bset-def rel-set-def Tree*$_\alpha$*.abs-eq-iff*)
qed

lemma Not_α*-eqvt* [*eqvt*, *simp*]: $p \cdot Not_\alpha\ t_\alpha = Not_\alpha\ (p \cdot t_\alpha)$
by (*induct* t_α) (*simp add*: Not_α*.abs-eq*)

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

by (*simp add*: $Pred_\alpha$.*abs-eq*)

lemma Act_α-*eqvt* [*eqvt*, *simp*]: $p \cdot Act_\alpha\ \alpha\ t_\alpha = Act_\alpha\ (p \cdot \alpha)\ (p \cdot t_\alpha)$
by (*induct* t_α) (*simp add*: Act_α.*abs-eq*)

The lifted constructors are injective (except for Act_α).

lemma $Conj_\alpha$-*eq-iff* [*simp*]: $Conj_\alpha\ tset1_\alpha = Conj_\alpha\ tset2_\alpha \longleftrightarrow tset1_\alpha = tset2_\alpha$
proof
assume $Conj_\alpha\ tset1_\alpha = Conj_\alpha\ tset2_\alpha$
then have *tConj* (*map-bset rep-Tree*$_\alpha$ $tset1_\alpha$) $=_\alpha$ *tConj* (*map-bset rep-Tree*$_\alpha$ $tset2_\alpha$)
by (*metis* $Conj_\alpha$-*def'* $Tree_\alpha$.*abs-eq-iff*)
then have *rel-bset* $(=_\alpha)$ (*map-bset rep-Tree*$_\alpha$ $tset1_\alpha$) (*map-bset rep-Tree*$_\alpha$ $tset2_\alpha$)
by (*auto elim*: *alpha-Tree.cases*)
then show $tset1_\alpha = tset2_\alpha$
using *Quotient-Tree*$_\alpha$ *Quotient-rel-abs2 bset-lifting.bset-quot-map map-bset-abs-rep-Tree*$_\alpha$
by *fastforce*
qed (*fact arg-cong*)

lemma Not_α-*eq-iff* [*simp*]: $Not_\alpha\ t1_\alpha = Not_\alpha\ t2_\alpha \longleftrightarrow t1_\alpha = t2_\alpha$
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_α.*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*]: $Pred_\alpha\ \varphi 1 = Pred_\alpha\ \varphi 2 \longleftrightarrow \varphi 1 = \varphi 2$
proof
assume $Pred_\alpha\ \varphi 1 = Pred_\alpha\ \varphi 2$
then have (*tPred* $\varphi 1$ :: $('d, 'b, 'e)$ *Tree*) $=_\alpha$ *tPred* $\varphi 2$ — note the unrelated type
by (*metis* $Pred_\alpha$.*abs-eq* $Tree_\alpha$.*abs-eq-iff*)
then show $\varphi 1 = \varphi 2$
using *alpha-Tree.cases* **by** *auto*
next
assume $\varphi 1 = \varphi 2$ **then show** $Pred_\alpha\ \varphi 1 = Pred_\alpha\ \varphi 2$
by *simp*
qed

lemma Act_α-*eq-iff*: $Act_\alpha\ \alpha 1\ t1 = Act_\alpha\ \alpha 2\ t2 \longleftrightarrow$ *tAct* $\alpha 1$ (*rep-Tree*$_\alpha$ $t1$) $=_\alpha$ *tAct* $\alpha 2$ (*rep-Tree*$_\alpha$ $t2$)
by (*metis* Act_α.*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*: $supp\ t_\alpha = supp\text{-}rel\ (=_\alpha)\ (rep\text{-}Tree_\alpha\ t_\alpha)$
unfolding *supp-def supp-rel-def* **by** (*metis* (*mono-tags*, *lifting*) *Collect-cong Tree$_\alpha$.abs-eq-iff Tree$_\alpha$-abs-rep alpha-Tree-permute-rep-commute*)

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

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

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

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

5.4 Induction over trees modulo α-equivalence

lemma *Tree$_\alpha$-induct* [*case-names Conj$_\alpha$ Not$_\alpha$ Pred$_\alpha$ Act$_\alpha$ Env$_\alpha$*, *induct type*: *Tree$_\alpha$*]:
  fixes t_α
  assumes $\bigwedge tset_\alpha.\ (\bigwedge x.\ x \in set\text{-}bset\ tset_\alpha \Longrightarrow P\ x) \Longrightarrow P\ (Conj_\alpha\ tset_\alpha)$
    and $\bigwedge t_\alpha.\ P\ t_\alpha \Longrightarrow P\ (Not_\alpha\ t_\alpha)$
    and $\bigwedge pred.\ P\ (Pred_\alpha\ pred)$
    and $\bigwedge act\ t_\alpha.\ P\ t_\alpha \Longrightarrow P\ (Act_\alpha\ act\ t_\alpha)$
  shows $P\ t_\alpha$
proof (*rule Tree$_\alpha$.abs-induct*)
  fix *t* **show** $P\ (abs\text{-}Tree_\alpha\ t)$
    proof (*induction t*)
      case (*tConj tset*)
        let $?tset_\alpha = map\text{-}bset\ abs\text{-}Tree_\alpha\ tset$
        have $abs\text{-}Tree_\alpha\ (tConj\ tset) = Conj_\alpha\ ?tset_\alpha$
          by (*simp add*: *Conj$_\alpha$.abs-eq*)
        then show *?case*

```
      using assms(1) tConj.IH by (metis imageE map-bset.rep-eq)
  next
    case tNot then show ?case
      using assms(2) by (metis 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.

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_α ⇒ bool where
  Conj_α: finite (supp tset_α) ⟹ (⋀t_α. t_α ∈ set-bset tset_α ⟹ hereditarily-fs t_α)
⟹ hereditarily-fs (Conj_α tset_α)
| Not_α: hereditarily-fs t_α ⟹ hereditarily-fs (Not_α t_α)
| Pred_α: hereditarily-fs (Pred_α φ)
| Act_α: hereditarily-fs t_α ⟹ hereditarily-fs (Act_α α t_α)
```

hereditarily-fs is equivariant.

```
lemma hereditarily-fs-eqvt [eqvt]:
  assumes hereditarily-fs t_α
  shows hereditarily-fs (p · t_α)
using assms proof (induction rule: hereditarily-fs.induct)
  case Conj_α then show ?case
   by (metis (erased, hide-lams) Conj_α-eqvt hereditarily-fs.Conj_α mem-permute-iff
permute-finite permute-minus-cancel(1) set-bset-eqvt supp-eqvt)
next
  case Not_α then show ?case
    by (metis Not_α-eqvt hereditarily-fs.Not_α)
next
  case Pred_α then show ?case
    by (metis Pred_α-eqvt hereditarily-fs.Pred_α)
```

next
case Act_α **then show** *?case*
by (*metis* Act_α*-eqvt hereditarily-fs.*Act_α)
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* $t_\alpha{}'$
using *assms* **by** (*auto simp add*: Act_α*-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_α
using *assms* **by** (*auto simp add*: Act_α*-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_α
shows *finite* (*supp* t_α)
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_α::('*idx*,'*pred*,'*act*) $Tree_\alpha$. *hereditarily-fs* t_α}
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_α
shows *Rep-formula* (*Abs-formula* t_α) = t_α
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* ⇒ ('*a*,'*b*,'*c*) *formula* ⇒ ('*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_α
shows $p \cdot$ *Abs-formula* t_α = *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 *supp* (*Abs-formula* t_α) = *supp* t_α
proof −
{
fix p :: *perm*
have $p \cdot$ *Abs-formula* t_α = *Abs-formula* $(p \cdot t_\alpha)$
using *assms* **by** (*metis Abs-formula-eqvt*)
moreover have *hereditarily-fs* $(p \cdot t_\alpha)$
using *assms* **by** (*metis hereditarily-fs-eqvt*)
ultimately have $p \cdot$ *Abs-formula* t_α = *Abs-formula* $t_\alpha \longleftrightarrow p \cdot t_\alpha = t_\alpha$
using *assms* **by** (*metis Abs-formula-inverse*)
}
then show *?thesis* **unfolding** *supp-def* **by** *simp*
qed

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

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

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

Formulas are in fact finitely supported.

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

5.7 Constructors for infinitary formulas

We lift the constructors for trees (modulo α-equivalence) to infinitary formulas. Since $Conj_\alpha$ 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* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula*
is
  Not_α
by (*fact hereditarily-fs*.Not_α)

lift-definition *Pred* :: '*pred* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* **is**
  $Pred_\alpha$
by (*fact hereditarily-fs*.$Pred_\alpha$)

lift-definition *Act* :: '*act* ⇒ ('*idx*,'*pred*,'*act*) *formula* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*)
formula **is**
  Act_α
by (*fact hereditarily-fs*.Act_α)

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

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

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

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

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

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
  assumes *finite* (*supp xset*)
  shows *supp* (*Conj xset*) = *supp xset*
using *assms* **unfolding** *Conj-def* **by** *simp*

lemma *supp-Not* [*simp*]: *supp* (*Not x*) = *supp x*
by (*metis Not.rep-eq supp-*Not_α *supp-Rep-formula*)

lemma *supp-Pred* [*simp*]: *supp* (*Pred* φ) = *supp* φ
by (*metis Pred.rep-eq supp-*$Pred_\alpha$ *supp-Rep-formula*)

lemma *supp-Act* [*simp*]: *supp* (*Act* α *x*) = *supp* α ∪ *supp x* − *bn* α
by (*metis Act.rep-eq finite-supp supp-*Act_α *supp-Rep-formula*)

lemma *bn-fresh-Act* [*simp*]: *bn* α ♯∗ *Act* α *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 *Conj xset1 = Conj xset2* $\longleftrightarrow$ *xset1 = xset2*
using *assms*
by (*metis* (*erased*, *hide-lams*) *Conj*$_\alpha$*-eq-iff Conj-rep-eq Rep-formula-inverse injI inj-eq bset.inj-map*)

lemma *Not-eq-iff* [*simp*]: *Not x1 = Not x2* $\longleftrightarrow$ *x1 = x2*
by (*metis Not.rep-eq Not*$_\alpha$*-eq-iff Rep-formula-inverse*)

lemma *Pred-eq-iff* [*simp*]: *Pred φ1 = Pred φ2* $\longleftrightarrow$ *φ1 = φ2*
by (*metis Pred.rep-eq Pred*$_\alpha$*-eq-iff*)

lemma *Act-eq-iff*: *Act α1 x1 = Act α2 x2* $\longleftrightarrow$ *Act*$_\alpha$ *α1* (*Rep-formula x1*) = *Act*$_\alpha$ *α2* (*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 α1 x1 = Act α2 x2* $\longleftrightarrow$
  ($\exists$ *p*. *supp x1 − bn α1 = supp x2 − bn α2* $\wedge$ (*supp x1 − bn α1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ *x1 = x2* $\wedge$ *supp α1 − bn α1 = supp α2 − bn α2* $\wedge$ (*supp α1 − bn α1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ *α1 = α2*)
  (**is** *?l* $\longleftrightarrow$ *?r*)
proof
  assume *?l*
  then obtain *p* **where** *alpha*: (*bn α1*, *rep-Tree*$_\alpha$ (*Rep-formula x1*)) $\approx$*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) *p* (*bn α2*, *rep-Tree*$_\alpha$ (*Rep-formula x2*)) **and** *eq*: (*bn α1*, *α1*) $\approx$*set* (=) *supp p* (*bn α2*, *α2*)
    by (*metis Act-eq-iff Act*$_\alpha$*-eq-iff alpha-tAct*)
  from *alpha* **have** *supp x1 − bn α1 = supp x2 − bn α2*
    by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)
  moreover from *alpha* **have** (*supp x1 − bn α1*) $\sharp*$ *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 α1 − bn α1 = supp α2 − bn α2*
    by (*metis alpha-set.simps*)
  moreover from *eq* **have** (*supp α1 − bn α1*) $\sharp*$ *p*
    by (*metis alpha-set.simps*)
  moreover from *eq* **have** *p* $\cdot$ *α1 = α2*
    by (*simp add*: *alpha-set.simps*)
  ultimately show *?r*
    by *metis*
next
  assume *?r*
  then obtain *p* **where** *1*: *supp x1 − bn α1 = supp x2 − bn α2* **and** *2*: (*supp x1 − bn α1*) $\sharp*$ *p* **and** *3*: *p* $\cdot$ *x1 = x2*

and *4*: $supp\ \alpha 1 - bn\ \alpha 1 = supp\ \alpha 2 - bn\ \alpha 2$ **and** *5*: $(supp\ \alpha 1 - bn\ \alpha 1) \sharp* p$ **and** *6*: $p \cdot \alpha 1 = \alpha 2$
by *metis*
from *1 2 3 6* **have** $(bn\ \alpha 1, rep\text{-}Tree_\alpha\ (Rep\text{-}formula\ x1)) \approx set\ (=_\alpha)\ (supp\text{-}rel\ (=_\alpha))\ p\ (bn\ \alpha 2, rep\text{-}Tree_\alpha\ (Rep\text{-}formula\ x2))$
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** $(bn\ \alpha 1, \alpha 1) \approx set\ (=)\ supp\ p\ (bn\ \alpha 2, \alpha 2)$
by (*simp add*: *alpha-set.simps bn-eqvt*)
ultimately show $Act\ \alpha 1\ x1 = Act\ \alpha 2\ x2$
by (*metis Act-eq-iff* Act_α*-eq-iff alpha-tAct*)
qed

lemma *Act-eq-iff-perm-renaming*: $Act\ \alpha 1\ x1 = Act\ \alpha 2\ x2 \longleftrightarrow$
$(\exists p.\ supp\ x1 - bn\ \alpha 1 = supp\ x2 - bn\ \alpha 2 \wedge (supp\ x1 - bn\ \alpha 1) \sharp* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha 1 - bn\ \alpha 1 = supp\ \alpha 2 - bn\ \alpha 2 \wedge (supp\ \alpha 1 - bn\ \alpha 1) \sharp* p \wedge p \cdot \alpha 1 = \alpha 2 \wedge supp\ p \subseteq bn\ \alpha 1 \cup p \cdot bn\ \alpha 1)$
(**is** $?l \longleftrightarrow ?r$)
proof
assume $?l$
then obtain p **where** p: $supp\ x1 - bn\ \alpha 1 = supp\ x2 - bn\ \alpha 2 \wedge (supp\ x1 - bn\ \alpha 1) \sharp* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha 1 - bn\ \alpha 1 = supp\ \alpha 2 - bn\ \alpha 2 \wedge (supp\ \alpha 1 - bn\ \alpha 1) \sharp* p \wedge p \cdot \alpha 1 = \alpha 2$
by (*metis Act-eq-iff-perm*)
moreover obtain q **where** $q\text{-}p$: $\forall b \in bn\ \alpha 1.\ q \cdot b = p \cdot b$ **and** $supp\text{-}q$: $supp\ q \subseteq bn\ \alpha 1 \cup p \cdot bn\ \alpha 1$
by (*metis set-renaming-perm2*)
have $supp\ q \subseteq supp\ p$
proof
fix a **assume** $*$: $a \in supp\ q$ **then show** $a \in supp\ p$
proof (*cases* $a \in bn\ \alpha 1$)
case *True* **then show** *?thesis*
using $*$ $q\text{-}p$ **by** (*metis mem-Collect-eq supp-perm*)
next
case *False* **then have** $a \in p \cdot bn\ \alpha 1$
using $*$ $supp\text{-}q$ **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** $(supp\ x1 - bn\ \alpha 1) \sharp* q$ **and** $(supp\ \alpha 1 - bn\ \alpha 1) \sharp* q$
by (*meson fresh-def fresh-star-def subset-iff*)+
moreover with p **and** $q\text{-}p$ **have** $\bigwedge a.\ a \in supp\ \alpha 1 \Longrightarrow q \cdot a = p \cdot a$ **and** $\bigwedge a.\ a \in supp\ x1 \Longrightarrow q \cdot a = p \cdot a$
by (*metis Diff-iff fresh-perm fresh-star-def*)+
then have $q \cdot \alpha 1 = p \cdot \alpha 1$ **and** $q \cdot x1 = p \cdot x1$
by (*metis supp-perm-perm-eq*)+

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

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

lemma *Tree-free* [*simp*]:
shows *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Not x*
and *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Pred* φ
and *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Act* α *x*
and *Not x* $\neq$ *Pred* φ
and *Not x1* $\neq$ *Act* α *x2*
and *Pred* φ $\neq$ *Act* α *x*
proof −
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Not x*
by (*metis Conj-rep-eq Not.rep-eq Tree*$_\alpha$*-free*(*1*))
next
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Pred* φ
by (*metis Conj-rep-eq Pred.rep-eq Tree*$_\alpha$*-free*(*2*))
next
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Act* α *x*
by (*metis Conj-rep-eq Act.rep-eq Tree*$_\alpha$*-free*(*3*))
next
show *Not x* $\neq$ *Pred* φ
by (*metis Not.rep-eq Pred.rep-eq Tree*$_\alpha$*-free*(*4*))
next
show *Not x1* $\neq$ *Act* α *x2*
by (*metis Not.rep-eq Act.rep-eq Tree*$_\alpha$*-free*(*5*))
next
show *Pred* φ $\neq$ *Act* α *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*) $\Longrightarrow$ ($\bigwedge$*x*. *x* $\in$ *set-bset xset* $\Longrightarrow$ *P x*) $\Longrightarrow$ *P* (*Conj xset*)
and $\bigwedge$*formula*. *P formula* $\Longrightarrow$ *P* (*Not formula*)
and $\bigwedge$*pred*. *P* (*Pred pred*)
and $\bigwedge$*act formula*. *P formula* $\Longrightarrow$ *P* (*Act act formula*)
shows *P x*
proof (*induction x*)
fix t_α :: ('*a*,'*b*,'*c*) *Tree*$_\alpha$
assume $t_\alpha \in \{t_\alpha.$ *hereditarily-fs* $t_\alpha\}$
then have *hereditarily-fs* t_α

```
    by simp
  then show P (Abs-formula t_α)
    proof (induction t_α)
      case (Conj_α tset_α) show ?case
        proof −
          let ?tset = map-bset Abs-formula tset_α
           have ⋀t_α'. t_α' ∈ set-bset tset_α ⟹ t_α' = (Rep-formula ∘ Abs-formula)
t_α'
            by (simp add: Conj_α.hyps)
          then have tset_α = map-bset (Rep-formula ∘ Abs-formula) tset_α
            by (metis bset.map-cong0 bset.map-id id-apply)
          then have ∗: tset_α = map-bset Rep-formula ?tset
            by (metis bset.map-comp)
          then have Abs-formula (Conj_α tset_α) = Conj ?tset
            by (metis Conj-def)
          moreover from ∗ have finite (supp ?tset)
            using Conj_α.hyps(1) by (metis supp-map-bset-Rep-formula)
          moreover have (⋀t. t ∈ set-bset ?tset ⟹ P t)
            using Conj_α.IH by (metis imageE map-bset.rep-eq)
          ultimately show ?thesis
            using assms(1) by metis
        qed
    next
      case Not_α then show ?case
     using assms(2) by (metis Formula.Abs-formula-inverse Not.rep-eq Rep-formula-inverse)
    next
      case Pred_α show ?case
        using assms(3) by (metis Pred.abs-eq)
    next
      case Act_α 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 ⋀xset c. finite (supp xset) ⟹ (⋀x. x ∈ set-bset xset ⟹ (⋀c. P c x))
⟹ P c (Conj xset)
    and ⋀formula c. (⋀c. P c formula) ⟹ P c (Not formula)
    and ⋀pred c. P c (Pred pred)
     and ⋀act formula c. bn act ♯∗ c ⟹ (⋀c. P c formula) ⟹ P c (Act act
formula)
  shows ⋀(c :: 'd::fs) p. P c (p · x)
proof (induction x)
  case (Conj xset)
    moreover then have finite (supp (p · xset))
      by (metis permute-finite supp-eqvt)
```

```
    moreover have (⋀x c. x ∈ set-bset (p ∙ xset) ⟹ 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 α x) show ?case
  proof −
    — rename bn (p ∙ α) to avoid c, without touching Act (p ∙ α) (p ∙ x)
    obtain q where 1: (q ∙ bn (p ∙ α)) ♯* c and 2: supp (Act (p ∙ α) (p ∙ x)) ♯* q
      proof (rule at-set-avoiding2[of bn (p ∙ α) c Act (p ∙ α) (p ∙ x), THEN exE])
        show finite (bn (p ∙ α)) by (fact bn-finite)
      next
        show finite (supp c) by (fact finite-supp)
      next
        show finite (supp (Act (p ∙ α) (p ∙ x))) by (simp add: finite-supp)
      next
       show bn (p ∙ α) ♯* Act (p ∙ α) (p ∙ x) by (simp add: fresh-def fresh-star-def)
      qed metis
    from 1 have bn (q ∙ p ∙ α) ♯* c
      by (simp add: bn-eqvt)
    moreover from Act.IH have ⋀c. P c (q ∙ p ∙ x)
      by (metis permute-plus)
    ultimately have P c (Act (q ∙ p ∙ α) (q ∙ p ∙ x))
      using assms(4) by simp
    moreover from 2 have Act (q ∙ p ∙ α) (q ∙ p ∙ x) = Act (p ∙ α) (p ∙ 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** $x =_\alpha y \Longrightarrow P$
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-eqvt' 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*) *Tree* $\times$ ('*d*, '*e*, '*f*) *Tree*
assume $x \in$ (*hull-rel O Tree-wf*) *O alpha-Tree-rel*
then obtain *x1 x2 x3 x4* **where** *x*: $x = (x1,x4)$ **and** *1*: $(x1,x2) \in$ *hull-rel* **and** *2*: $(x2,x3) \in$ *Tree-wf* **and** *3*: $(x3,x4) \in$ *alpha-Tree-rel*
by *auto*
from *2* **have** $(x1,x4) \in$ *alpha-Tree-rel O hull-rel O Tree-wf*
using *1* **and** *3* **proof** (*induct rule*: *Tree-wf.induct*)
— *tConj*
fix *t* **and** *tset* :: ('*d*,'*e*,'*f*) *Tree set*['*d*]
assume $*$: $t \in$ *set-bset tset* **and** $**$: $(x1,t) \in$ *hull-rel* **and** $***$: (*tConj tset*, *x4*) $\in$ *alpha-Tree-rel*
from $**$ **obtain** p **where** *x1*: $x1 = p \cdot t$
using *hull-rel.cases* **by** *blast*
from $***$ **have** *tConj tset* $=_\alpha$ *x4*
by (*rule alpha-Tree-relE*)
then obtain *tset'* **where** *x4*: $x4 =$ *tConj tset'* **and** *rel-bset* $(=_\alpha)$ *tset tset'*
by (*cases x4*) *simp-all*
with $*$ **obtain** t' **where** t': $t' \in$ *set-bset tset'* **and** $t =_\alpha t'$
by (*metis rel-bset.rep-eq rel-set-def*)
with *x1* **have** $(x1,\ p \cdot t') \in$ *alpha-Tree-rel*
by (*metis* $Tree_\alpha$*.abs-eq-iff alpha-Tree-relI permute-*$Tree_\alpha$*.abs-eq*)
moreover have $(p \cdot t',\ t') \in$ *hull-rel*
by (*rule hull-rel.intros*)
moreover from *x4* **and** *t'* **have** $(t',\ x4) \in$ *Tree-wf*
by (*simp add*: *Tree-wf.intros*(*1*))
ultimately show $(x1,x4) \in$ *alpha-Tree-rel O hull-rel O Tree-wf*

```
      by auto
    next
     — tNot
     fix t
     assume ∗: (x1,t) ∈ hull-rel and ∗∗: (tNot t, x4) ∈ alpha-Tree-rel
     from ∗ obtain p where x1: x1 = p · t
      using hull-rel.cases by blast
     from ∗∗ have tNot t =α x4
      by (rule alpha-Tree-relE)
     then obtain t′ where x4: x4 = tNot t′ and t =α t′
      by (cases x4) simp-all
     with x1 have (x1, p · t′) ∈ alpha-Tree-rel
      by (metis Treeα.abs-eq-iff alpha-Tree-relI permute-Treeα.abs-eq x1)
     moreover have (p · t′, t′) ∈ hull-rel
      by (rule hull-rel.intros)
     moreover from x4 have (t′, x4) ∈ Tree-wf
      using Tree-wf.intros(2) by blast
     ultimately show (x1,x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
      by auto
    next
     — tAct
     fix α t
     assume ∗: (x1,t) ∈ hull-rel and ∗∗: (tAct α t, x4) ∈ alpha-Tree-rel
     from ∗ obtain p where x1: x1 = p · t
      using hull-rel.cases by blast
     from ∗∗ have tAct α t =α x4
      by (rule alpha-Tree-relE)
     then obtain q t′ where x4: x4 = tAct (q · α) t′ and q · t =α t′
      by (cases x4) (auto simp add: alpha-set)
     with x1 have (x1, p · −q · t′) ∈ alpha-Tree-rel
     by (metis Treeα.abs-eq-iff alpha-Tree-relI permute-Treeα.abs-eq permute-minus-cancel(1))
     moreover have (p · −q · t′, t′) ∈ hull-rel
      by (metis hull-rel.simps permute-plus)
     moreover from x4 have (t′, x4) ∈ Tree-wf
      by (simp add: Tree-wf.intros(3))
     ultimately show (x1,x4) ∈ alpha-Tree-rel O hull-rel O Tree-wf
      by auto
    qed
   with x show x ∈ alpha-Tree-rel O hull-rel O Tree-wf
    by simp
  qed
qed

lemma alpha-Tree-rel-relcomp-trivialI [simp]:
  assumes (x, y) ∈ R
  shows (x, y) ∈ alpha-Tree-rel O R
using assms unfolding alpha-Tree-rel-def
by (metis Treeα.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 $(x, y) \in$ *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 ⇒ ('idx,'pred,'act) Tree ⇒ bool where
    valid-Tree P (tConj tset) ⟷ (∀ t∈set-bset tset. valid-Tree P t)
  | valid-Tree P (tNot t) ⟷ ¬ valid-Tree P t
  | valid-Tree P (tPred φ) ⟷ P ⊢ φ
  | valid-Tree P (tAct α t) ⟷ (∃α' t' P'. tAct α t =α tAct α' t' ∧ P → ⟨α',P'⟩
∧ 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 ∈ set-bset tset then show ((P, t), (P, tConj tset)) ∈ ?R
        by (simp add: Tree-wf.intros(1))
    next
      fix P :: 'state and t :: ('idx,'pred,'act) Tree
      show ((P, t), (P, tNot t)) ∈ ?R
        by (simp add: Tree-wf.intros(2))
    next
      fix P1 P2 :: 'state and α1 α2 :: 'act and t1 t2 :: ('idx,'pred,'act) Tree
      assume tAct α1 t1 =α tAct α2 t2
      then obtain p where t2 =α p • t1
        by (auto simp add: alphas) (metis alpha-Tree-symp sympE)
      then show ((P2, t2), (P1, tAct α1 t1)) ∈ ?R
        by (simp add: Tree-wf.intros(3))
    }
  qed
```

valid-Tree is equivariant.

```
lemma valid-Tree-eqvt': valid-Tree P t ⟷ valid-Tree (p · P) (p · t)
proof (induction P t rule: valid-Tree.induct)
  case (1 P tset) show ?case
    proof
      assume ∗: valid-Tree P (tConj tset)
      {
        fix t
        assume t ∈ p · set-bset tset
        with 1.IH and ∗ have valid-Tree (p · P) t
       by (metis (no-types, lifting) imageE permute-set-eq-image valid-Tree.simps(1))
      }
      then show valid-Tree (p · P) (p · tConj tset)
        by simp
    next
      assume ∗: valid-Tree (p · P) (p · tConj tset)
      {
        fix t
        assume t ∈ set-bset tset
        with 1.IH and ∗ have valid-Tree P t
       by (metis mem-permute-iff permute-Tree-tConj set-bset-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

  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 =α t2
    shows valid-Tree P t1 ⟷ 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 α1 t1 α2 t2) show ?case
    proof
      assume valid-Tree P (tAct α1 t1)
       then obtain α′ t′ P′ where tAct α1 t1 =α tAct α′ t′ ∧ P → ⟨α′,P′⟩ ∧
valid-Tree P′ t′
        by auto
      moreover from tAct.hyps have tAct α1 t1 =α tAct α2 t2
        using alpha-tAct by blast
      ultimately show valid-Tree P (tAct α2 t2)
        by (metis Treeα.abs-eq-iff valid-Tree.simps(4))
    next
      assume valid-Tree P (tAct α2 t2)
       then obtain α′ t′ P′ where tAct α2 t2 =α tAct α′ t′ ∧ P → ⟨α′,P′⟩ ∧
valid-Tree P′ t′
        by auto
      moreover from tAct.hyps have tAct α1 t1 =α tAct α2 t2
        using alpha-tAct by blast
      ultimately show valid-Tree P (tAct α1 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 ⇒ ('idx,'pred,'act) Treeα ⇒ bool is
    valid-Tree
  by (fact alpha-Tree-valid-Tree)
```

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

lemma *valid-Tree$_\alpha$-Conj$_\alpha$* [*simp*]: *valid-Tree$_\alpha$ P* (*Conj$_\alpha$ tset$_\alpha$*) ⟷ (∀ *t$_\alpha$*∈*set-bset tset$_\alpha$*. *valid-Tree$_\alpha$ P t$_\alpha$*)
proof −
have *valid-Tree P* (*rep-Tree$_\alpha$* (*abs-Tree$_\alpha$* (*tConj* (*map-bset rep-Tree$_\alpha$ tset$_\alpha$*)))) ⟷ *valid-Tree P* (*tConj* (*map-bset rep-Tree$_\alpha$ tset$_\alpha$*))
by (*metis Tree$_\alpha$-rep-abs alpha-Tree-valid-Tree*)
then show *?thesis*
by (*simp add*: *valid-Tree$_\alpha$-def Conj$_\alpha$-def map-bset.rep-eq*)
qed

lemma *valid-Tree$_\alpha$-Not$_\alpha$* [*simp*]: *valid-Tree$_\alpha$ P* (*Not$_\alpha$ t$_\alpha$*) ⟷ ¬ *valid-Tree$_\alpha$ P t$_\alpha$*
by *transfer simp*

lemma *valid-Tree$_\alpha$-Pred$_\alpha$* [*simp*]: *valid-Tree$_\alpha$ P* (*Pred$_\alpha$ φ*) ⟷ *P* ⊢ *φ*
by *transfer simp*

lemma *valid-Tree$_\alpha$-Act$_\alpha$* [*simp*]: *valid-Tree$_\alpha$ P* (*Act$_\alpha$ α t$_\alpha$*) ⟷ (∃ *α' t$_\alpha$' P'*. *Act$_\alpha$ α t$_\alpha$* = *Act$_\alpha$ α' t$_\alpha$'* ∧ *P* → ⟨*α'*,*P'*⟩ ∧ *valid-Tree$_\alpha$ P' t$_\alpha$'*)
proof
assume *valid-Tree$_\alpha$ P* (*Act$_\alpha$ α t$_\alpha$*)
moreover have *Act$_\alpha$ α t$_\alpha$* = *abs-Tree$_\alpha$* (*tAct α* (*rep-Tree$_\alpha$ t$_\alpha$*))
by (*metis Act$_\alpha$.abs-eq Tree$_\alpha$-abs-rep*)
ultimately show ∃ *α' t$_\alpha$' P'*. *Act$_\alpha$ α t$_\alpha$* = *Act$_\alpha$ α' t$_\alpha$'* ∧ *P* → ⟨*α'*,*P'*⟩ ∧ *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 ∃ *α' t$_\alpha$' P'*. *Act$_\alpha$ α t$_\alpha$* = *Act$_\alpha$ α' t$_\alpha$'* ∧ *P* → ⟨*α'*,*P'*⟩ ∧ *valid-Tree$_\alpha$ P' t$_\alpha$'*
moreover have ⋀*α' t$_\alpha$'*. *Act$_\alpha$ α' t$_\alpha$'* = *abs-Tree$_\alpha$* (*tAct α'* (*rep-Tree$_\alpha$ t$_\alpha$'*))
by (*metis Act$_\alpha$.abs-eq Tree$_\alpha$-abs-rep*)
ultimately show *valid-Tree$_\alpha$ P* (*Act$_\alpha$ α 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* ⇒ (*'idx*,*'pred*,*'act*) *formula* ⇒ *bool* (**infix** ⊨ *70*) **is**
valid-Tree$_\alpha$
.

lemma *valid-eqvt* :
assumes *P* ⊨ *x* **shows** (*p* · *P*) ⊨ (*p* · *x*)
using *assms* **by** *transfer* (*metis valid-Tree$_\alpha$-eqvt*)

lemma *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*: *valid-def Conj-def map-bset.rep-eq*)

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

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

lemma *valid-Act*: $P \models Act\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x')$
proof
assume $P \models Act\ \alpha\ x$
moreover have $Rep\text{-}formula\ (Abs\text{-}formula\ (Act_\alpha\ \alpha\ (Rep\text{-}formula\ x))) = Act_\alpha\ \alpha\ (Rep\text{-}formula\ x)$
by (*metis Act.rep-eq Rep-formula-inverse*)
ultimately show $\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$
by (*auto simp add*: *valid-def Act-def*) (*metis Abs-formula-inverse Rep-formula' hereditarily-fs-alpha-renaming*)
next
assume $\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$
then show $P \models Act\ \alpha\ x$
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 $P \models Act\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge bn\ \alpha' \sharp* X)$
proof
assume $P \models Act\ \alpha\ x$
then obtain $\alpha'\ x'\ P'$ **where** *eq*: $Act\ \alpha\ x = Act\ \alpha'\ x'$ **and** *trans*: $P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$
by (*metis valid-Act*)
have *finite* (*bn* α')
by (*fact bn-finite*)
moreover note ‹*finite* (*supp X*)›
moreover have *finite* (*supp* ($Act\ \alpha'\ x'$, $\langle \alpha', P' \rangle$))
by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)
moreover have *bn* $\alpha' \sharp*$ ($Act\ \alpha'\ x'$, $\langle \alpha', P' \rangle$)
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
ultimately obtain p **where** *fresh-X*: $(p \cdot bn\ \alpha') \sharp* X$ **and** *supp* ($Act\ \alpha'\ x'$, $\langle \alpha', P' \rangle$) $\sharp* p$
by (*metis at-set-avoiding2*)

then have *supp* (*Act* α' x') $\sharp*$ p **and** *supp* $\langle\alpha',P'\rangle$ $\sharp*$ p
by (*metis fresh-star-Un supp-Pair*)+
then have *Act* ($p \cdot \alpha'$) ($p \cdot x'$) = *Act* α' x' **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'. *Act* α x = *Act* α' x' $\wedge$ $P \rightarrow \langle\alpha',P'\rangle \wedge P' \models x' \wedge$ *bn* α' $\sharp*$ X
using *eq* **and** *trans* **and** *valid* **and** *fresh-X* **by** (*metis bn-eqvt valid-eqvt*)
next
assume $\exists\alpha'$ x' P'. *Act* α x = *Act* α' x' $\wedge$ $P \rightarrow \langle\alpha',P'\rangle \wedge P' \models x' \wedge$ *bn* α' $\sharp*$ X
then show $P \models$ *Act* α x
by (*metis valid-Act*)
qed

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

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

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

from *assms* **and** *fresh* **have** (*bn* $\alpha \cup p \cdot$ *bn* α) $\sharp*$ P
using *p-α* **by** (*metis bn-eqvt fresh-star-Un*)
then have *supp* p $\sharp*$ P
using *supp-p* **by** (*metis fresh-star-def subset-eq*)
then have *p-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-P p-α* **by** (*metis permute-minus-cancel*(*1*) *transition-eqvt'*)
moreover from *valid* **have** $-p \cdot P' \models x$
using *p-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** $P \models$ *Act* α 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* ⇒ *'pred*::*fs* ⇒ *bool* (**infix** ⊢ *70*)
  and *transition* :: *'state* ⇒ (*'act*::*bn*,*'state*) *residual* ⇒ *bool* (**infix** → *70*) +
  assumes *card-idx-perm*: |*UNIV*::*perm set*| <*o* |*UNIV*::*'idx set*|
    and *card-idx-state*: |*UNIV*::*'state set*| <*o* |*UNIV*::*'idx set*|
begin

  definition *logically-equivalent* :: *'state* ⇒ *'state* ⇒ *bool* **where**
    logically-equivalent P Q ≡ (∀ *x*::(*'idx*,*'pred*,*'act*) *formula*. *P* ⊨ *x* ⟷ *Q* ⊨ *x*)

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

  lemma *logically-equivalent-eqvt*:
    assumes *P* =· *Q* **shows** *p* ∙ *P* =· *p* ∙ *Q*
  using *assms* **unfolding** *logically-equivalent-def*
  by (*metis* (*mono-tags*) *permute-minus-cancel*(*1*) *valid-eqvt*)

end

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

8 Bisimilarity Implies Logical Equivalence

context *indexed-nominal-ts*
begin

  lemma *bisimilarity-implies-equivalence-Act*:
    assumes ⋀*P Q*. *P* ∼· *Q* ⟹ *P* ⊨ *x* ⟷ *Q* ⊨ *x*
    and *P* ∼· *Q*
    and *P* ⊨ *Act α x*
    shows *Q* ⊨ *Act α x*

```
  proof −
    have finite (supp Q)
      by (fact finite-supp)
     with ‹P ⊨ Act α x› obtain α' x' P' where eq: Act α x = Act α' x' and
trans: P → ⟨α',P'⟩ and valid: P' ⊨ x' and fresh: bn α' ♯* Q
      by (metis valid-Act-strong)

     from ‹P ∼· Q› and fresh and trans obtain Q' where trans': Q → ⟨α',Q'⟩
and bisim': P' ∼· Q'
      by (metis bisimilar-simulation-step)

    from eq obtain p where px: x' = p · x
      by (metis Act-eq-iff-perm)

    with valid have −p · P' ⊨ x
      by (metis permute-minus-cancel(1) valid-eqvt)
    moreover from bisim' have (−p · P') ∼· (−p · Q')
      by (metis bisimilar-eqvt)
    ultimately have −p · Q' ⊨ x
      using ‹⋀P Q. P ∼· Q ⟹ P ⊨ x ⟷ Q ⊨ x› by metis
    with px have Q' ⊨ x'
      by (metis permute-minus-cancel(1) valid-eqvt)

    with eq and trans' show Q ⊨ Act α x
      unfolding valid-Act by metis
  qed

  theorem bisimilarity-implies-equivalence: assumes P ∼· Q shows P =· Q
  unfolding logically-equivalent-def proof
    fix x :: ('idx, 'pred, 'act) formula
    from assms show P ⊨ x ⟷ Q ⊨ 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 α 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 *distinguishing-formula* :: (*'idx*, *'pred*, *'act*) *formula* ⇒ *'state* ⇒ *'state* ⇒ *bool*
(- *distinguishes* - *from* - [*100*,*100*,*100*] *100*)
where
x distinguishes P from Q ≡ *P* ⊨ *x* ∧ ¬ *Q* ⊨ *x*

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

lemma *equivalent-iff-not-distinguished*: (*P* =· *Q*) ⟷ ¬(∃ *x*. *x distinguishes P from Q*)
by (*metis* (*full-types*) *distinguishing-formula-def logically-equivalent-def valid-Not*)

There exists a distinguishing formula for P and Q whose support is contained in *supp P*.

lemma *distinguished-bounded-support*:
assumes *x distinguishes P from Q*
obtains *y* **where** *supp y* ⊆ *supp P* **and** *y distinguishes P from Q*
proof −
let *?B* = {*p* ∙ *x*|*p*. *supp P* ♯* *p*}
have *supp P supports ?B*
unfolding *supports-def* **proof** (*clarify*)
fix *a b*
assume *a*: *a* ∉ *supp P* **and** *b*: *b* ∉ *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
   from finite-supp-B and card-B and supp-B-subset-supp-P have supp ?y ⊆
supp P
    by simp
  moreover have ?y distinguishes P from Q
    unfolding distinguishing-formula-def proof
      from assms show P ⊨ ?y
       by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
supp-perm-eq valid-eqvt)
    next
      from assms show ¬ Q ⊨ ?y
       by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
    qed
  ultimately show ?thesis ..
qed

lemma equivalence-is-bisimulation: is-bisimulation logically-equivalent
proof −
```

```
  have symp logically-equivalent
    by (metis logically-equivalent-def sympI)
  moreover
  {
    fix P Q 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' ∧ x distinguishes P' from Q'
            by (metis distinguished-bounded-support)
          then obtain f :: 'state ⇒ ('idx, 'pred, 'act) formula where
            ∗: ∀ Q'∈?Q'. supp (f Q') ⊆ supp P' ∧ (f Q') distinguishes P' from Q'
            by metis
          have supp P' supports (f ‘ ?Q')
            unfolding supports-def proof (clarify)
              fix a b
              assume a: a ♯ supp P' and b: b ♯ supp P'
              have (a ⇌ b) · (f ‘ ?Q') ⊆ f ‘ ?Q'
              proof
                fix x'
                assume x' ∈ (a ⇌ b) · (f ‘ ?Q')
               then obtain Q' where 1: x' = (a ⇌ b) · f Q' and 2: Q → ⟨α,Q'⟩
                  by auto (metis permute-swap-cancel transition-eqvt')
                with ∗ and a and b have a ♯ supp (f Q') and b ♯ 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' ∈ f ‘ ?Q'
                  by simp
              qed
              moreover have f ‘ ?Q' ⊆ (a ⇌ b) · (f ‘ ?Q')
              proof
                fix x'
                assume x' ∈ f ‘ ?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 P ⊨ Act α ?y
      unfolding valid-Act proof (standard+)
        show P → ⟨α,P'⟩ by fact
      next
        {
          fix Q'
          assume Q → ⟨α,Q'⟩
          with ∗ have P' ⊨ f Q'
            by (metis distinguishing-formula-def mem-Collect-eq)
        }
        then show P' ⊨ ?y
          by (simp add: finite-supp-image card-image)
      qed
    moreover have ¬ Q ⊨ Act α ?y
      proof
        assume Q ⊨ Act α ?y
        then obtain Q' where 1: Q → ⟨α,Q'⟩ and 2: Q' ⊨ ?y
          using ‹bn α ♯* Q› by (metis valid-Act-fresh)
        from 2 have ⋀Q''. Q → ⟨α,Q''⟩ ⟶ Q' ⊨ f Q''
          by (simp add: finite-supp-image card-image)
        with 1 and ∗ show False
          using distinguishing-formula-def by blast
      qed
    ultimately have False
      by (metis ‹P =· Q› logically-equivalent-def)
  }
  then show ?thesis by auto
 qed
}
ultimately show ?thesis
```

unfolding *is-bisimulation-def* **by** *metis*
qed

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

end

end
theory *FS-Set*
imports
Nominal2.Nominal2
begin

10 Finitely Supported Sets

We define the type of finitely supported sets (over some permutation type $'a$).

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

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

setup-lifting *type-definition-fs-set*

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

instantiation *fs-set* :: (*pt*) *pt*
begin

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

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

end

instantiation *fs-set* :: (*pt*) *fs*
begin

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

hence *finite* {*a*. *infinite* {*b*. (*a* ⇌ *b*) · *Rep-fs-set x* ≠ *Rep-fs-set x*}} **by** (*unfold supp-def*)
hence *finite* {*a*. *infinite* {*b*. ((*a* ⇌ *b*) · *x*) ≠ *x*}} **by** *transfer*
thus *finite* (*supp x*) **by** (*fold supp-def*)
qed

end

Set membership.

lift-definition *member-fs-set* :: *'a*::*pt* ⇒ *'a fs-set* ⇒ *bool* **is** (∈) **.**

notation
member-fs-set (*'*(∈$_{fs}$*'*)) **and**
member-fs-set ((-/ ∈$_{fs}$ -) [*51*, *51*] *50*)

lemma *member-fs-set-permute-iff* [*simp*]: *p* · *x* ∈$_{fs}$ *p* · *X* ⟷ *x* ∈$_{fs}$ *X*
by *transfer* (*simp add*: *mem-permute-iff*)

lemma *member-fs-set-eqvt* [*eqvt*]: *x* ∈$_{fs}$ *X* ⟹ *p* · *x* ∈$_{fs}$ *p* · *X*
by *simp*

end
theory *FL-Transition-System*
imports
Transition-System FS-Set
begin

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

11.1 Nominal transition systems with effects

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

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

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

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

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

type-synonym (*'a*,*'eff*) *later* = *'a* × *'eff first* × *'eff* ⇒ *'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-eqvt*: *eqvt effect-apply*
      and *L-eqvt*: *eqvt L* — *L* is assumed to be equivariant.

begin

  lemma *effect-apply-eqvt-aux* [*simp*]: *p* ∙ *effect-apply* = *effect-apply*
  by (*metis effect-apply-eqvt eqvt-def*)

  lemma *effect-apply-eqvt'* [*eqvt*]: *p* ∙ ⟨*f*⟩*P* = ⟨*p* ∙ *f*⟩(*p* ∙ *P*)
  by *simp*

  lemma *L-eqvt-aux* [*simp*]: *p* ∙ *L* = *L*
  by (*metis L-eqvt eqvt-def*)

  lemma *L-eqvt'* [*eqvt*]: *p* ∙ *L* (*α*, *P*, *f*) = *L* (*p* ∙ *α*, *p* ∙ *P*, *p* ∙ *f*)
  by *simp*

end

11.2 L-bisimulations and F/L-bisimilarity

context *effect-nominal-ts*
begin

  definition *is-L-bisimulation*:: (*'effect first* ⇒ *'state* ⇒ *'state* ⇒ *bool*) ⇒ *bool*
where
    is-L-bisimulation R ≡
      ∀ *F*. *symp* (*R F*) ∧
        (∀ *P Q*. *R F P Q* ⟶ (∀ *f*. *f* ∈*fs F* ⟶ (∀ *φ*. ⟨*f*⟩*P* ⊢ *φ* ⟶ ⟨*f*⟩*Q* ⊢ *φ*))) ∧
        (∀ *P Q*. *R F P Q* ⟶ (∀ *f*. *f* ∈*fs F* ⟶ (∀ *α P'*. *bn α* ♯* (⟨*f*⟩*Q*, *F*, *f*) ⟶
              ⟨*f*⟩*P* → ⟨*α*,*P'*⟩ ⟶ (∃ *Q'*. ⟨*f*⟩*Q* → ⟨*α*,*Q'*⟩ ∧ *R* (*L* (*α*,*F*,*f*)) *P'* *Q'*))))

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

  abbreviation *FL-bisimilar'* (- ∼∙[-] - [*51*,*0*,*51*] *50*) **where**
    P ∼∙[*F*] *Q* ≡ *FL-bisimilar F P Q*

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

  lemma *is-L-bisimulation-eqvt* [*eqvt*]:
    assumes *is-L-bisimulation R* **shows** *is-L-bisimulation* (*p* ∙ *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 $\forall P\ Q.\ (p \cdot R)\ F\ P\ Q \longrightarrow (\forall f.\ f \in_{fs} F \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha \sharp* (\langle f\rangle Q, F, f) \longrightarrow \langle f\rangle P \to \langle \alpha, P'\rangle \longrightarrow (\exists Q'.\ \langle f\rangle Q \to \langle \alpha, Q'\rangle \wedge (p \cdot R)\ (L\ (\alpha, F, f))\ P'\ Q')))$ (**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*: $bn\ \alpha \sharp* (\langle f\rangle Q, F, f)$ **and** *trans*: $\langle f\rangle P \to \langle \alpha, P'\rangle$
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 $bn\ (-p \cdot \alpha) \sharp* (\langle -p \cdot f\rangle(-p \cdot Q), -p \cdot F, -p \cdot f)$
using *fresh* **by** (*metis* (*full-types*) *effect-apply-eqvt′ bn-eqvt fresh-star-Pair fresh-star-permute-iff*)
moreover have $\langle -p \cdot f\rangle(-p \cdot P) \to \langle -p \cdot \alpha, -p \cdot P'\rangle$
using *trans* **by** (*metis effect-apply-eqvt′ transition-eqvt′*)
ultimately obtain Q' **where** *T*: $\langle -p \cdot f\rangle(-p \cdot Q) \to \langle -p \cdot \alpha, Q'\rangle$ **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** $\langle f\rangle Q \to \langle \alpha, p \cdot Q'\rangle$
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 $\exists Q'.\ \langle f\rangle Q \to \langle \alpha, Q'\rangle \wedge (p \cdot R)\ (L\ (\alpha,F,f))\ P'\ Q'$
by *metis*

qed
ultimately show *?S* ∧ *?T* ∧ *?U* **by** *simp*
qed

lemma *FL-bisimilar-eqvt*:
assumes *P* ∼·[*F*] *Q* **shows** (*p* · *P*) ∼·[*p* · *F*] (*p* · *Q*)
using *assms*
by (*metis eqvt-apply permute-boolI is-L-bisimulation-eqvt FL-bisimilar-def*)

lemma *FL-bisimilar-reflp*: *reflp* (*FL-bisimilar F*)
proof (*rule reflpI*)
fix *x*
have *is-L-bisimulation* (λ-. (=))
unfolding *is-L-bisimulation-def* **by** (*simp add*: *symp-def*)
then show *x* ∼·[*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* ∼·[*F*] *Q*
then obtain *R* **where** ∗: *is-L-bisimulation R* ∧ *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* ∼·[*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 ∀ *P Q*. *P* ∼·[*F*] *Q* ⟶ (∀ *f*. *f* ∈$_{fs}$ *F* ⟶ (∀ φ. ⟨*f*⟩*P* ⊢ φ ⟶ ⟨*f*⟩*Q* ⊢ φ)) (**is** *?S*)
by (*auto simp add*: *is-L-bisimulation-def FL-bisimilar-def*)
moreover have ∀ *P Q*. *P* ∼·[*F*] *Q* ⟶ (∀ *f*. *f* ∈$_{fs}$ *F* ⟶ (∀ α *P′*. *bn* α ♯∗ (⟨*f*⟩*Q*, *F*, *f*) ⟶
⟨*f*⟩*P* → ⟨α,*P′*⟩ ⟶ (∃ *Q′*. ⟨*f*⟩*Q* → ⟨α,*Q′*⟩ ∧ *P′* ∼·[*L* (α, *F*, *f*)] *Q′*))) (**is** *?T*)
by (*auto simp add*: *is-L-bisimulation-def FL-bisimilar-def*) *blast*
ultimately show *?R* ∧ *?S* ∧ *?T*
by *metis*
qed

lemma *FL-bisimilar-simulation-step*:
assumes *P* ∼·[*F*] *Q* **and** *f* ∈$_{fs}$ *F* **and** *bn* α ♯∗ (⟨*f*⟩*Q*, *F*, *f*) **and** ⟨*f*⟩*P* →

$\langle\alpha,P'\rangle$
obtains Q' **where** $\langle f\rangle Q \to \langle\alpha,Q'\rangle$ **and** $P' \sim\cdot[L\ (\alpha,F,f)]\ Q'$
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*: $P \sim\cdot[F]\ Q$ **and** *QR*: $Q \sim\cdot[F]\ R$
let *?FL-bisim* = $\lambda F.$ (*FL-bisimilar F*) *OO* (*FL-bisimilar F*)
have $\bigwedge F.$ *symp* (*?FL-bisim F*)
proof (*rule sympI*)
fix *F P R*
assume *?FL-bisim F P R*
then obtain *Q* **where** $P \sim\cdot[F]\ Q$ **and** $Q \sim\cdot[F]\ R$
by *blast*
then have $R \sim\cdot[F]\ Q$ **and** $Q \sim\cdot[F]\ P$
by (*metis FL-bisimilar-symp sympE*)+
then show *?FL-bisim F R P*
by *blast*
qed
moreover have $\bigwedge F.\ \forall P\ Q.$ *?FL-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-bisim F P Q* $\longrightarrow$
$(\forall f.\ f \in_{fs} F \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha\ \sharp*\ (\langle f\rangle Q,\ F,\ f) \longrightarrow$
$\langle f\rangle P \to \langle\alpha,P'\rangle \longrightarrow (\exists Q'.\ \langle f\rangle Q \to \langle\alpha,Q'\rangle \wedge$ *?FL-bisim* $(L\ (\alpha,F,f))$ $P'\ Q')))$
proof (*clarify*)
fix *F P R Q f* α P'
assume *PR*: $P \sim\cdot[F]\ R$ **and** *RQ*: $R \sim\cdot[F]\ Q$ **and** *effect*: $f \in_{fs} F$ **and** *fresh*: *bn* α $\sharp*$ $(\langle f\rangle Q,\ F,\ f)$ **and** *trans*: $\langle f\rangle P \to \langle\alpha,P'\rangle$
— rename $\langle\alpha,P'\rangle$ to avoid $(\langle f\rangle R,\ F)$, without touching $(\langle f\rangle Q,\ F,\ f)$
obtain *p* **where** *1*: $(p \cdot bn\ \alpha)\ \sharp*\ (\langle f\rangle R,\ F,\ f)$ **and** *2*: *supp* $(\langle\alpha,P'\rangle, (\langle f\rangle Q,$ $F,\ f)) \sharp*\ p$
proof (*rule at-set-avoiding2*[*of bn* α $(\langle f\rangle R,\ F,\ f)$ $(\langle\alpha,P'\rangle, (\langle f\rangle Q,\ F,\ f))$, *THEN exE*])
show *finite* (*bn* α) **by** (*fact bn-finite*)
next
show *finite* (*supp* $(\langle f\rangle R,\ F,\ f)$) **by** (*fact finite-supp*)
next
show *finite* (*supp* $(\langle\alpha,P'\rangle, (\langle f\rangle Q,\ F,\ f)))$ **by** (*simp add*: *finite-supp supp-Pair*)
next
show *bn* α $\sharp*$ $(\langle\alpha,P'\rangle, (\langle f\rangle Q,\ F,\ f))$
using *bn-abs-residual-fresh fresh fresh-star-Pair* **by** *blast*
qed *metis*
from *2* **have** *3*: *supp* $\langle\alpha,P'\rangle$ $\sharp*$ *p* **and** *4*: *supp* $(\langle f\rangle Q,\ F,\ f)$ $\sharp*$ *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*: $(p \cdot P') \sim\cdot[L\ (p \cdot \alpha,F,f)]\ pR'$
using *PR effect trans 1* **by** (*metis FL-bisimilar-simulation-step bn-eqvt*)
from *fresh* **and** *4* **have** *bn* $(p \cdot \alpha) \sharp* (\langle f\rangle Q,\ F,\ f)$
by (*metis bn-eqvt fresh-star-permute-iff supp-perm-eq*)
then obtain pQ' **where** *7*: $\langle f\rangle Q \rightarrow \langle p \cdot \alpha,\ pQ'\rangle$ **and** *8*: $pR' \sim\cdot[L\ (p \cdot \alpha,F,f)]\ pQ'$
using *RQ effect 5* **by** (*metis FL-bisimilar-simulation-step*)
from *4* **have** *supp* $(\langle f\rangle Q) \sharp* p$
by (*simp add*: *fresh-star-Un supp-Pair*)
with *7* **have** $\langle f\rangle Q \rightarrow \langle \alpha,\ -p \cdot pQ'\rangle$
by (*metis permute-minus-cancel*(*2*) *supp-perm-eq transition-eqvt'*)
moreover from *6* **and** *8* **have** *?FL-bisim* $(L\ (p \cdot \alpha,\ F,\ f))\ (p \cdot P')\ pQ'$
by (*metis relcompp.relcompI*)
then have *?FL-bisim* $(-p \cdot L\ (p \cdot \alpha,\ F,\ f))\ (-p \cdot p \cdot P')\ (-p \cdot pQ')$
using *FL-bisimilar-eqvt* **by** *blast*
then have *?FL-bisim* $(L\ (\alpha,\ -p \cdot F,\ -p \cdot f))\ P'\ (-p \cdot pQ')$
by (*simp add*: *L-eqvt'*)
then have *?FL-bisim* $(L\ (\alpha,F,f))\ P'\ (-p \cdot pQ')$
using *4* **by** (*metis fresh-star-Un permute-minus-cancel*(*2*) *supp-Pair supp-perm-eq*)
ultimately show $\exists\, Q'.\ \langle f\rangle Q \rightarrow \langle \alpha,Q'\rangle \wedge$ *?FL-bisim* $(L\ (\alpha,F,f))\ P'\ Q'$
by *metis*
qed
ultimately have *is-L-bisimulation ?FL-bisim*
unfolding *is-L-bisimulation-def* **by** *metis*
moreover have *?FL-bisim F P R*
using *PQ QR* **by** *blast*
ultimately show $P \sim\cdot[F]\ R$
unfolding *FL-bisimilar-def* **by** *meson*
qed

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

end

end
theory *FL-Formula*
imports
Nominal-Bounded-Set
Nominal-Wellfounded
Residual
FL-Transition-System
begin

12 Infinitary Formulas With Effects

12.1 Infinitely branching trees

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

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

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

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

inductive-set *Tree-wf* :: $('idx,'pred,'act,'eff)$ *Tree rel* **where**
  $t \in$ *set-bset tset* $\Longrightarrow$ $(t,$ *tConj tset*$) \in$ *Tree-wf*
| $(t,$ *tNot* $t) \in$ *Tree-wf*
| $(t,$ *tAct* $f\ \alpha\ t) \in$ *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* $\Rightarrow$ *bool* **and** t
  assume $\forall x.\ (\forall y.\ (y,\ x) \in$ *Tree-wf* $\longrightarrow P\ y) \longrightarrow P\ x$
  then show $P\ t$
    proof (*induction* t)
      case *tConj* **then show** *?case*
        by (*metis Tree.distinct*(*2*) *Tree.distinct*(*5*) *Tree.inject*(*1*) *Tree-wf.cases*)
    next
      case *tNot* **then show** *?case*
        by (*metis Tree.distinct*(*1*) *Tree.distinct*(*9*) *Tree.inject*(*2*) *Tree-wf.cases*)
    next
      case *tPred* **then show** *?case*
       by (*metis Tree.distinct*(*11*) *Tree.distinct*(*3*) *Tree.distinct*(*7*) *Tree-wf.cases*)
    next
      case *tAct* **then show** *?case*
        by (*metis Tree.distinct*(*10*) *Tree.distinct*(*6*) *Tree.inject*(*4*) *Tree-wf.cases*)
    qed
qed

We define a permutation operation on the type of trees.

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

primrec *permute-Tree* :: *perm* ⇒ (-,-,-,-) *Tree* ⇒ (-,-,-,-) *Tree* **where**
  $p \cdot (tConj\ tset) = tConj\ (map\text{-}bset\ (permute\ p)\ tset)$ — 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*) ∈ *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** α
show $(p \cdot t,\ p \cdot tAct\ f\ \alpha\ t) \in$ *Tree-wf*
by (*metis Tree-wf.intros*(*3*) *permute-Tree.simps*(*4*))
qed

lemma *Tree-wf-eqvt* [*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-eqvt-aux*)
next
show *Tree-wf* $\subseteq p \cdot$ *Tree-wf*
by (*auto simp add*: *permute-set-def*) (*metis Tree-wf-eqvt-aux permute-minus-cancel*(*1*))
qed

lemma *Tree-wf-eqvt'*: *eqvt Tree-wf*
by (*metis Tree-wf-eqvt 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* φ) = *supp f* $\cup$ *supp* φ
unfolding *supp-def* **by** (*simp add*: *Collect-imp-eq Collect-neg-eq*)

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

12.2 Trees modulo α-equivalence

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

lemma *supp-rel-eqvt* [*eqvt*]:
$p \cdot$ *supp-rel R x* = *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* (*tConj tset*) = *supp-rel alpha-Tree* (*tConj tset*).

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

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

lemma *supp-rel-cong* [*fundef-cong*]:
$[\![x{=}x'; \bigwedge a\ b.\ R\ ((a \rightleftharpoons b) \cdot x')\ x' \longleftrightarrow R'\ ((a \rightleftharpoons b) \cdot x')\ x']\!] \Longrightarrow$ *supp-rel* $R\ x$ = *supp-rel* $R'\ x'$
by (*simp add*: *supp-rel-def*)

lemma *rel-bset-cong* [*fundef-cong*]:
$[\![x{=}x'; y{=}y'; \bigwedge a\ b.\ a \in$ *set-bset* $x' \Longrightarrow b \in$ *set-bset* $y' \Longrightarrow R\ a\ b \longleftrightarrow R'\ a\ b]\!]$
$\Longrightarrow$ *rel-bset* $R\ x\ y \longleftrightarrow$ *rel-bset* $R'\ x'\ y'$
by (*simp add*: *rel-bset-def rel-set-def*)

lemma *alpha-set-cong* [*fundef-cong*]:
$[\![bs{=}bs'; x{=}x'; R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y'; f\ x' = f'\ x'; f\ y' = f'\ y'; p{=}p'; cs{=}cs'; y{=}y']\!] \Longrightarrow$
alpha-set $(bs, x)\ R\ f\ p\ (cs, y) \longleftrightarrow$ *alpha-set* $(bs', x')\ R'\ f'\ p'\ (cs', y')$
by (*simp add*: *alpha-set*)

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

lemma *abs-Tree*$_p$*-eq* [*simp*]: *abs-Tree*$_p\ (p \cdot t) =$ *abs-Tree*$_p\ t$
by (*metis hull-relp.simps Tree*$_p$*.abs-eq-iff*)

lemma *permute-rep-abs-Tree*$_p$:
obtains p **where** *rep-Tree*$_p\ ($*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 *Tree-wf*$_p$ :: $('idx,'pred{::}pt,'act{::}bn,'eff{::}fs)\ Tree_p$ *rel* **is**
Tree-wf .

lemma *Tree-wf*$_p$*I* [*simp*]:
assumes $(a, b) \in$ *Tree-wf*
shows (*abs-Tree*$_p\ (p \cdot a)$, *abs-Tree*$_p\ b) \in$ *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 $(a,\ b) \in$ *Tree-wf*
shows $(\textit{abs-Tree}_p\ a,\ \textit{abs-Tree}_p\ b) \in \textit{Tree-wf}_p$
using *assms* **by** (*metis Tree-wf*$_p$*I permute-zero*)

lemma *Tree-wf*$_p$*E*:
assumes $(a_p,\ b_p) \in \textit{Tree-wf}_p$
obtains *a b* **where** $a_p = \textit{abs-Tree}_p\ a$ **and** $b_p = \textit{abs-Tree}_p\ b$ **and** $(a,b) \in$ *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*$_p$*E*)
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* $(p2 \cdot t1,\ p2 \cdot t2) \in$ *Tree-wf*)
apply (*subgoal-tac* $(p1 \cdot t1,\ p2 \cdot t1) \in$ *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* :: $('a,\ 'b,\ 'c,\ 'd)$ *Tree* $\times$ $('a,\ 'b,\ 'c,\ 'd)$ *Tree* $\Rightarrow$ $('a,\ 'b::pt,\ 'c::bn,\ 'd::fs)$ *Tree*$_p$ *set* **where**
alpha-Tree-termination $(t1,\ t2) = \{\textit{abs-Tree}_p\ t1,\ \textit{abs-Tree}_p\ t2\}$

Here it comes ...

function (*sequential*)
alpha-Tree :: $('idx,'pred::pt,'act::bn,'eff::fs)$ *Tree* $\Rightarrow$ $('idx,'pred,'act,'eff)$ *Tree* $\Rightarrow$ *bool* (**infix** $=_\alpha$ *50*) **where**
— $(=_\alpha)$
alpha-tConj: *tConj tset1* $=_\alpha$ *tConj tset2* $\longleftrightarrow$ *rel-bset alpha-Tree tset1 tset2*
| *alpha-tNot*: *tNot t1* $=_\alpha$ *tNot t2* $\longleftrightarrow$ *t1* $=_\alpha$ *t2*
| *alpha-tPred*: *tPred f1* $\varphi 1$ $=_\alpha$ *tPred f2* $\varphi 2$ $\longleftrightarrow$ $f1 = f2 \wedge \varphi 1 = \varphi 2$
— the action may have binding names
| *alpha-tAct*: *tAct f1* $\alpha 1$ *t1* $=_\alpha$ *tAct f2* $\alpha 2$ *t2* $\longleftrightarrow$
$f1 = f2 \wedge (\exists p.\ (bn\ \alpha 1,\ t1) \approx set\ alpha\text{-}Tree\ (supp\text{-}rel\ alpha\text{-}Tree)\ p\ (bn\ \alpha 2,\ t2) \wedge (bn\ \alpha 1,\ \alpha 1) \approx set\ ((=))\ supp\ p\ (bn\ \alpha 2,\ \alpha 2))$
| *alpha-other*: - $=_\alpha$ - $\longleftrightarrow$ *False*
— 254 subgoals (!)
by *pat-completeness auto*
termination
proof
let *?R* = *inv-image* (*max-ext Tree-wf*$_p$) *alpha-Tree-termination*
show *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* $\in$ *set-bset tset1* $\Longrightarrow$ *b* $\in$ *set-bset tset2* $\Longrightarrow$ *a* $=_\alpha$ *b* $\Longrightarrow$ *P a b*) $\Longrightarrow$
rel-bset ($=_\alpha$) *tset1 tset2* $\Longrightarrow$ *P* (*tConj tset1*) (*tConj tset2*)
and $\bigwedge$*t1 t2*. *t1* $=_\alpha$ *t2* $\Longrightarrow$ *P t1 t2* $\Longrightarrow$ *P* (*tNot t1*) (*tNot t2*)
and $\bigwedge$*f* φ. *P* (*tPred f* φ) (*tPred f* φ)
and $\bigwedge$*f1* α*1 t1 f2* α*2 t2*. ($\bigwedge$*p*. *p* $\cdot$ *t1* $=_\alpha$ *t2* $\Longrightarrow$ *P* (*p* $\cdot$ *t1*) *t2*) $\Longrightarrow$ *f1* = *f2* $\Longrightarrow$
($\exists$ *p*. (*bn* α*1*, *t1*) $\approx$*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) *p* (*bn* α*2*, *t2*) $\wedge$ (*bn* α*1*, α*1*) $\approx$*set* (=) *supp p* (*bn* α*2*, α*2*)) $\Longrightarrow$
P (*tAct f1* α*1 t1*) (*tAct f2* α*2 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-rel* ($=_\alpha$) *t* = *supp-rel* ($=_\alpha$) (*p* $\cdot$ *t*)
proof −
{
fix *a*
let *?B* = {*b*. ¬ ((*a* $\rightleftharpoons$ *b*) $\cdot$ *t*) $=_\alpha$ *t*}
let *?pB* = {*b*. ¬ ((*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 ⟷ 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 =α t2 ⟷ p • t1 =α p • 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) FL-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-eqvt)
   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) FL-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-eqvt)
   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-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** $p \cdot q \cdot -p \cdot bn\ (p \cdot \alpha 1) = bn\ (p \cdot \alpha 2)$
by (*simp add*: *alpha-set bn-eqvt*)
ultimately have (*bn* $(p \cdot \alpha 1)$, $p \cdot t1$) $\approx$*set* $(=_\alpha)$ (*supp-rel* $(=_\alpha)$) $(p + q - p)$ (*bn* $(p \cdot \alpha 2)$, $p \cdot t2$)
by (*simp add*: *alpha-set*)
moreover from *2* **have** (*bn* $(p \cdot \alpha 1)$, $p \cdot \alpha 1$) $\approx$*set* $(=)$ *supp* $(p + q - p)$ (*bn* $(p \cdot \alpha 2)$, $p \cdot \alpha 2$)
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 $p \cdot tAct\ f1\ \alpha 1\ t1 =_\alpha p \cdot tAct\ f2\ \alpha 2\ t2$ **using** *0*
by *auto*
next
assume $p \cdot tAct\ f1\ \alpha 1\ t1 =_\alpha p \cdot tAct\ f2\ \alpha 2\ t2$
then obtain *q* **where** *0*: *f1* = *f2* **and** *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* $\alpha 1$) $\sharp*$ $(-\ p + q + p)$
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* $\alpha 1$) $\sharp*$ $(-p + q + p)$
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 α1*, *α1*) ≈*set* (=) *supp* (−*p* + *q* + *p*) (*bn α2*, *α2*)
by (*simp add*: *alpha-set*)
}
ultimately show *tAct f1 α1 t1* =$_\alpha$ *tAct f2 α2 t2* **using** *0*
by *auto*
qed
qed *simp-all*

lemma *alpha-Tree-eqvt* [*eqvt*]: *t1* =$_\alpha$ *t2* ⟹ *p* ∙ *t1* =$_\alpha$ *p* ∙ *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* **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 α1 t1 f2 α2 t2*)
then obtain *p* **where** *f1*=*f2* ∧ (*bn α1*, *t1*) ≈*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) *p* (*bn α2*, *t2*) ∧ (*bn α1*, *α1*) ≈*set* (=) *supp p* (*bn α2*, *α2*)
by *auto*
then have *f1*=*f2* ∧ (*bn α2*, *t2*) ≈*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) (−*p*) (*bn α1*, *t1*) ∧ (*bn α2*, *α2*) ≈*set* (=) *supp* (−*p*) (*bn α1*, *α1*)
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 ∀x'. x' ∈ set-bset tset-x ⟶ (∃z'. z' ∈ set-bset tset-z ∧ x' =α 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.prems z)
              from 1 2 3 5 have x' =α z'
                by (rule tConj.IH)
              with 4 show ∃z'. z' ∈ set-bset tset-z ∧ x' =α z'
                by auto
            qed
          next
            show ∀z'. z' ∈ set-bset tset-z ⟶ (∃x'. x' ∈ set-bset tset-x ∧ x' =α 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.prems z)
              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 ∃x'. x' ∈ set-bset tset-x ∧ x' =α z'
                by auto
            qed
          qed
        with z show tConj tset-x =α z
          by simp
      qed (insert tConj.prems, 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 α t
      assume z: z = tAct f α t
       obtain p where 1: f1=f2 ∧ (bn α1, t1) ≈set (=α) (supp-rel (=α)) p (bn
α2, t2) ∧ (bn α1, α1) ≈set (=) supp p (bn α2, α2)
        using tAct.hyps by auto
      obtain q where 2: f2=f ∧ (bn α2, t2) ≈set (=α) (supp-rel (=α)) q (bn α,
t) ∧ (bn α2, α2) ≈set (=) supp q (bn α, α)
        using tAct.prems z by auto
      have f1=f ∧ (bn α1, t1) ≈set (=α) (supp-rel (=α)) (q + p) (bn α, t)
        proof −
          have supp-rel (=α) t1 − bn α1 = supp-rel (=α) t − bn α
            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)
          moreover have f1=f
            using 1 and 2 by simp
          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 f1 α1 t1 =α z
        using z by auto
    qed (insert tAct.prems, 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 =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: ⋀x y. rel-bset (=α) x y ⟷ rel-bset (=α) y x
    by (meson alpha-Tree-symp bset.rel-symp sympE)
  {
    fix a b
    from tConj.hyps have ∗: rel-bset (=α) ((a ⇌ b) · tset1) ((a ⇌ b) · tset2)
      by (metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj)
    have rel-bset (=α) ((a ⇌ b) · tset1) tset1 ⟷ rel-bset (=α) ((a ⇌ b) · 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 α1 t1 f2 α2 t2*)
{
fix *a b*
have *tAct f1 α1 t1* $=_\alpha$ *tAct f2 α2 t2*
using *tAct.hyps* **by** *simp*
then have (*a* ⇌ *b*) · *tAct f1 α1 t1* $=_\alpha$ *tAct f1 α1 t1* ⟷ (*a* ⇌ *b*) · *tAct f2 α2 t2* $=_\alpha$ *tAct f2 α2 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 α t1* $=_\alpha$ *tAct f α t2*
proof −
have (*bn α*, *t1*) ≈*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) *0* (*bn α*, *t2*)
using *assms* **by** (*simp add*: *alpha-Tree-supp-rel alpha-set fresh-star-zero*)
moreover have (*bn α*, *α*) ≈*set* (=) *supp 0* (*bn α*, *α*)
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 φ*) = *supp f* ∪ *supp φ*
unfolding *supp-rel-def supp-def* **by** (*simp add*: *Collect-imp-eq Collect-neg-eq*)

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* ⟹ (⋀*x*. *x* ∈ *S* ⟹ *x* ∈ *T*) ⟹ *infinite T*

by (*metis infinite-super subsetI*)

lemma *supp-rel-tAct* [*simp*]:
assumes *finite* (*supp-rel* ($=_\alpha$) *t*)
shows *supp-rel* ($=_\alpha$) (*tAct f* α *t*) = *supp f* $\cup$ (*supp* α $\cup$ *supp-rel* ($=_\alpha$) *t* $-$ *bn* α)
proof
show *supp f* $\cup$ (*supp* α $\cup$ *supp-rel* ($=_\alpha$) *t* $-$ *bn* α) $\subseteq$ *supp-rel* ($=_\alpha$) (*tAct f* α *t*)
proof
fix *x*
assume *x* $\in$ *supp f* $\cup$ (*supp* α $\cup$ *supp-rel* ($=_\alpha$) *t* $-$ *bn* α)
moreover
{
assume *x1*: *x* $\in$ *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*} $-$ *supp f*)
by (*simp add*: *finite-supp*)
moreover
{
fix *b*
assume *b* $\in$ {*b*. (*x* $\rightleftharpoons$ *b*) $\cdot$ *f* $\neq$ *f*} $-$ *supp f*
then have *b1*: (*x* $\rightleftharpoons$ *b*) $\cdot$ *f* $\neq$ *f* **and** *b2*: *b* $\notin$ *supp f*
by *simp*+
then have *sort-of x* = *sort-of b*
using *swap-different-sorts* **by** *fastforce*
then have (*x* $\rightleftharpoons$ *b*) $\cdot$ *supp f* $\neq$ *supp f*
using *b2 x1* **using** *swap-set-in* **by** *blast*
then have *b* $\in$ {*b*. $\neg$ (*x* $\rightleftharpoons$ *b*) $\cdot$ *tAct f* α *t* $=_\alpha$ *tAct f* α *t*}
by *auto*
}
ultimately have *infinite* {*b*. $\neg$ (*x* $\rightleftharpoons$ *b*) $\cdot$ *tAct f* α *t* $=_\alpha$ *tAct f* α *t*}
by (*rule infinite-mono*)
then have *x* $\in$ *supp-rel* ($=_\alpha$) (*tAct f* α *t*)
unfolding *supp-rel-def* **..**
}
moreover
{
assume *x1*: *x* $\in$ *supp* α **and** *x2*: *x* $\notin$ *bn* α
from *x1* **have** *infinite* {*b*. (*x* $\rightleftharpoons$ *b*) $\cdot$ α $\neq$ α}
unfolding *supp-def* **..**
then have *infinite* ({*b*. (*x* $\rightleftharpoons$ *b*) $\cdot$ α $\neq$ α} $-$ *supp* α)
by (*simp add*: *finite-supp*)
moreover
{
fix *b*
assume *b* $\in$ {*b*. (*x* $\rightleftharpoons$ *b*) $\cdot$ α $\neq$ α} $-$ *supp* α
then have *b1*: (*x* $\rightleftharpoons$ *b*) $\cdot$ α $\neq$ α **and** *b2*: *b* $\notin$ *supp* α $-$ *bn* α
by *simp*+

```
      from b1 have sort-of x = sort-of b
        using swap-different-sorts by fastforce
      then have (x ⇌ b) · (supp α − bn α) ≠ supp α − bn α
        using b2 x1 x2 by (simp add: swap-set-in)
      then have b ∈ {b. ¬ (x ⇌ b) · tAct f α t =α tAct f α t}
        by (auto simp add: alpha-set Diff-eqvt bn-eqvt)
    }
    ultimately have infinite {b. ¬ (x ⇌ b) · tAct f α t =α tAct f α t}
      by (rule infinite-mono)
    then have x ∈ supp-rel (=α) (tAct f α t)
      unfolding supp-rel-def ..
  }
  moreover
  {
    assume x1: x ∈ supp-rel (=α) t and x2: x ∉ bn α
    from x1 have infinite {b. ¬ (x ⇌ b) · t =α t}
      unfolding supp-rel-def ..
    then have infinite ({b. ¬ (x ⇌ b) · t =α t} − supp-rel (=α) t)
      using assms by simp
    moreover
    {
      fix b
      assume b ∈ {b. ¬ (x ⇌ b) · t =α t} − supp-rel (=α) t
      then have b1: ¬ (x ⇌ b) · t =α t and b2: b ∉ supp-rel (=α) t − bn α
        by simp+
      from b1 have (x ⇌ b) · t ≠ t
        by (metis alpha-Tree-reflp reflpE)
      then have sort-of x = sort-of b
        using swap-different-sorts by fastforce
      then have (x ⇌ b) · (supp-rel (=α) t − bn α) ≠ supp-rel (=α) t − bn α
        using b2 x1 x2 by (simp add: swap-set-in)
     then have supp-rel (=α) ((x ⇌ b) · t) − bn ((x ⇌ b) · α) ≠ supp-rel (=α)
t − bn α
        by (simp add: Diff-eqvt bn-eqvt)
      then have b ∈ {b. ¬ (x ⇌ b) · tAct f α t =α tAct f α t}
        by (simp add: alpha-set)
    }
    ultimately have infinite {b. ¬ (x ⇌ b) · tAct f α t =α tAct f α t}
      by (rule infinite-mono)
    then have x ∈ supp-rel (=α) (tAct f α t)
      unfolding supp-rel-def ..
  }
  ultimately show x ∈ supp-rel (=α) (tAct f α t)
    by auto
 qed
next
 show supp-rel (=α) (tAct f α t) ⊆ supp f ∪ (supp α ∪ supp-rel (=α) t − bn α)
 proof
   fix x
```

assume $x \in$ *supp-rel* $(=_\alpha)$ $(tAct\ f\ \alpha\ t)$
then have $*$: *infinite* $\{b.\ \neg\ (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_\alpha tAct\ f\ \alpha\ t\}$
unfolding *supp-rel-def* **..**
moreover
{
fix b
assume $\neg\ (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_\alpha tAct\ f\ \alpha\ t$
then have $(x \rightleftharpoons b) \cdot f \neq f \vee (x \rightleftharpoons b) \cdot \alpha \neq \alpha \vee \neg\ (x \rightleftharpoons b) \cdot t =_\alpha t$
using *alpha-Tree-tAct* **by** *force*
}
ultimately have *infinite* $\{b.\ (x \rightleftharpoons b) \cdot f \neq f \vee (x \rightleftharpoons b) \cdot \alpha \neq \alpha \vee \neg\ (x \rightleftharpoons b) \cdot t =_\alpha t\}$
using *infinite-mono mem-Collect-eq* **by** *force*
then have *infinite* $\{b.\ (x \rightleftharpoons b) \cdot f \neq f\} \vee$ *infinite* $\{b.\ (x \rightleftharpoons b) \cdot \alpha \neq \alpha\} \vee$ *infinite* $\{b.\ \neg\ (x \rightleftharpoons b) \cdot t =_\alpha t\}$
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*: $\neg\ (x \rightleftharpoons b) \cdot tAct\ f\ \alpha\ t =_\alpha tAct\ f\ \alpha\ t$ **and** *b1*: $b \notin$ *supp* f **and** *b2*: $b \notin$ *supp* α **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* $= (x \rightleftharpoons b)$
have *supp-rel* $(=_\alpha)$ $((x \rightleftharpoons b) \cdot t) - bn\ ((x \rightleftharpoons b) \cdot \alpha) =$ *supp-rel* $(=_\alpha)$ $t - bn\ \alpha$
using $**$ **and** *b3* **by** (*metis* (*no-types*, *lifting*) *Diff-eqvt Diff-iff alpha-Tree-eqvt' alpha-Tree-eqvt-aux bn-eqvt swap-set-not-in*)
moreover then have (*supp-rel* $(=_\alpha)$ $((x \rightleftharpoons b) \cdot t) - bn\ ((x \rightleftharpoons b) \cdot \alpha))\ \sharp*$ *?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* $\cdot\ bn\ ((x \rightleftharpoons b) \cdot \alpha) = bn\ \alpha$
by (*simp add*: *bn-eqvt*)
moreover have *supp* $((x \rightleftharpoons b) \cdot \alpha) - bn\ ((x \rightleftharpoons b) \cdot \alpha) =$ *supp* $\alpha - bn\ \alpha$
using $**$ **and** *b2* **by** (*metis* (*mono-tags*, *hide-lams*) *Diff-eqvt Diff-iff bn-eqvt supp-eqvt swap-set-not-in*)
moreover then have (*supp* $((x \rightleftharpoons b) \cdot \alpha) - bn\ ((x \rightleftharpoons b) \cdot \alpha))\ \sharp*$ *?p*
using $**$ **and** *b2* **by** (*simp add*: *fresh-star-def fresh-def supp-perm*) (*metis Diff-iff swap-atom-simps*(*3*))
moreover have *?p* $\cdot\ (x \rightleftharpoons b) \cdot \alpha = \alpha$
by *simp*
ultimately have $\exists p.\ (bn\ ((x \rightleftharpoons b) \cdot \alpha), (x \rightleftharpoons b) \cdot t) \approx set\ (=_\alpha)$ *supp-rel* $(=_\alpha)\ p\ (bn\ \alpha, t) \wedge$
$(bn\ ((x \rightleftharpoons b) \cdot \alpha), (x \rightleftharpoons b) \cdot \alpha) \approx set\ (=)$ *supp* $p\ (bn\ \alpha, \alpha)$
by (*auto simp add*: *alpha-set.simps*)

```
      moreover have (x ⇌ b) · f = f using ∗∗ and b1
        by (simp add: fresh-def swap-fresh-fresh)
      ultimately have (x ⇌ b) · tAct f α t =α tAct f α t
       by simp
      with b0 have False ..
    }
    ultimately show x ∈ supp f ∪ (supp α ∪ supp-rel (=α) t − bn α)
     by blast
  qed
qed
```

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

```
quotient-type
  ('idx,'pred,'act,'eff) Tree_α = ('idx,'pred::pt,'act::bn,'eff::fs) Tree / alpha-Tree
  by (fact alpha-Tree-equivp)

lemma Tree_α-abs-rep [simp]: abs-Tree_α (rep-Tree_α t_α) = t_α
by (metis Quotient-Tree_α Quotient-abs-rep)

lemma Tree_α-rep-abs [simp]: rep-Tree_α (abs-Tree_α t) =α t
by (metis Tree_α.abs-eq-iff Tree_α-abs-rep)
```

The permutation operation is lifted from trees.

```
instantiation Tree_α :: (type, pt, bn, fs) pt
begin

  lift-definition permute-Tree_α :: perm ⇒ ('a,'b,'c,'d) Tree_α ⇒ ('a,'b,'c,'d) Tree_α
    is permute
  by (fact alpha-Tree-eqvt)

  instance
  proof
    fix t_α :: (-,-,-,-) Tree_α
    show 0 · t_α = t_α
      by transfer (metis alpha-Tree-equivp equivp-reflp permute-zero)
  next
    fix p q :: perm and t_α :: (-,-,-,-) Tree_α
    show (p + q) · t_α = p · q · t_α
      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_α.abs-eq [eqvt, simp]

lemma alpha-Tree-permute-rep-commute [simp]: p · rep-Tree_α t_α =α rep-Tree_α (p
· t_α)
```

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

12.3 Constructors for trees modulo α-equivalence

The constructors are lifted from trees.

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

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

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

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

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

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

The lifted constructors are equivariant.

lemma $Conj_\alpha$-*eqvt* [*eqvt*, *simp*]: $p \cdot Conj_\alpha\ tset_\alpha = Conj_\alpha\ (p \cdot tset_\alpha)$
proof −
{
fix x
assume $x \in$ *set-bset* ($p \cdot$ *map-bset rep-*$Tree_\alpha$ $tset_\alpha$)
then obtain y **where** $y \in$ *set-bset* (*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*: $t_\alpha \in$ *set-bset* $tset_\alpha$ **and** *2*: $x = p \cdot$ *rep-*$Tree_\alpha$ t_α
by (*metis imageE map-bset.rep-eq*)
let *?x'* = *rep-*$Tree_\alpha$ ($p \cdot t_\alpha$)
from *1* **have** $p \cdot t_\alpha \in$ *set-bset* ($p \cdot tset_\alpha$)
by (*metis mem-permute-iff permute-bset.rep-eq*)
then have *?x'* ∈ *set-bset* (*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 ∃ *x'*∈*set-bset* (*map-bset rep-*$Tree_\alpha$ ($p \cdot tset_\alpha$)). $x =_\alpha$ *x'*

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

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

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

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

The lifted constructors are injective (except for Act_α).

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

lemma *Not*$_\alpha$*-eq-iff* [*simp*]: *Not*$_\alpha$ *t1*$_\alpha$ $=$ *Not*$_\alpha$ *t2*$_\alpha$ $\longleftrightarrow$ *t1*$_\alpha$ $=$ *t2*$_\alpha$
proof

assume $Not_\alpha\ t1_\alpha = Not_\alpha\ t2_\alpha$
then have $tNot\ (rep\text{-}Tree_\alpha\ t1_\alpha) =_\alpha tNot\ (rep\text{-}Tree_\alpha\ t2_\alpha)$
by (*metis* $Not_\alpha.abs\text{-}eq\ Tree_\alpha.abs\text{-}eq\text{-}iff\ Tree_\alpha\text{-}abs\text{-}rep$)
then have $rep\text{-}Tree_\alpha\ t1_\alpha =_\alpha rep\text{-}Tree_\alpha\ t2_\alpha$
using *alpha-Tree.cases* **by** *auto*
then show $t1_\alpha = t2_\alpha$
by (*metis* $Tree_\alpha.abs\text{-}eq\text{-}iff\ Tree_\alpha\text{-}abs\text{-}rep$)
next
assume $t1_\alpha = t2_\alpha$ **then show** $Not_\alpha\ t1_\alpha = Not_\alpha\ t2_\alpha$
by *simp*
qed

lemma $Pred_\alpha\text{-}eq\text{-}iff$ [*simp*]: $Pred_\alpha\ f1\ \varphi1 = Pred_\alpha\ f2\ \varphi2 \longleftrightarrow f1 = f2 \wedge \varphi1 = \varphi2$
proof
assume $Pred_\alpha\ f1\ \varphi1 = Pred_\alpha\ f2\ \varphi2$
then have ($tPred\ f1\ \varphi1 :: ('e, 'b, 'f, 'd)\ Tree) =_\alpha tPred\ f2\ \varphi2$ — note the unrelated type
by (*metis* $Pred_\alpha.abs\text{-}eq\ Tree_\alpha.abs\text{-}eq\text{-}iff$)
then show $f1 = f2 \wedge \varphi1 = \varphi2$
using *alpha-Tree.cases* **by** *auto*
next
assume $f1 = f2 \wedge \varphi1 = \varphi2$ **then show** $Pred_\alpha\ f1\ \varphi1 = Pred_\alpha\ f2\ \varphi2$
by *simp*
qed

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

The lifted constructors are free (except for Act_α).

lemma $Tree_\alpha\text{-}free$ [*simp*]:
shows $Conj_\alpha\ tset_\alpha \neq Not_\alpha\ t_\alpha$
and $Conj_\alpha\ tset_\alpha \neq Pred_\alpha\ f\ \varphi$
and $Conj_\alpha\ tset_\alpha \neq Act_\alpha\ f\ \alpha\ t_\alpha$
and $Not_\alpha\ t_\alpha \neq Pred_\alpha\ f\ \varphi$
and $Not_\alpha\ t1_\alpha \neq Act_\alpha\ f\ \alpha\ t2_\alpha$
and $Pred_\alpha\ f1\ \varphi \neq Act_\alpha\ f2\ \alpha\ t_\alpha$
by (*simp add*: $Conj_\alpha\text{-}def'\ Not_\alpha\text{-}def\ Pred_\alpha\text{-}def\ Act_\alpha\text{-}def\ Tree_\alpha.abs\text{-}eq\text{-}iff$)+

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

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

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

lemma $supp\text{-}Not_\alpha$ [*simp*]: $supp\ (Not_\alpha\ t_\alpha) = supp\ t_\alpha$

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

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

lemma *supp-Act*$_\alpha$ [*simp*]:
assumes *finite* (*supp* t_α)
shows *supp* (*Act*$_\alpha$ *f* α t_α) = *supp f* $\cup$ (*supp* α $\cup$ *supp* t_α $-$ *bn* α)
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*)

12.4 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* $\in$ *set-bset tset*$_\alpha$ $\Longrightarrow$ *P x*) $\Longrightarrow$ *P* (*Conj*$_\alpha$ *tset*$_\alpha$)
and $\bigwedge t_\alpha$. *P* t_α $\Longrightarrow$ *P* (*Not*$_\alpha$ t_α)
and $\bigwedge$*f pred*. *P* (*Pred*$_\alpha$ *f pred*)
and $\bigwedge$*f act* t_α. *P* t_α $\Longrightarrow$ *P* (*Act*$_\alpha$ *f act* t_α)
shows *P* t_α
proof (*rule Tree*$_\alpha$*.abs-induct*)
fix *t* **show** *P* (*abs-Tree*$_\alpha$ *t*)
proof (*induction t*)
case (*tConj tset*)
let *?tset*$_\alpha$ = *map-bset abs-Tree*$_\alpha$ *tset*
have *abs-Tree*$_\alpha$ (*tConj tset*) = *Conj*$_\alpha$ *?tset*$_\alpha$
by (*simp add*: *Conj*$_\alpha$*.abs-eq*)
then show *?case*
using *assms*(*1*) *tConj.IH* **by** (*metis imageE map-bset.rep-eq*)
next
case *tNot* **then show** *?case*
using *assms*(*2*) **by** (*metis 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.

12.5 Hereditarily finitely supported trees

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

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

inductive *hereditarily-fs* :: ('*idx*,'*pred*::*fs*,'*act*::*bn*,'*eff*::*fs*) $Tree_\alpha \Rightarrow$ *bool* **where**
$Conj_\alpha$: *finite* (*supp* $tset_\alpha$) $\Longrightarrow$ ($\bigwedge t_\alpha.\ t_\alpha \in$ *set-bset* $tset_\alpha \Longrightarrow$ *hereditarily-fs* t_α) $\Longrightarrow$ *hereditarily-fs* ($Conj_\alpha$ $tset_\alpha$)
| Not_α: *hereditarily-fs* $t_\alpha \Longrightarrow$ *hereditarily-fs* (Not_α t_α)
| $Pred_\alpha$: *hereditarily-fs* ($Pred_\alpha$ f φ)
| Act_α: *hereditarily-fs* $t_\alpha \Longrightarrow$ *hereditarily-fs* (Act_α f α t_α)

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*, *hide-lams*) $Conj_\alpha$*-eqvt hereditarily-fs.*$Conj_\alpha$ *mem-permute-iff permute-finite permute-minus-cancel*(*1*) *set-bset-eqvt supp-eqvt*)
next
case Not_α **then show** *?case*
by (*metis* Not_α*-eqvt hereditarily-fs.*Not_α)
next
case $Pred_\alpha$ **then show** *?case*
by (*metis* $Pred_\alpha$*-eqvt hereditarily-fs.*$Pred_\alpha$)
next
case Act_α **then show** *?case*
by (*metis* Act_α*-eqvt hereditarily-fs.*Act_α)
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_α
then show *hereditarily-fs* $t_\alpha{}'$
using *assms* **by** (*auto simp add*: Act_α*-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_α

using *assms* **by** (*auto simp add*: Act_α*-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_α
shows *finite* (*supp* t_α)
using *assms* **by** (*induction rule*: *hereditarily-fs.induct*) (*simp-all add*: *finite-supp*)

12.6 Infinitary formulas

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

typedef (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* = {t_α::(*'idx*,*'pred*,*'act*,*'eff*) $Tree_\alpha$. *hereditarily-fs* t_α}
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_α
shows *Rep-formula* (*Abs-formula* t_α) = t_α
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* ⇒ (*'a*,*'b*,*'c*,*'d*) *formula* ⇒ (*'a*,*'b*,*'c*,*'d*) *formula*
is *permute*
by (*fact hereditarily-fs-eqvt*)

instance
by *standard* (*transfer*, *simp*)+

end

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

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

assumes *hereditarily-fs* t_α
shows $p \cdot$ *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_α
shows *supp* (*Abs-formula* t_α) = *supp* t_α
proof −
{
fix p :: *perm*
have $p \cdot$ *Abs-formula* $t_\alpha =$ *Abs-formula* $(p \cdot t_\alpha)$
using *assms* **by** (*metis Abs-formula-eqvt*)
moreover have *hereditarily-fs* $(p \cdot t_\alpha)$
using *assms* **by** (*metis hereditarily-fs-eqvt*)
ultimately have $p \cdot$ *Abs-formula* $t_\alpha =$ *Abs-formula* $t_\alpha \longleftrightarrow p \cdot t_\alpha = t_\alpha$
using *assms* **by** (*metis Abs-formula-inverse*)
}
then show *?thesis* **unfolding** *supp-def* **by** *simp*
qed

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

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

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

Formulas are in fact finitely supported.

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

12.7 Constructors for infinitary formulas

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

definition *Conj* :: ('*idx*,'*pred*,'*act*,'*eff*) *formula set*['*idx*] ⇒ ('*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_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*,'*eff*) *formula* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*,'*eff*::*fs*) *formula* **is**
  Not$_\alpha$
by (*fact hereditarily-fs.Not*$_\alpha$)

lift-definition *Pred* :: '*eff* ⇒ '*pred* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*,'*eff*::*fs*) *formula* **is**
  Pred$_\alpha$
by (*fact hereditarily-fs.Pred*$_\alpha$)

lift-definition *Act* :: '*eff* ⇒ '*act* ⇒ ('*idx*,'*pred*,'*act*,'*eff*) *formula* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*,'*eff*::*fs*) *formula* **is**
  Act$_\alpha$
by (*fact hereditarily-fs.Act*$_\alpha$)

The lifted constructors are equivariant (in the case of *Conj*, on finitely sup-

ported arguments).

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

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

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

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

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
  assumes *finite* (*supp xset*)
  shows *supp* (*Conj xset*) = *supp xset*
using *assms* **unfolding** *Conj-def* **by** *simp*

lemma *supp-Not* [*simp*]: *supp* (*Not x*) = *supp x*
by (*metis Not.rep-eq supp-Not*$_\alpha$ *supp-Rep-formula*)

lemma *supp-Pred* [*simp*]: *supp* (*Pred f* φ) = *supp f* $\cup$ *supp* φ
by (*metis Pred.rep-eq supp-Pred*$_\alpha$ *supp-Rep-formula*)

lemma *supp-Act* [*simp*]: *supp* (*Act f* α *x*) = *supp f* $\cup$ (*supp* α $\cup$ *supp x* $-$ *bn* α)
by (*metis Act.rep-eq finite-supp supp-Act*$_\alpha$ *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 *Conj xset1* = *Conj xset2* $\longleftrightarrow$ *xset1* = *xset2*
using *assms*
by (*metis* (*erased*, *hide-lams*) *Conj*$_\alpha$*-eq-iff Conj-rep-eq Rep-formula-inverse injI inj-eq bset.inj-map*)

lemma *Not-eq-iff* [*simp*]: *Not x1* = *Not x2* $\longleftrightarrow$ *x1* = *x2*
by (*metis Not.rep-eq Not*$_\alpha$*-eq-iff Rep-formula-inverse*)

lemma *Pred-eq-iff* [*simp*]: *Pred f1* φ*1* = *Pred f2* φ*2* $\longleftrightarrow$ *f1* = *f2* $\wedge$ φ*1* = φ*2*
by (*metis Pred.rep-eq Pred*$_\alpha$*-eq-iff*)

lemma *Act-eq-iff*: *Act f1* α*1 x1* = *Act f2* α*2 x2* $\longleftrightarrow$ *Act*$_\alpha$ *f1* α*1* (*Rep-formula x1*) = *Act*$_\alpha$ *f2* α*2* (*Rep-formula x2*)
by (*metis Act.rep-eq Rep-formula-inverse*)

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: *Act f1 α1 x1 = Act f2 α2 x2 ⟷*
f1 = f2 ∧ (∃ p. supp x1 − bn α1 = supp x2 − bn α2 ∧ (supp x1 − bn α1) ♯∗ p ∧ p ∙ x1 = x2 ∧ supp α1 − bn α1 = supp α2 − bn α2 ∧ (supp α1 − bn α1) ♯∗ p ∧ p ∙ α1 = α2)
(**is** *?l ⟷ ?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 α1*, *rep-Tree$_\alpha$* (*Rep-formula x1*)) ≈*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) *p* (*bn α2*, *rep-Tree$_\alpha$* (*Rep-formula x2*)) **and** *eq*: (*bn α1*, *α1*) ≈*set* (=) *supp p* (*bn α2*, *α2*)
by (*metis Act-eq-iff Act$_\alpha$-eq-iff alpha-tAct*)
from *alpha* **have** *supp x1 − bn α1 = supp x2 − bn α2*
by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)
moreover from *alpha* **have** (*supp x1 − bn α1*) ♯∗ *p*
by (*metis alpha-set.simps supp-Rep-formula supp-alpha-supp-rel*)
moreover from *alpha* **have** *p ∙ 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 α1 − bn α1 = supp α2 − bn α2*
by (*metis alpha-set.simps*)
moreover from *eq* **have** (*supp α1 − bn α1*) ♯∗ *p*
by (*metis alpha-set.simps*)
moreover from *eq* **have** *p ∙ α1 = α2*
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 α1 = supp x2 − bn α2* **and** *2*: (*supp x1 − bn α1*) ♯∗ *p* **and** *3*: *p ∙ x1 = x2*
and *4*: *supp α1 − bn α1 = supp α2 − bn α2* **and** *5*: (*supp α1 − bn α1*) ♯∗ *p* **and** *6*: *p ∙ α1 = α2*
by *metis*
from *1 2 3 6* **have** (*bn α1*, *rep-Tree$_\alpha$* (*Rep-formula x1*)) ≈*set* (=$_\alpha$) (*supp-rel* (=$_\alpha$)) *p* (*bn α2*, *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 α1*, *α1*) ≈*set* (=) *supp p* (*bn α2*, *α2*)
by (*simp add*: *alpha-set.simps bn-eqvt*)
ultimately show *Act f1 α1 x1 = Act f2 α2 x2*
by (*metis Act-eq-iff Act$_\alpha$-eq-iff alpha-tAct*)
qed

lemma *Act-eq-iff-perm-renaming*: *Act f1 α1 x1 = Act f2 α2 x2 ⟷*
f1 = f2 ∧ (∃ p. supp x1 − bn α1 = supp x2 − bn α2 ∧ (supp x1 − bn α1) ♯∗

$p \wedge p \cdot x1 = x2 \wedge supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2 \wedge (supp\ \alpha1 - bn\ \alpha1) \sharp* p \wedge p \cdot \alpha1 = \alpha2 \wedge supp\ p \subseteq bn\ \alpha1 \cup p \cdot bn\ \alpha1$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *?l* **then have** *f1 = f2*
by (*metis Act-eq-iff-perm*)
moreover from ‹*?l*› **obtain** *p* **where** *p*: $supp\ x1 - bn\ \alpha1 = supp\ x2 - bn\ \alpha2 \wedge (supp\ x1 - bn\ \alpha1) \sharp* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha1 - bn\ \alpha1 = supp\ \alpha2 - bn\ \alpha2 \wedge (supp\ \alpha1 - bn\ \alpha1) \sharp* p \wedge p \cdot \alpha1 = \alpha2$
by (*metis Act-eq-iff-perm*)
moreover obtain *q* **where** *q-p*: $\forall b \in bn\ \alpha1.\ q \cdot b = p \cdot b$ **and** *supp-q*: $supp\ q \subseteq bn\ \alpha1 \cup p \cdot bn\ \alpha1$
by (*metis set-renaming-perm2*)
have $supp\ q \subseteq supp\ p$
proof
fix *a* **assume** ∗: $a \in supp\ q$ **then show** $a \in supp\ p$
proof (*cases* $a \in bn\ \alpha1$)
case *True* **then show** *?thesis*
using ∗ *q-p* **by** (*metis mem-Collect-eq supp-perm*)
next
case *False* **then have** $a \in p \cdot bn\ \alpha1$
using ∗ *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** $(supp\ x1 - bn\ \alpha1) \sharp* q$ **and** $(supp\ \alpha1 - bn\ \alpha1) \sharp* q$
by (*meson fresh-def fresh-star-def subset-iff*)+
moreover with *p* **and** *q-p* **have** $\bigwedge a.\ a \in supp\ \alpha1 \Longrightarrow q \cdot a = p \cdot a$ **and** $\bigwedge a.\ a \in supp\ x1 \Longrightarrow q \cdot a = p \cdot a$
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 *supp-q* **by** (*metis bn-eqvt*)
next
assume *?r* **then show** *?l*
by (*meson Act-eq-iff-perm*)
qed

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

lemma *Tree-free* [*simp*]:
shows $finite\ (supp\ xset) \Longrightarrow Conj\ xset \neq Not\ x$
and $finite\ (supp\ xset) \Longrightarrow Conj\ xset \neq Pred\ f\ \varphi$
and $finite\ (supp\ xset) \Longrightarrow Conj\ xset \neq Act\ f\ \alpha\ x$
and $Not\ x \neq Pred\ f\ \varphi$
and $Not\ x1 \neq Act\ f\ \alpha\ x2$

and *Pred f1* $\varphi \neq$ *Act f2* α *x*
proof −
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Not x*
by (*metis Conj-rep-eq Not.rep-eq Tree*$_\alpha$*-free*(*1*))
next
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Pred f* φ
by (*metis Conj-rep-eq Pred.rep-eq Tree*$_\alpha$*-free*(*2*))
next
show *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Act f* α *x*
by (*metis Conj-rep-eq Act.rep-eq Tree*$_\alpha$*-free*(*3*))
next
show *Not x* $\neq$ *Pred f* φ
by (*metis Not.rep-eq Pred.rep-eq Tree*$_\alpha$*-free*(*4*))
next
show *Not x1* $\neq$ *Act f* α *x2*
by (*metis Not.rep-eq Act.rep-eq Tree*$_\alpha$*-free*(*5*))
next
show *Pred f1* $\varphi \neq$ *Act f2* α *x*
by (*metis Pred.rep-eq Act.rep-eq Tree*$_\alpha$*-free*(*6*))
qed

12.8 F/L-formulas

context *effect-nominal-ts*
begin

The predicate *is-FL-formula* will characterise exactly those formulas in a particular set $A^{F/L}$.

inductive *is-FL-formula* :: *'effect first* ⇒ (*'idx*,*'pred*,*'act*,*'effect*) *formula* ⇒ *bool*
where
Conj: *finite* (*supp xset*) $\Longrightarrow$ ($\bigwedge$*x*. *x* ∈ *set-bset xset* $\Longrightarrow$ *is-FL-formula F x*) $\Longrightarrow$ *is-FL-formula F* (*Conj xset*)
| *Not*: *is-FL-formula F x* $\Longrightarrow$ *is-FL-formula F* (*Not x*)
| *Pred*: *f* $\in_{fs}$ *F* $\Longrightarrow$ *is-FL-formula F* (*Pred f* φ)
| *Act*: *f* $\in_{fs}$ *F* $\Longrightarrow$ *bn* α ♯* (*F*,*f*) $\Longrightarrow$ *is-FL-formula* (*L* (α,*F*,*f*)) *x* $\Longrightarrow$ *is-FL-formula F* (*Act f* α *x*)

abbreviation *in-*$\mathcal{A}$ :: (*'idx*,*'pred*,*'act*,*'effect*) *formula* ⇒ *'effect first* ⇒ *bool*
(- ∈ $\mathcal{A}$[-] [*51*,*0*] *50*) **where**
x ∈ $\mathcal{A}$[*F*] ≡ *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* ∈ $\mathcal{A}$[*F*] $\Longrightarrow$ *p* ∙ *x* ∈ $\mathcal{A}$[*p* ∙ *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* ∈ *set-bset xset* $\Longrightarrow$ *p* ∙ *x* ∈ $\mathcal{A}$[*p* ∙ *F*]
from *1* **have** *finite* (*supp* (*p* ∙ *xset*))

by (*metis permute-finite supp-eqvt*)
moreover from *2* **have** $\bigwedge x.\ x \in \mathit{set\text{-}bset}\ (p \cdot \mathit{xset}) \Longrightarrow x \in \mathcal{A}[p \cdot F]$
by (*metis* (*mono-tags*) *imageE permute-set-eq-image set-bset-eqvt*)
ultimately show $p \cdot \mathit{Conj}\ \mathit{xset} \in \mathcal{A}[p \cdot F]$
using *1* **by** (*simp add*: *Conj*)
next
fix F **and** $x :: ('a, 'pred, 'act, 'effect)\ formula$
assume $p \cdot x \in \mathcal{A}[p \cdot F]$
then show $p \cdot \mathit{Not}\ x \in \mathcal{A}[p \cdot F]$
by (*simp add*: *Not*)
next
fix f **and** $F :: 'effect\ first$ **and** φ
assume $f \in_{fs} F$
then show $p \cdot \mathit{Pred}\ f\ \varphi \in \mathcal{A}[p \cdot F]$
by (*simp add*: *Pred*)
next
fix $f\ F\ \alpha$ **and** $x :: ('a, 'pred, 'act, 'effect)\ formula$
assume $f \in_{fs} F$ **and** $bn\ \alpha\ \sharp*\ (F,f)$ **and** $p \cdot x \in \mathcal{A}[p \cdot L\ (\alpha, F, f)]$
then show $p \cdot \mathit{Act}\ f\ \alpha\ x \in \mathcal{A}[p \cdot F]$
by (*metis* (*mono-tags*, *lifting*) *Act Act-eqvt L-eqvt' Pair-eqvt bn-eqvt fresh-star-permute-iff member-fs-set-permute-iff*)
qed

end

12.9 Induction over infinitary formulas

12.10 Strong induction over infinitary formulas

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

13 Validity With Effects

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

definition *alpha-Tree-rel* **where**
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** $x =_\alpha y \Longrightarrow P$

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-eqvt' wf-Tree-wf wf-hull-rel-relcomp*)
next
show (*hull-rel O Tree-wf*) *O alpha-Tree-rel* ⊆ *alpha-Tree-rel O* (*hull-rel O Tree-wf*)
proof
fix *x* :: ('*e*, '*f*, '*g*, '*h*) *Tree* × ('*e*, '*f*, '*g*, '*h*) *Tree*
assume *x* ∈ (*hull-rel O Tree-wf*) *O alpha-Tree-rel*
then obtain *x1 x2 x3 x4* **where** *x*: *x* = (*x1*,*x4*) **and** *1*: (*x1*,*x2*) ∈ *hull-rel* **and** *2*: (*x2*,*x3*) ∈ *Tree-wf* **and** *3*: (*x3*,*x4*) ∈ *alpha-Tree-rel*
by *auto*
from *2* **have** (*x1*,*x4*) ∈ *alpha-Tree-rel O hull-rel O Tree-wf*
using *1* **and** *3* **proof** (*induct rule*: *Tree-wf.induct*)
— *tConj*
fix *t* **and** *tset* :: ('*e*,'*f*,'*g*,'*h*) *Tree set*['*e*]
assume ∗: *t* ∈ *set-bset tset* **and** ∗∗: (*x1*,*t*) ∈ *hull-rel* **and** ∗∗∗: (*tConj tset*, *x4*) ∈ *alpha-Tree-rel*
from ∗∗ **obtain** *p* **where** *x1*: *x1* = *p* ∙ *t*
using *hull-rel.cases* **by** *blast*
from ∗∗∗ **have** *tConj tset* =$_\alpha$ *x4*
by (*rule alpha-Tree-relE*)
then obtain *tset'* **where** *x4*: *x4* = *tConj tset'* **and** *rel-bset* (=$_\alpha$) *tset tset'*
by (*cases x4*) *simp-all*
with ∗ **obtain** *t'* **where** *t'*: *t'* ∈ *set-bset tset'* **and** *t* =$_\alpha$ *t'*
by (*metis rel-bset.rep-eq rel-set-def*)
with *x1* **have** (*x1*, *p* ∙ *t'*) ∈ *alpha-Tree-rel*
by (*metis Tree$_\alpha$.abs-eq-iff alpha-Tree-relI permute-Tree$_\alpha$.abs-eq*)
moreover have (*p* ∙ *t'*, *t'*) ∈ *hull-rel*
by (*rule hull-rel.intros*)
moreover from *x4* **and** *t'* **have** (*t'*, *x4*) ∈ *Tree-wf*
by (*simp add*: *Tree-wf.intros*(*1*))
ultimately show (*x1*,*x4*) ∈ *alpha-Tree-rel O hull-rel O Tree-wf*
by *auto*
next
— *tNot*
fix *t*
assume ∗: (*x1*,*t*) ∈ *hull-rel* **and** ∗∗: (*tNot t*, *x4*) ∈ *alpha-Tree-rel*
from ∗ **obtain** *p* **where** *x1*: *x1* = *p* ∙ *t*
using *hull-rel.cases* **by** *blast*
from ∗∗ **have** *tNot t* =$_\alpha$ *x4*
by (*rule alpha-Tree-relE*)
then obtain *t'* **where** *x4*: *x4* = *tNot t'* **and** *t* =$_\alpha$ *t'*
by (*cases x4*) *simp-all*
with *x1* **have** (*x1*, *p* ∙ *t'*) ∈ *alpha-Tree-rel*

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

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

lemma alpha-Tree-rel-relcompI [simp]:
 assumes x =_α x' and (x', y) ∈ R
 shows (x, y) ∈ alpha-Tree-rel O R
using assms unfolding alpha-Tree-rel-def
by (metis case-prodI mem-Collect-eq relcomp.relcompI)
```

13.1 Validity for infinitely branching trees

```
context effect-nominal-ts
begin
```

Since we defined formulas via a manual quotient construction, we also need to define validity via lifting from the underlying type of infinitely branching

trees. We cannot use **nominal_function** because that generates proof obligations where, for formulas of the form *Conj xset*, the assumption that *xset* has finite support is missing.

```
declare conj-cong [fundef-cong]

function (sequential) FL-valid-Tree :: 'state ⇒ ('idx,'pred,'act,'effect) Tree ⇒
bool where
  FL-valid-Tree P (tConj tset) ⟷ (∀ t∈set-bset tset. FL-valid-Tree P t)
| FL-valid-Tree P (tNot t) ⟷ ¬ FL-valid-Tree P t
| FL-valid-Tree P (tPred f φ) ⟷ ⟨f⟩P ⊢ φ
| FL-valid-Tree P (tAct f α t) ⟷ (∃ α' t' P'. tAct f α t =α tAct f α' t' ∧ ⟨f⟩P
→ ⟨α',P'⟩ ∧ 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 ∈ set-bset tset then show ((P, t), (P, tConj tset)) ∈ ?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)) ∈ ?R
      by (simp add: Tree-wf.intros(2))
  next
   fix P1 P2 :: 'state and f and α1 α2 and t1 t2 :: ('idx,'pred,'act,'effect) Tree
    assume tAct f α1 t1 =α tAct f α2 t2
    then obtain p where t2 =α p · t1
      by (auto simp add: alphas) (metis alpha-Tree-symp sympE)
    then show ((P2, t2), (P1, tAct f α1 t1)) ∈ ?R
      by (simp add: Tree-wf.intros(3))
  }
qed
```

FL-valid-Tree is equivariant.

```
lemma FL-valid-Tree-eqvt': FL-valid-Tree P t ⟷ FL-valid-Tree (p · P) (p · 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 ∈ p · set-bset tset
        with 1.IH and ∗ have FL-valid-Tree (p · P) t
       by (metis (no-types, lifting) imageE permute-set-eq-image FL-valid-Tree.simps(1))
      }
      then show FL-valid-Tree (p · P) (p · 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-eqvt FL-valid-Tree.simps(1))
        }
        then show FL-valid-Tree P (tConj tset)
          by simp
      qed
    next
      case 2 then show ?case by simp
    next
      case 3 show ?case by simp (metis effect-apply-eqvt' permute-minus-cancel(2)
satisfies-eqvt)
    next
      case (4 P f α t) show ?case
        proof
          assume FL-valid-Tree P (tAct f α t)
         then obtain α' t' P' where ∗: tAct f α t =α tAct f α' t' ∧ ⟨f⟩P → ⟨α',P'⟩
∧ FL-valid-Tree P' t'
          by auto
        with 4.IH have FL-valid-Tree (p · P') (p · t')
          by blast
        moreover from ∗ have p · ⟨f⟩P → ⟨p · α', p · P'⟩
          by (metis transition-eqvt')
        moreover from ∗ have p · tAct f α t =α tAct (p · f) (p · α') (p · t')
          by (metis alpha-Tree-eqvt permute-Tree.simps(4))
        ultimately show FL-valid-Tree (p · P) (p · tAct f α t)
          by auto
      next
        assume FL-valid-Tree (p · P) (p · tAct f α t)
        then obtain α' t' P' where ∗: p · tAct f α t =α tAct (p · f) α' t' ∧ (p ·
⟨f⟩P) → ⟨α',P'⟩ ∧ FL-valid-Tree P' t'
          by auto
        then have eq: tAct f α t =α tAct f (−p · α') (−p · t')
        by (metis alpha-Tree-eqvt permute-Tree.simps(4) permute-minus-cancel(2))
        moreover from ∗ have ⟨f⟩P → ⟨−p · α', −p · P'⟩
          by (metis permute-minus-cancel(2) transition-eqvt')
        moreover with 4.IH have FL-valid-Tree (−p · P') (−p · t')
          using eq and ∗ by simp
        ultimately show FL-valid-Tree P (tAct f α t)
          by auto
      qed
  qed

  lemma FL-valid-Tree-eqvt [eqvt]:
```

assumes *FL-valid-Tree P t* **shows** *FL-valid-Tree* $(p \cdot P)$ $(p \cdot t)$
using *assms* **by** (*metis FL-valid-Tree-eqvt'*)

α-equivalent trees validate the same states.

lemma *alpha-Tree-FL-valid-Tree*:
assumes *t1* $=_\alpha$ *t2*
shows *FL-valid-Tree P t1* $\longleftrightarrow$ *FL-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* $\alpha 1$ *t1 f2* $\alpha 2$ *t2*) **show** *?case*
proof
assume *FL-valid-Tree P* (*tAct f1* $\alpha 1$ *t1*)
then obtain α' t' P' **where** *tAct f1* $\alpha 1$ *t1* $=_\alpha$ *tAct f1* α' t' $\wedge$ $\langle f1 \rangle P \rightarrow \langle \alpha', P' \rangle \wedge$ *FL-valid-Tree* P' t'
by *auto*
moreover from *tAct.hyps* **have** *tAct f1* $\alpha 1$ *t1* $=_\alpha$ *tAct f2* $\alpha 2$ *t2*
using *alpha-tAct* **by** *blast*
ultimately show *FL-valid-Tree P* (*tAct f2* $\alpha 2$ *t2*)
using *tAct.hyps* **by** (*metis* $Tree_\alpha$*.abs-eq-iff FL-valid-Tree.simps*(4))
next
assume *FL-valid-Tree P* (*tAct f2* $\alpha 2$ *t2*)
then obtain α' t' P' **where** *tAct f2* $\alpha 2$ *t2* $=_\alpha$ *tAct f2* α' t' $\wedge$ $\langle f2 \rangle P \rightarrow \langle \alpha', P' \rangle \wedge$ *FL-valid-Tree* P' t'
by *auto*
moreover from *tAct.hyps* **have** *tAct f1* $\alpha 1$ *t1* $=_\alpha$ *tAct f2* $\alpha 2$ *t2*
using *alpha-tAct* **by** *blast*
ultimately show *FL-valid-Tree P* (*tAct f1* $\alpha 1$ *t1*)
using *tAct.hyps* **by** (*metis* $Tree_\alpha$*.abs-eq-iff FL-valid-Tree.simps*(4))
qed
qed *simp-all*

13.2 Validity for trees modulo α-equivalence

lift-definition $FL\text{-}valid\text{-}Tree_\alpha$ :: *'state* $\Rightarrow$ (*'idx*,*'pred*,*'act*,*'effect*) $Tree_\alpha$ $\Rightarrow$ *bool*
is
FL-valid-Tree
by (*fact alpha-Tree-FL-valid-Tree*)

lemma $FL\text{-}valid\text{-}Tree_\alpha$*-eqvt* [*eqvt*]:
assumes $FL\text{-}valid\text{-}Tree_\alpha$ *P t* **shows** $FL\text{-}valid\text{-}Tree_\alpha$ $(p \cdot P)$ $(p \cdot t)$
using *assms* **by** *transfer* (*fact FL-valid-Tree-eqvt*)

lemma $FL\text{-}valid\text{-}Tree_\alpha$*-*$Conj_\alpha$ [*simp*]: $FL\text{-}valid\text{-}Tree_\alpha$ *P* ($Conj_\alpha$ $tset_\alpha$) $\longleftrightarrow$ ($\forall\, t_\alpha \in$*set-bset* $tset_\alpha$. $FL\text{-}valid\text{-}Tree_\alpha$ *P* t_α)
proof −
have *FL-valid-Tree P* (*rep-*$Tree_\alpha$ (*abs-*$Tree_\alpha$ (*tConj* (*map-bset rep-*$Tree_\alpha$ $tset_\alpha$)))) $\longleftrightarrow$ *FL-valid-Tree P* (*tConj* (*map-bset rep-*$Tree_\alpha$ $tset_\alpha$))

by (*metis* $Tree_\alpha$*-rep-abs alpha-Tree-FL-valid-Tree*)
then show *?thesis*
by (*simp add*: $FL\text{-}valid\text{-}Tree_\alpha$*-def* $Conj_\alpha$*-def map-bset.rep-eq*)
qed

lemma $FL\text{-}valid\text{-}Tree_\alpha\text{-}Not_\alpha$ [*simp*]: $FL\text{-}valid\text{-}Tree_\alpha\ P\ (Not_\alpha\ t_\alpha) \longleftrightarrow \neg\ FL\text{-}valid\text{-}Tree_\alpha\ P\ t_\alpha$
by *transfer simp*

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

lemma $FL\text{-}valid\text{-}Tree_\alpha\text{-}Act_\alpha$ [*simp*]: $FL\text{-}valid\text{-}Tree_\alpha\ P\ (Act_\alpha\ f\ \alpha\ t_\alpha) \longleftrightarrow (\exists \alpha'\ t_\alpha'\ P'.\ Act_\alpha\ f\ \alpha\ t_\alpha = Act_\alpha\ f\ \alpha'\ t_\alpha' \wedge \langle f\rangle P \to \langle \alpha', P'\rangle \wedge FL\text{-}valid\text{-}Tree_\alpha\ P'\ t_\alpha')$
proof
assume $FL\text{-}valid\text{-}Tree_\alpha\ P\ (Act_\alpha\ f\ \alpha\ t_\alpha)$
moreover have $Act_\alpha\ f\ \alpha\ t_\alpha = abs\text{-}Tree_\alpha\ (tAct\ f\ \alpha\ (rep\text{-}Tree_\alpha\ t_\alpha))$
by (*metis* Act_α*.abs-eq* $Tree_\alpha$*-abs-rep*)
ultimately show $\exists \alpha'\ t_\alpha'\ P'.\ Act_\alpha\ f\ \alpha\ t_\alpha = Act_\alpha\ f\ \alpha'\ t_\alpha' \wedge \langle f\rangle P \to \langle \alpha', P'\rangle \wedge FL\text{-}valid\text{-}Tree_\alpha\ P'\ t_\alpha'$
by (*metis* Act_α*.abs-eq* $Tree_\alpha$*.abs-eq-iff FL-valid-Tree.simps*(*4*) $FL\text{-}valid\text{-}Tree_\alpha$*.abs-eq*)
next
assume $\exists \alpha'\ t_\alpha'\ P'.\ Act_\alpha\ f\ \alpha\ t_\alpha = Act_\alpha\ f\ \alpha'\ t_\alpha' \wedge \langle f\rangle P \to \langle \alpha', P'\rangle \wedge FL\text{-}valid\text{-}Tree_\alpha\ P'\ t_\alpha'$
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_α*.abs-eq* $Tree_\alpha$*-abs-rep*)
ultimately show $FL\text{-}valid\text{-}Tree_\alpha\ P\ (Act_\alpha\ f\ \alpha\ t_\alpha)$
by (*metis* $Tree_\alpha$*.abs-eq-iff FL-valid-Tree.simps*(*4*) $FL\text{-}valid\text{-}Tree_\alpha$*.abs-eq* $FL\text{-}valid\text{-}Tree_\alpha$*.rep-eq*)
qed

13.3 Validity for infinitary formulas

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

lemma *FL-valid-eqvt* [*eqvt*]:
assumes $P \models x$ **shows** $(p \cdot P) \models (p \cdot x)$
using *assms* **by** *transfer* (*metis* $FL\text{-}valid\text{-}Tree_\alpha$*-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*]: $P \models \mathit{Pred}\ f\ \varphi \longleftrightarrow \langle f \rangle P \vdash \varphi$
by *transfer simp*

lemma *FL-valid-Act*: $P \models \mathit{Act}\ f\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x')$
proof
assume $P \models \mathit{Act}\ f\ \alpha\ x$
moreover have *Rep-formula* (*Abs-formula* (Act_α f α (*Rep-formula* x))) = Act_α f α (*Rep-formula* x)
by (*metis Act.rep-eq Rep-formula-inverse*)
ultimately show $\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$
by (*auto simp add*: *FL-valid-def Act-def*) (*metis Abs-formula-inverse Rep-formula' hereditarily-fs-alpha-renaming*)
next
assume $\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x'$
then show $P \models \mathit{Act}\ f\ \alpha\ x$
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 $P \models \mathit{Act}\ f\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \mathit{bn}\ \alpha' \sharp* X)$
proof
assume $P \models \mathit{Act}\ f\ \alpha\ x$
then obtain $\alpha'\ x'\ P'$ **where** *eq*: $\mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x'$ **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$
by (*metis FL-valid-Act*)

have *finite* (*bn* α')
by (*fact bn-finite*)
moreover note ‹*finite* (*supp* X)›
moreover have *finite* (*supp* (*supp* x' − *bn* α', *supp* α' − *bn* α', $\langle \alpha', P' \rangle$))
by (*simp add*: *supp-Pair finite-sets-supp finite-supp*)
moreover have *bn* $\alpha' \sharp*$ (*supp* x' − *bn* α', *supp* α' − *bn* α', $\langle \alpha', P' \rangle$)
by (*simp add*: *atom-fresh-star-disjoint finite-supp fresh-star-Pair*)
ultimately obtain p **where** *fresh-X*: $(p \cdot \mathit{bn}\ \alpha') \sharp* X$ **and** *fresh-p*: *supp* (*supp* x' − *bn* α', *supp* α' − *bn* α', $\langle \alpha', P' \rangle$) $\sharp*$ p
by (*metis at-set-avoiding2*)

from *fresh-p* **have** *supp* (*supp* x' − *bn* α') $\sharp*$ p **and** *supp* (*supp* α' − *bn* α') $\sharp*$ p **and** *1*: *supp* $\langle \alpha', P' \rangle$ $\sharp*$ p
by (*meson fresh-Pair fresh-def fresh-star-def*)+
then have *2*: (*supp* x' − *bn* α') $\sharp*$ p **and** *3*: (*supp* α' − *bn* α') $\sharp*$ p
by (*simp add*: *finite-supp supp-finite-atom-set*)+

moreover from *2* **have** *supp* $(p \cdot x') - bn\ (p \cdot \alpha') = supp\ x' - bn\ \alpha'$
by (*metis Diff-eqvt atom-set-perm-eq bn-eqvt supp-eqvt*)
moreover from *3* **have** *supp* $(p \cdot \alpha') - bn\ (p \cdot \alpha') = supp\ \alpha' - bn\ \alpha'$
by (*metis* (*no-types*, *hide-lams*) *Diff-eqvt atom-set-perm-eq bn-eqvt supp-eqvt*)
ultimately have $Act\ f\ \alpha'\ x' = Act\ f\ (p \cdot \alpha')\ (p \cdot x')$
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 $\exists \alpha'\ x'\ P'.\ Act\ f\ \alpha\ x = Act\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge bn\ \alpha' \sharp* X$
using *eq* **and** *trans* **and** *valid* **and** *fresh-X* **by** (*metis bn-eqvt FL-valid-eqvt*)
next
assume $\exists \alpha'\ x'\ P'.\ Act\ f\ \alpha\ x = Act\ f\ \alpha'\ x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge bn\ \alpha' \sharp* X$
then show $P \models Act\ f\ \alpha\ x$ **by** (*metis FL-valid-Act*)
qed

lemma *FL-valid-Act-fresh*:
assumes $bn\ \alpha \sharp* \langle f \rangle P$
shows $P \models Act\ f\ \alpha\ x \longleftrightarrow (\exists P'.\ \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x)$
proof
assume $P \models Act\ f\ \alpha\ x$

moreover have *finite* (*supp* $(\langle f \rangle P)$)
by (*fact finite-supp*)
ultimately obtain $\alpha'\ x'\ P'$ **where**
eq: $Act\ f\ \alpha\ x = Act\ f\ \alpha'\ x'$ **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$ **and** *fresh*: $bn\ \alpha' \sharp* \langle f \rangle P$
by (*metis FL-valid-Act-strong*)

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

from *assms* **and** *fresh* **have** $(bn\ \alpha \cup p \cdot bn\ \alpha) \sharp* \langle f \rangle P$
using *p-α* **by** (*metis bn-eqvt fresh-star-Un*)
then have $supp\ p \sharp* \langle f \rangle P$
using *supp-p* **by** (*metis fresh-star-def subset-eq*)
then have *p-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-P p-α* **by** (*metis permute-minus-cancel*(*1*) *transition-eqvt'*)
moreover from *valid* **have** $-p \cdot P' \models x$
using *p-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 $P \models Act\ f\ \alpha\ x$
by (*metis FL-valid-Act*)
qed

end

end
theory *FL-Logical-Equivalence*
imports
FL-Validity
begin

14 (Strong) Logical Equivalence

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

locale *indexed-effect-nominal-ts* = *effect-nominal-ts satisfies transition effect-apply*
for *satisfies* :: *'state*::*fs* ⇒ *'pred*::*fs* ⇒ *bool* (**infix** ⊢ *70*)
and *transition* :: *'state* ⇒ (*'act*::*bn*,*'state*) *residual* ⇒ *bool* (**infix** → *70*)
and *effect-apply* :: *'effect*::*fs* ⇒ *'state* ⇒ *'state* (⟨-⟩- [*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* ⇒ *'state* ⇒ *'state* ⇒ *bool* **where**
FL-logically-equivalent F P Q ≡
∀ *x*::(*'idx*,*'pred*,*'act*,*'effect*) *formula*. *x* ∈ $\mathcal{A}$[*F*] ⟶ (*P* ⊨ *x* ⟷ *Q* ⊨ *x*)

We could (but didn't need to) prove that this defines an equivariant equivalence relation.

end

end
theory *FL-Bisimilarity-Implies-Equivalence*
imports
FL-Logical-Equivalence
begin

15 *F*/*L*-Bisimilarity Implies Logical Equivalence

context *indexed-effect-nominal-ts*
begin

lemma *FL-bisimilarity-implies-equivalence-Act*:
assumes *f* ∈$_{fs}$ *F*

and $bn\ \alpha \sharp* (F, f)$
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 $P \models Act\ f\ \alpha\ x$
shows $Q \models Act\ f\ \alpha\ x$
proof −
have *finite* $(supp\ (\langle f \rangle Q, F, f))$
by (*fact finite-supp*)
with ‹$P \models Act\ f\ \alpha\ x$› **obtain** $\alpha'\ x'\ P'$ **where** *eq*: $Act\ f\ \alpha\ x = Act\ f\ \alpha'\ x'$ **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: $P' \models x'$ **and** *fresh*: $bn\ \alpha' \sharp* (\langle f \rangle Q, F, f)$
by (*metis FL-valid-Act-strong*)

from ‹$P \sim\cdot[F]\ Q$› **and** ‹$f \in_{fs} F$› **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-α*: $\alpha' = p \cdot \alpha$ **and** *p-x*: $x' = p \cdot x$
and *fresh-p*: $(supp\ x - bn\ \alpha) \sharp* p \wedge (supp\ \alpha - bn\ \alpha) \sharp* p$
and *supp-p*: $supp\ p \subseteq bn\ \alpha \cup p \cdot bn\ \alpha$
by (*metis Act-eq-iff-perm-renaming*)

from *valid* **and** *p-x* **have** $-p \cdot P' \models x$
by (*metis permute-minus-cancel*(*2*) *FL-valid-eqvt*)

moreover from *fresh* **and** *p-α* **have** $(p \cdot bn\ \alpha) \sharp* (F, f)$
by (*simp add*: *bn-eqvt fresh-star-Pair*)
with ‹$bn\ \alpha \sharp* (F, f)$› **and** *supp-p* **have** $supp\ (F, f) \sharp* p$
by (*meson UnE fresh-def fresh-star-def subsetCE*)
then have $supp\ F \sharp* p$ **and** $supp\ f \sharp* p$
by (*simp add*: *fresh-star-Un supp-Pair*)+

with *bisim'* **and** *p-α* **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 ‹$\bigwedge P\ Q.\ P \sim\cdot[L\ (\alpha, F, f)]\ Q \Longrightarrow P \models x \longleftrightarrow Q \models x$› **by** *metis*

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

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

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

show $x \in \mathcal{A}[F] \longrightarrow P \models x \longleftrightarrow Q \models x$
proof
assume $x \in \mathcal{A}[F]$ **then show** $P \models x \longleftrightarrow Q \models x$
using *assms* **proof** (*induction x arbitrary*: *P Q*)
case *Conj* **then show** *?case*
by *simp*
next
case *Not* **then show** *?case*
by *simp*
next
case *Pred* **then show** *?case*
by (*metis FL-bisimilar-is-L-bisimulation is-L-bisimulation-def symp-def FL-valid-Pred*)
next
case *Act* **then show** *?case*
by (*metis FL-bisimilar-symp FL-bisimilarity-implies-equivalence-Act sympE*)
qed
qed
qed

end

end
theory *FL-Equivalence-Implies-Bisimilarity*
imports
FL-Logical-Equivalence
begin

16 Logical Equivalence Implies F/L-Bisimilarity

context *indexed-effect-nominal-ts*
begin

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

lemma *distinguishing-formula-eqvt* :
assumes *x distinguishes P from Q* **shows** (*p* ∙ *x*) *distinguishes* (*p* ∙ *P*) *from* (*p* ∙ *Q*)
using *assms* **unfolding** *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 distinguishes P from Q*)
by (*meson FL-logically-equivalent-def Not distinguishing-formula-def FL-valid-Not*)

There exists a distinguishing formula for P and Q in $\mathcal{A}[F]$ whose support is contained in *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** *supp* $y \subseteq$ *supp* (F,P) **and** y *distinguishes* P *from* Q
proof −
let *?B* $= \{p \cdot x | p.$ *supp* $(F,P) \sharp* p\}$
have *supp* (F,P) *supports ?B*
unfolding *supports-def* **proof** (*clarify*)
fix a b
assume a: $a \notin$ *supp* (F,P) **and** b: $b \notin$ *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*: *supp* $(F,P) \sharp* p$
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** *supp* $(F,P) \sharp* ?q$
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*: *supp* $(F,P) \sharp* p$
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** *supp* $(F,P) \sharp* ?q$
by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps*(*3*))
ultimately show $x' \in (a \rightleftharpoons b) \cdot$ *?B*
using *mem-permute-iff* **by** *blast*
qed
ultimately show $(a \rightleftharpoons b) \cdot$ *?B* $=$ *?B* **..**
qed
then have *supp-B-subset-supp-P*: *supp ?B* $\subseteq$ *supp* (F,P)
by (*metis* (*erased*, *lifting*) *finite-supp supp-is-subset*)
then have *finite-supp-B*: *finite* (*supp ?B*)
using *finite-supp rev-finite-subset* **by** *blast*

have *?B* $\subseteq (\lambda p.\ p \cdot x)$ ‘ *UNIV*
by *auto*
then have $|$*?B*$| \leq o\ |$*UNIV* :: *perm set*$|$

```
    by (rule surj-imp-ordLeq)
  also have |UNIV :: perm set| <o |UNIV :: 'idx set|
    by (metis card-idx-perm)
  also have |UNIV :: 'idx set| ≤o natLeq +c |UNIV :: 'idx set|
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally have card-B: |?B| <o natLeq +c |UNIV :: 'idx set| .

  let ?y = Conj (Abs-bset ?B) :: ('idx, 'pred, 'act, 'effect) formula

   from finite-supp-B and card-B and supp-B-subset-supp-P have supp ?y ⊆
supp (F,P)
    by simp
  moreover have ?y ∈ A[F]
    proof
      show finite (supp (Abs-bset ?B :: (-,-,-,-) formula set['idx]))
        using finite-supp-B card-B by simp
    next
      fix x'
      assume x' ∈ set-bset (Abs-bset ?B :: (-,-,-,-) formula set['idx])
      then obtain p where p-x: x' = p • x and fresh-p: supp (F,P) ♯* p
        using card-B by auto
      from fresh-p have p • F = F
        using fresh-star-Pair fresh-star-supp-conv perm-supp-eq by blast
      with ‹x ∈ A[F]› show x' ∈ A[F]
        using p-x by (metis is-FL-formula-eqvt)
    qed
  moreover have ?y distinguishes P from Q
    unfolding distinguishing-formula-def proof
      from ‹x distinguishes P from Q› show P ⊨ ?y
       by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
fresh-star-Un supp-Pair supp-perm-eq FL-valid-eqvt)
    next
      from ‹x distinguishes P from Q› show ¬ Q ⊨ ?y
       by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
    qed
  ultimately show ?thesis
    using that by blast
 qed

 lemma FL-equivalence-is-L-bisimulation: is-L-bisimulation FL-logically-equivalent
 proof −
  {
    fix F have symp (FL-logically-equivalent F)
      by (rule sympI) (metis FL-logically-equivalent-def)
  }
  moreover
  {
    fix F P Q f φ
```

assume *FL-logically-equivalent F P Q* **and** $f \in_{fs} F$ **and** $\langle f \rangle P \vdash \varphi$
then have $\langle f \rangle Q \vdash \varphi$
by (*metis FL-logically-equivalent-def Pred FL-valid-Pred*)
}
moreover
{
fix *F P Q f* α P'
assume *FL-logically-equivalent F P Q* **and** $f \in_{fs} F$ **and** *bn* $\alpha \sharp* (\langle f \rangle Q, F, f)$ **and** $\langle f \rangle P \to \langle \alpha, P' \rangle$
then have $\exists\, Q'.\ \langle f \rangle Q \to \langle \alpha, Q' \rangle \wedge$ *FL-logically-equivalent* (*L* (α,F,f)) P' Q'
proof −
{
let $?Q' = \{Q'.\ \langle f \rangle Q \to \langle \alpha, Q' \rangle\}$
assume $\forall\, Q' \in ?Q'.\ \neg$ *FL-logically-equivalent* (*L* (α,F,f)) P' Q'
then have $\forall\, Q' \in ?Q'.\ \exists\, x :: ('idx, 'pred, 'act, 'effect)$ *formula*. $x \in \mathcal{A}[L (\alpha,F,f)] \wedge x$ *distinguishes* P' *from* Q'
by (*metis FL-equivalent-iff-not-distinguished*)
then have $\forall\, Q' \in ?Q'.\ \exists\, x :: ('idx, 'pred, 'act, 'effect)$ *formula*. $x \in \mathcal{A}[L (\alpha,F,f)] \wedge$ *supp* $x \subseteq$ *supp* (*L* (α,F,f), P') $\wedge x$ *distinguishes* P' *from* Q'
by (*metis FL-distinguished-bounded-support*)
then obtain $g :: 'state \Rightarrow ('idx, 'pred, 'act, 'effect)$ *formula* **where**
$*$: $\forall\, Q' \in ?Q'.\ g\ Q' \in \mathcal{A}[L (\alpha,F,f)] \wedge$ *supp* $(g\ Q') \subseteq$ *supp* (*L* (α,F,f), P') $\wedge$ $(g\ Q')$ *distinguishes* P' *from* Q'
by *metis*
have *supp* (*L* (α,F,f), P') *supports* $(g$ ‘ $?Q')$
unfolding *supports-def* **proof** (*clarify*)
fix *a b*
assume *a*: $a \notin$ *supp* (*L* (α,F,f), P') **and** *b*: $b \notin$ *supp* (*L* (α,F,f), P')
have $(a \rightleftharpoons b) \cdot (g$ ‘ $?Q') \subseteq g$ ‘ $?Q'$
proof
fix x'
assume $x' \in (a \rightleftharpoons b) \cdot (g$ ‘ $?Q')$
then obtain Q' **where** *1*: $x' = (a \rightleftharpoons b) \cdot g\ Q'$ **and** *2*: $\langle f \rangle Q \to \langle \alpha, Q' \rangle$
by *auto* (*metis effect-apply-eqvt' permute-swap-cancel transition-eqvt'*)
with $*$ **and** *a* **and** *b* **have** $a \notin$ *supp* $(g\ Q')$ **and** $b \notin$ *supp* $(g\ Q')$
by *auto*
with *1* **have** $x' = g\ Q'$
by (*metis fresh-perm fresh-star-def supp-perm-eq swap-atom*)
with *2* **show** $x' \in g$ ‘ $?Q'$
by *simp*
qed
moreover have g ‘ $?Q' \subseteq (a \rightleftharpoons b) \cdot (g$ ‘ $?Q')$
proof
fix x'
assume $x' \in g$ ‘ $?Q'$
then obtain Q' **where** *1*: $x' = g\ Q'$ **and** *2*: $\langle f \rangle Q \to \langle \alpha, Q' \rangle$
by *auto*
with $*$ **and** *a* **and** *b* **have** $a \notin$ *supp* $(g\ Q')$ **and** $b \notin$ *supp* $(g\ Q')$

```
      by auto
    with 1 have x' = (a ⇌ b) · g Q'
      by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)
    with 2 show x' ∈ (a ⇌ b) · (g ‘ ?Q')
      using mem-permute-iff by blast
  qed
  ultimately show (a ⇌ b) · (g ‘ ?Q') = g ‘ ?Q' ..
 qed
then have supp-image-subset-supp-P': supp (g ‘ ?Q') ⊆ supp (L (α,F,f),
P')
  by (metis (erased, lifting) finite-supp supp-is-subset)
 then have finite-supp-image: finite (supp (g ‘ ?Q'))
  using finite-supp rev-finite-subset by blast
 have |g ‘ ?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: |g ‘ ?Q'| <o natLeq +c |UNIV :: 'idx set| .
 let ?y = Conj (Abs-bset (g ‘ ?Q')) :: ('idx, 'pred, 'act, 'effect) formula
 have Act f α ?y ∈ A[F]
  proof
   from ‹f ∈fs F› show f ∈fs F .
  next
   from ‹bn α ♯* (⟨f⟩Q, F, f)› show bn α ♯* (F, f)
    using fresh-star-Pair by blast
  next
   show Conj (Abs-bset (g ‘ ?Q')) ∈ A[L (α, 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' ∈ set-bset (Abs-bset (g ‘ ?Q') :: (-,-,-,-) formula set['idx])
     then obtain Q' where x' = g Q' and ⟨f⟩Q → ⟨α,Q'⟩
      using card-image by auto
     with ∗ show x' ∈ A[L (α, F, f)]
      using mem-Collect-eq by blast
    qed
  qed
 moreover have P ⊨ Act f α ?y
  unfolding FL-valid-Act proof (standard+)
   show ⟨f⟩P → ⟨α,P'⟩ by fact
  next
   {
    fix Q'
    assume ⟨f⟩Q → ⟨α,Q'⟩
    with ∗ have P' ⊨ g Q'
```

```
            by (metis distinguishing-formula-def mem-Collect-eq)
          }
          then show P' ⊨ ?y
            by (simp add: finite-supp-image card-image)
        qed
       moreover have ¬ Q ⊨ Act f α ?y
        proof
          assume Q ⊨ Act f α ?y
          then obtain Q' where 1: ⟨f⟩Q → ⟨α,Q'⟩ and 2: Q' ⊨ ?y
       using ‹bn α ♯* (⟨f⟩Q, F, f)› by (metis fresh-star-Pair FL-valid-Act-fresh)
          from 2 have ⋀Q''. ⟨f⟩Q → ⟨α,Q''⟩ ⟶ Q' ⊨ g Q''
            by (simp add: finite-supp-image card-image)
          with 1 and * show False
            using distinguishing-formula-def by blast
        qed
       ultimately have False
        by (metis ‹FL-logically-equivalent F P Q› 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 ∼·[F] Q
 using assms by (metis FL-bisimilar-def FL-equivalence-is-L-bisimulation)

end

end
theory L-Transform
imports
  Validity
  Bisimilarity-Implies-Equivalence
  FL-Equivalence-Implies-Bisimilarity
begin
```

17 L-Transform

17.1 States

The intuition is that states of kind AC can perform ordinary actions, and states of kind EF can commit effects.

```
datatype ('state,'effect) L-state =
    AC 'effect × 'effect fs-set × 'state
  | EF 'effect fs-set × 'state
```

```
instantiation L-state :: (pt,pt) pt
begin

  fun permute-L-state :: perm ⇒ ('a,'b) L-state ⇒ ('a,'b) L-state where
    p · (AC x) = AC (p · x)
  | p · (EF x) = EF (p · x)

  instance
  proof
    fix x :: ('a,'b) L-state
    show 0 · x = x by (cases x, simp-all)
  next
    fix p q and x :: ('a,'b) L-state
    show (p + q) · x = p · q · x by (cases x, simp-all)
  qed

end

declare permute-L-state.simps [eqvt]

lemma supp-AC [simp]: supp (AC x) = supp x
unfolding supp-def by simp

lemma supp-EF [simp]: supp (EF x) = supp x
unfolding supp-def by simp

instantiation L-state :: (fs,fs) fs
begin

  instance
  proof
    fix x :: ('a,'b) L-state
    show finite (supp x)
      by (cases x) (simp add: finite-supp)+
  qed

end
```

17.2 Actions and binding names

```
datatype ('act,'effect) L-action =
    Act 'act
  | Eff 'effect

instantiation L-action :: (pt,pt) pt
begin

  fun permute-L-action :: perm ⇒ ('a,'b) L-action ⇒ ('a,'b) L-action where
```

```
  p · (Act α) = Act (p · α)
| p · (Eff f) = Eff (p · f)

instance
proof
  fix x :: ('a,'b) L-action
  show 0 · x = x by (cases x, simp-all)
next
  fix p q and x :: ('a,'b) L-action
  show (p + q) · x = p · q · x by (cases x, simp-all)
qed

end

declare permute-L-action.simps [eqvt]

lemma supp-Act [simp]: supp (Act α) = supp α
unfolding supp-def by simp

lemma supp-Eff [simp]: supp (Eff f) = supp f
unfolding supp-def by simp

instantiation L-action :: (fs,fs) fs
begin

instance
proof
  fix x :: ('a,'b) L-action
  show finite (supp x)
    by (cases x) (simp add: finite-supp)+
qed

end

instantiation L-action :: (bn,fs) bn
begin

fun bn-L-action :: ('a,'b) L-action ⇒ atom set where
  bn-L-action (Act α) = bn α
| bn-L-action (Eff -) = {}

instance
proof
  fix p and α :: ('a,'b) L-action
  show p · bn α = bn (p · α)
    by (cases α) (simp add: bn-eqvt, simp)
next
  fix α :: ('a,'b) L-action
  show finite (bn α)
```

by (*cases* α) (*simp add*: *bn-finite*, *simp*)
qed

end

17.3 Satisfaction

context *effect-nominal-ts*
begin

fun *L-satisfies* :: (*'state*,*'effect*) *L-state* ⇒ *'pred* ⇒ *bool* (**infix** $\vdash_L$ *70*) **where**
AC (-,-,*P*) $\vdash_L \varphi \longleftrightarrow P \vdash \varphi$
| *EF* - $\vdash_L \varphi \longleftrightarrow$ *False*

lemma *L-satisfies-eqvt*: **assumes** $P_L \vdash_L \varphi$ **shows** $(p \cdot P_L) \vdash_L (p \cdot \varphi)$
proof (*cases* P_L)
case (*AC fFP*)
with *assms* **have** *snd* (*snd fFP*) $\vdash \varphi$
by (*metis L-satisfies.simps*(*1*) *prod.collapse*)
then have *snd* (*snd* ($p \cdot$ *fFP*)) $\vdash p \cdot \varphi$
by (*metis satisfies-eqvt snd-eqvt*)
then show *?thesis*
using *AC* **by** (*metis L-satisfies.simps*(*1*) *permute-L-state.simps*(*1*) *prod.collapse*)
next
case *EF*
with *assms* **have** *False*
by *simp*
then show *?thesis* **..**
qed

end

17.4 Transitions

context *effect-nominal-ts*
begin

fun *L-transition* :: (*'state*,*'effect*) *L-state* ⇒ ((*'act*,*'effect*) *L-action*, (*'state*,*'effect*) *L-state*) *residual* ⇒ *bool* (**infix** $\rightarrow_L$ *70*) **where**
AC (*f*,*F*,*P*) $\rightarrow_L \alpha P' \longleftrightarrow (\exists \alpha\ P'.\ P \rightarrow \langle \alpha, P' \rangle \wedge \alpha P' = \langle Act\ \alpha,\ EF\ (L\ (\alpha, F, f),\ P') \rangle \wedge bn\ \alpha \sharp* (F, f))$ — note the freshness condition
| *EF* (*F*,*P*) $\rightarrow_L \alpha P' \longleftrightarrow (\exists f.\ f \in_{fs} F \wedge \alpha P' = \langle Eff\ f,\ AC\ (f,\ F,\ \langle f \rangle P) \rangle)$

lemma *L-transition-eqvt*: **assumes** $P_L \rightarrow_L \alpha_L P_L'$ **shows** $(p \cdot P_L) \rightarrow_L (p \cdot \alpha_L P_L')$
proof (*cases* P_L)
case *AC*
{
fix *f F P*
assume ∗: $P_L = AC\ (f, F, P)$

with *assms* **obtain** α P' **where** *trans*: $P \to \langle \alpha, P' \rangle$ **and** $\alpha P'$: ${\alpha_L P_L}' = \langle Act\ \alpha,\ EF\ (L\ (\alpha,F,f),\ P') \rangle$ **and** *fresh*: *bn* α $\sharp*$ (F,f)
by *auto*
from *trans* **have** $p \cdot P \to \langle p \cdot \alpha,\ p \cdot P' \rangle$
by (*simp add*: *transition-eqvt'*)
moreover from $\alpha P'$ **have** $p \cdot {\alpha_L P_L}' = \langle Act\ (p \cdot \alpha),\ EF\ (L\ (p \cdot \alpha,\ p \cdot F,\ p \cdot f),\ p \cdot P') \rangle$
by (*simp add*: *L-eqvt'*)
moreover from *fresh* **have** *bn* $(p \cdot \alpha)$ $\sharp*$ $(p \cdot F,\ p \cdot f)$
by (*metis bn-eqvt fresh-star-Pair fresh-star-permute-iff*)
ultimately have $p \cdot P_L \to_L p \cdot {\alpha_L P_L}'$
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 $p \cdot P_L \to_L p \cdot {\alpha_L P_L}'$
using $*$ *L-transition.simps*(*2*) *Pair-eqvt permute-L-state.simps*(*2*) **by** *force*
}
with *EF* **show** *?thesis*
by (*metis prod.collapse*)
qed

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

lemma *L-transition-AC-strong*:
assumes *finite* (*supp X*) **and** $AC\ (f,F,P) \to_L \langle \alpha_L, {P_L}' \rangle$
shows $\exists\, \alpha\ P'.\ P \to \langle \alpha, P' \rangle \wedge \langle \alpha_L, {P_L}' \rangle = \langle Act\ \alpha,\ EF\ (L\ (\alpha,F,f),\ P') \rangle \wedge bn\ \alpha\ \sharp*\ X$
using *assms* **proof** −
from ‹$AC\ (f,F,P) \to_L \langle \alpha_L, {P_L}' \rangle$› **obtain** α P' **where** *transition*: $P \to \langle \alpha, P' \rangle$ **and** *alpha*: $\langle \alpha_L, {P_L}' \rangle = \langle Act\ \alpha,\ EF\ (L\ (\alpha,F,f),\ P') \rangle$ **and** *fresh*: *bn* α $\sharp*$ (F,f)
by (*metis L-transition.simps*(*1*))
let *?Act* = *Act* α :: (*'act*,*'effect*) *L-action* — the type annotation prevents a type that is too polymorphic and doesn't fix *'effect*
have *finite* (*bn* α)
by (*fact bn-finite*)
moreover note ‹*finite* (*supp X*)›
moreover have *finite* (*supp* $(\langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P') \rangle, \langle \alpha, P' \rangle,\ F,\ f)$)
by (*metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair*)

moreover from *fresh* **have** $bn\ \alpha\ \sharp*\ (\langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle,\ \langle\alpha,P'\rangle,\ F,\ f)$
by (*auto simp add*: *fresh-star-def fresh-def supp-Pair supp-abs-residual-pair*)
ultimately obtain p **where** *fresh-X*: $(p \cdot bn\ \alpha)\ \sharp*\ X$ **and** $supp\ (\langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle,\ \langle\alpha,P'\rangle,\ F,\ f)\ \sharp*\ p$
by (*metis at-set-avoiding2*)
then have $supp\ \langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle\ \sharp*\ p$ **and** $supp\ \langle\alpha,P'\rangle\ \sharp*\ p$ **and** $supp\ (F,f)\ \sharp*\ p$
by (*metis fresh-star-Un supp-Pair*)+
then have $p \cdot \langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle = \langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle$ **and** $p \cdot \langle\alpha,P'\rangle = \langle\alpha,P'\rangle$ **and** $p \cdot (F,f) = (F,f)$
by (*metis supp-perm-eq*)+
then have $\langle Act\ (p \cdot \alpha),\ EF\ (L\ (p \cdot \alpha,\ F,\ f),\ p \cdot P')\rangle = \langle ?Act,\ EF\ (L\ (\alpha,F,f),\ P')\rangle$ **and** $\langle p \cdot \alpha,\ p \cdot P'\rangle = \langle\alpha,P'\rangle$
using *permute-L-action.simps*(*1*) *permute-L-state.simps*(*2*) *abs-residual-pair-eqvt L-eqvt' Pair-eqvt* **by** *auto*
then show $\exists\alpha\ P'.\ P \rightarrow \langle\alpha,P'\rangle \wedge \langle\alpha_L,P_L'\rangle = \langle Act\ \alpha,\ EF\ (L\ (\alpha,F,f),\ P')\rangle \wedge bn\ \alpha\ \sharp*\ X$
using *transition* **and** *alpha* **and** *fresh-X* **by** (*metis bn-eqvt*)
qed

lemma *L-transition-AC-fresh*:
assumes $bn\ \alpha\ \sharp*\ (F,f,P)$
shows $AC\ (f,F,P) \rightarrow_L \langle Act\ \alpha,\ P_L'\rangle \longleftrightarrow (\exists P'.\ P_L' = EF\ (L\ (\alpha,F,f),\ P') \wedge P \rightarrow \langle\alpha,P'\rangle)$
proof
assume $AC\ (f,F,P) \rightarrow_L \langle Act\ \alpha,\ P_L'\rangle$
moreover have *finite* $(supp\ (F,f,P))$
by (*fact finite-supp*)
ultimately obtain $\alpha'\ P'$ **where** *trans*: $P \rightarrow \langle\alpha',P'\rangle$ **and** *eq*: $\langle Act\ \alpha :: ('act,'effect)\ L\text{-}action,\ P_L'\rangle = \langle Act\ \alpha',\ EF\ (L\ (\alpha',F,f),\ P')\rangle$ **and** *fresh*: $bn\ \alpha'\ \sharp*\ (F,f,P)$
using *L-transition-AC-strong* **by** *blast*
from *eq* **obtain** p **where** p: $p \cdot (Act\ \alpha :: ('act,'effect)\ L\text{-}action,\ P_L') = (Act\ \alpha',\ EF\ (L\ (\alpha',F,f),\ P'))$ **and** *supp-p*: $supp\ p \subseteq bn\ (Act\ \alpha :: ('act,'effect)\ L\text{-}action) \cup p \cdot bn\ (Act\ \alpha :: ('act,'effect)\ L\text{-}action)$
using *residual-eq-iff-perm-renaming* **by** *metis*

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

from *supp-p* **and** *p-α* **and** *assms* **and** *fresh* **have** $supp\ p\ \sharp*\ (F,\ f,\ P)$
by (*simp add*: *bn-eqvt fresh-star-def*) *blast*
then have *p-F*: $p \cdot F = F$ **and** *p-f*: $p \cdot f = f$ **and** *p-P*: $p \cdot P = P$
by (*simp-all add*: *fresh-star-Pair perm-supp-eq*)

from *p-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*-α *p*-*F* *p*-*f* **by** *simp* (*metis* (*full-types*) *permute-minus-cancel*(*2*))

moreover from *trans* **have** $P \to \langle \alpha,\ -p \cdot P' \rangle$
using *p*-*P* **and** *p*-α **by** (*metis permute-minus-cancel*(*2*) *transition-eqvt'*)

ultimately show $\exists P'.\ P_L{}' = EF\ (L\ (\alpha,F,f),\ P') \wedge P \to \langle \alpha,P' \rangle$
by *blast*
next
assume $\exists P'.\ P_L{}' = EF\ (L\ (\alpha,F,f),\ P') \wedge P \to \langle \alpha,P' \rangle$
moreover from *assms* **have** *bn* $\alpha \sharp* (F,f)$
by (*simp add*: *fresh-star-Pair*)
ultimately show $AC\ (f,\ F,\ P) \to_L \langle Act\ \alpha,\ P_L{}' \rangle$
using *L-transition.simps*(*1*) **by** *blast*
qed

end

17.5 Translation of F/L-formulas into formulas without effects

Since we defined formulas via a manual quotient construction, we also need to define the L-transform via lifting from the underlying type of infinitely branching trees. As before, we cannot use **nominal_function** because that generates proof obligations where, for formulas of the form *FL-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-transform-Tree* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *Tree* ⇒ (*'idx*, *'pred*, (*'act*,*'eff*) *L-action*) $Formula.Tree_\alpha$ **where**
L-transform-Tree (*tConj tset*) = $Formula.Conj_\alpha$ (*map-bset L-transform-Tree tset*)
| *L-transform-Tree* (*tNot t*) = $Formula.Not_\alpha$ (*L-transform-Tree t*)
| *L-transform-Tree* (*tPred f* φ) = $Formula.Act_\alpha$ (*Eff f*) ($Formula.Pred_\alpha$ φ)
| *L-transform-Tree* (*tAct f* α *t*) = $Formula.Act_\alpha$ (*Eff f*) ($Formula.Act_\alpha$ (*Act* α) (*L-transform-Tree t*))

lemma *L-transform-Tree-eqvt* [*eqvt*]: $p \cdot$ *L-transform-Tree t* = *L-transform-Tree* ($p \cdot t$)
proof (*induct t*)
case (*tConj tset*)
then show *?case*
by *simp* (*metis* (*no-types*, *hide-lams*) *bset.map-cong0 map-bset-eqvt permute-fun-def permute-minus-cancel*(*1*))

qed *simp-all*

L-transform-Tree respects α-equivalence.

lemma *alpha-Tree-L-transform-Tree*:
assumes *alpha-Tree t1 t2*
shows *L-transform-Tree t1 = L-transform-Tree t2*
using *assms* **proof** (*induction t1 t2 rule*: *alpha-Tree-induct'*)
case (*alpha-tConj tset1 tset2*)
then have *rel-bset* (=) (*map-bset L-transform-Tree tset1*) (*map-bset L-transform-Tree tset2*)
by (*simp add*: *bset.rel-map*(*1*) *bset.rel-map*(*2*) *bset.rel-mono-strong*)
then show *?case*
by (*simp add*: *bset.rel-eq*)
next
case (*alpha-tAct f1 α1 t1 f2 α2 t2*)
from ‹*alpha-Tree* (*FL-Formula.Tree.tAct f1 α1 t1*) (*FL-Formula.Tree.tAct f2 α2 t2*)›
obtain *p* **where** ∗: (*bn α1*, *t1*) ≈*set alpha-Tree* (*supp-rel alpha-Tree*) *p* (*bn α2*, *t2*)
and ∗∗: (*bn α1*, *α1*) ≈*set* (=) *supp p* (*bn α2*, *α2*) **and** *f1 = f2*
by *auto*
from ∗ **have** *fresh*: (*supp-rel alpha-Tree t1 − bn α1*) ♯∗ *p* **and** *alpha*: *alpha-Tree* (*p · t1*) *t2* **and** *eq*: *p · bn α1 = bn α2*
by (*auto simp add*: *alpha-set*)
from *alpha-tAct.IH*(*2*) **have** *supp-rel Formula.alpha-Tree* (*Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t1*)) ⊆ *supp-rel alpha-Tree t1*
by (*metis* (*no-types*, *lifting*) *infinite-mono Formula.alpha-Tree-permute-rep-commute L-transform-Tree-eqvt mem-Collect-eq subsetI supp-rel-def*)
with *fresh* **have** *fresh'*: (*supp-rel Formula.alpha-Tree* (*Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t1*)) − *bn α1*) ♯∗ *p*
by (*meson DiffD1 DiffD2 DiffI fresh-star-def subsetCE*)
moreover from *alpha* **have** *alpha'*: *Formula.alpha-Tree* (*p · Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t1*)) (*Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t2*))
using *alpha-tAct.IH*(*1*) **by** (*metis Formula.alpha-Tree-permute-rep-commute L-transform-Tree-eqvt*)
moreover from *fresh' alpha' eq* **have** *supp-rel Formula.alpha-Tree* (*Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t1*)) − *bn α1 = supp-rel Formula.alpha-Tree* (*Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t2*)) − *bn α2*
by (*metis* (*mono-tags*) *Diff-eqvt Formula.alpha-Tree-eqvt' Formula.alpha-Tree-eqvt-aux Formula.alpha-Tree-supp-rel atom-set-perm-eq*)
ultimately have (*bn α1*, *Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t1*)) ≈*set Formula.alpha-Tree* (*supp-rel Formula.alpha-Tree*) *p* (*bn α2*, *Formula.rep-Tree*$_\alpha$ (*L-transform-Tree t2*))
using *eq* **by** (*simp add*: *alpha-set*)
moreover from ∗∗ **have** (*bn α1*, *Act α1*) ≈*set* (=) *supp p* (*bn α2*, *Act α2*)
by (*metis* (*mono-tags*, *lifting*) *L-Transform.supp-Act alpha-set permute-L-action.simps*(*1*))
ultimately have *Formula.Act*$_\alpha$ (*Act α1*) (*L-transform-Tree t1*) = *Formula.Act*$_\alpha$ (*Act α2*) (*L-transform-Tree t2*)
by (*auto simp add*: *Formula.Act*$_\alpha$*-eq-iff*)

with ‹*f1* = *f2*› **show** *?case*
by *simp*
qed *simp-all*

L-transform for trees modulo α-equivalence.

lift-definition *L-transform-Tree*$_\alpha$ :: (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *Tree*$_\alpha$ ⇒ (*'idx*, *'pred*, (*'act*,*'eff*) *L-action*) *Formula.Tree*$_\alpha$ **is**
L-transform-Tree
by (*fact alpha-Tree-L-transform-Tree*)

lemma *L-transform-Tree*$_\alpha$*-eqvt* [*eqvt*]: *p* ∙ *L-transform-Tree*$_\alpha$ *t*$_\alpha$ = *L-transform-Tree*$_\alpha$ (*p* ∙ *t*$_\alpha$)
by *transfer* (*simp*)

lemma *L-transform-Tree*$_\alpha$*-Conj*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Conj*$_\alpha$ *tset*$_\alpha$) = *Formula.Conj*$_\alpha$ (*map-bset L-transform-Tree*$_\alpha$ *tset*$_\alpha$)
by (*simp add*: *Conj*$_\alpha$*-def' L-transform-Tree*$_\alpha$*.abs-eq*) (*metis* (*no-types*, *lifting*) *L-transform-Tree*$_\alpha$*.rep-eq bset.map-comp bset.map-cong0 comp-apply*)

lemma *L-transform-Tree*$_\alpha$*-Not*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Not*$_\alpha$ *t*$_\alpha$) = *Formula.Not*$_\alpha$ (*L-transform-Tree*$_\alpha$ *t*$_\alpha$)
by *transfer simp*

lemma *L-transform-Tree*$_\alpha$*-Pred*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Pred*$_\alpha$ *f* *φ*) = *Formula.Act*$_\alpha$ (*Eff f*) (*Formula.Pred*$_\alpha$ *φ*)
by *transfer simp*

lemma *L-transform-Tree*$_\alpha$*-Act*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Act*$_\alpha$ *f* *α* *t*$_\alpha$) = *Formula.Act*$_\alpha$ (*Eff f*) (*Formula.Act*$_\alpha$ (*Act α*) (*L-transform-Tree*$_\alpha$ *t*$_\alpha$))
by *transfer simp*

lemma *finite-supp-map-bset-L-transform-Tree*$_\alpha$ [*simp*]:
assumes *finite* (*supp tset*$_\alpha$)
shows *finite* (*supp* (*map-bset L-transform-Tree*$_\alpha$ *tset*$_\alpha$))
proof −
have *eqvt map-bset* **and** *eqvt L-transform-Tree*$_\alpha$
by (*simp add*: *eqvtI*)+
then have *supp* (*map-bset L-transform-Tree*$_\alpha$) = {}
using *supp-fun-eqvt supp-fun-app-eqvt* **by** *blast*
then have *supp* (*map-bset L-transform-Tree*$_\alpha$ *tset*$_\alpha$) ⊆ *supp tset*$_\alpha$
using *supp-fun-app* **by** *blast*
with *assms* **show** *finite* (*supp* (*map-bset L-transform-Tree*$_\alpha$ *tset*$_\alpha$))
by (*metis finite-subset*)
qed

lemma *L-transform-Tree*$_\alpha$*-preserves-hereditarily-fs*:
assumes *hereditarily-fs t*$_\alpha$
shows *Formula.hereditarily-fs* (*L-transform-Tree*$_\alpha$ *t*$_\alpha$)
using *assms* **proof** (*induct rule*: *hereditarily-fs.induct*)

case ($Conj_\alpha$ $tset_\alpha$)
then show *?case*
by (*auto intro*!: *Formula.hereditarily-fs.Conj*$_\alpha$) (*metis imageE map-bset.rep-eq*)
next
case (Not_α t_α)
then show *?case*
by (*simp add*: *Formula.hereditarily-fs.Not*$_\alpha$)
next
case ($Pred_\alpha$ f φ)
then show *?case*
by (*simp add*: *Formula.hereditarily-fs.Act*$_\alpha$ *Formula.hereditarily-fs.Pred*$_\alpha$)
next
case (Act_α t_α f α)
then show *?case*
by (*simp add*: *Formula.hereditarily-fs.Act*$_\alpha$)
qed

L-transform for F/L-formulas.

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

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

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

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

lemma *L-transform-formula-Pred* [*simp*]: *L-transform-formula* (*Pred f* φ) = *Formula.Act*$_\alpha$ (*Eff f*) (*Formula.Pred*$_\alpha$ φ)
by *transfer simp*

lemma *L-transform-formula-Act* [*simp*]: *L-transform-formula* (*FL-Formula.Act f* α x) = *Formula.Act*$_\alpha$ (*Eff f*) (*Formula.Act*$_\alpha$ (*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*$_\alpha$*-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* ⇒ (*'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* ∙ *L-transform x* = *L-transform* (*p* ∙ *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*) ⊆ *supp xset*
using *supp-fun-app* **by** *blast*
with *assms* **show** *finite* (*supp* (*map-bset L-transform xset*))
by (*metis finite-subset*)
qed

lemma *L-transform-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows *L-transform* (*Conj xset*) = *Formula.Conj* (*map-bset L-transform xset*)
using *assms* **unfolding** *L-transform-def* **by** (*simp add*: *Formula.Conj-def bset.map-comp o-def*)

lemma *L-transform-Not* [*simp*]: *L-transform* (*Not x*) = *Formula.Not* (*L-transform x*)
unfolding *L-transform-def* **by** (*simp add*: *Formula.Not-def*)

lemma *L-transform-Pred* [*simp*]: *L-transform* (*Pred f φ*) = *Formula.Act* (*Eff f*) (*Formula.Pred φ*)
unfolding *L-transform-def* **by** (*simp add*: *Formula.Act-def Formula.Pred-def Formula.hereditarily-fs.Pred*$_\alpha$)

lemma *L-transform-Act* [*simp*]: *L-transform* (*FL-Formula.Act f α x*) = *Formula.Act* (*Eff f*) (*Formula.Act* (*Act α*) (*L-transform x*))
unfolding *L-transform-def* **by** (*simp add*: *Formula.Act-def Formula.hereditarily-fs.Act*$_\alpha$)

context *effect-nominal-ts*
begin

interpretation *L-transform*: *nominal-ts* (⊢$_L$) (→$_L$)
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* φ)
let $?\varphi$ = *Formula.Pred* φ :: (*'idx*, *'pred*, (*'act*,*'effect*) *L-action*) *Formula.formula*
show *?case*
proof
assume *FL-valid P* (*Pred f* φ)
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*) $\rightarrow_L$ $\langle$*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* φ))
using *L-transform.valid-Act* **by** *fastforce*
next
assume *L-transform.valid* (*EF* (*F*, *P*)) (*L-transform* (*Pred f* φ))
then obtain P' **where** *trans*: *EF* (*F*, *P*) $\rightarrow_L$ $\langle$*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* φ)
by *simp*
qed
next
case (*Act f F* α *x*)
show *?case*
proof
assume *FL-valid P* (*FL-Formula.Act f* α *x*)
then obtain α' x' P' **where** *eq*: *FL-Formula.Act f* α *x* = *FL-Formula.Act f* α' x' **and** *trans*: $\langle f \rangle P \rightarrow \langle \alpha', P' \rangle$ **and** *valid*: *FL-valid* P' x' **and** *fresh*: *bn* α' $\sharp*$ (*F*, *f*)
by (*metis FL-valid-Act-strong finite-supp*)
from *eq* **obtain** *p* **where** *p-x*: $p \cdot x = x'$ **and** *p-α*: $p \cdot \alpha = \alpha'$ **and** *supp-p*: *supp p* $\subseteq$ *bn* α $\cup$ *bn* α'
by (*metis bn-eqvt FL-Formula.Act-eq-iff-perm-renaming*)
from $\langle$*bn* α $\sharp*$ (*F*, *f*)$\rangle$ **and** *fresh* **have** *supp* (*F*, *f*) $\sharp*$ *p*
using *supp-p* **by** (*auto simp add*: *fresh-star-Pair fresh-star-def supp-Pair fresh-def*)
then have $p \cdot F = F$ **and** $p \cdot f = f$

using *supp-perm-eq* **by** *fastforce+*

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

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

— rename *bn* α to avoid (F, f, P), without touching F or *FL-Formula.Act* f α x
obtain p **where** *1*: $(p \cdot bn\ \alpha) \sharp* (F, f, P)$ **and** *2*: *supp* (F, *FL-Formula.Act* f α x) $\sharp*$ p
proof (*rule at-set-avoiding2*[*of bn* α (F, f, P) (F, *FL-Formula.Act* f α x), *THEN exE*])
show *finite* (*bn* α) **by** (*fact bn-finite*)
next
show *finite* (*supp* (F, f, P)) **by** (*fact finite-supp*)
next
show *finite* (*supp* (F, *FL-Formula.Act* f α x)) **by** (*simp add*: *finite-supp*)
next
from ‹*bn* α $\sharp*$ (F, f)› **show** *bn* α $\sharp*$ (F, *FL-Formula.Act* f α 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 α x) $\sharp*$ p
by (*simp add*: *fresh-star-Pair fresh-star-def supp-Pair*)+
from ‹*supp* F $\sharp*$ p› **have** $p \cdot F = F$
by (*metis supp-perm-eq*)
from *Act-fresh* **have** $p \cdot f = f$
using *fresh-star-Un supp-perm-eq* **by** *fastforce*
from *Act-fresh* **have** *eq*: *FL-Formula.Act* f α x = *FL-Formula.Act* f ($p \cdot \alpha$) ($p \cdot x$)
by (*metis FL-Formula.Act-eq-iff-perm FL-Formula.Act-eqvt supp-perm-eq*)

with $*$ **obtain** P' **where** *trans*: *EF* (F, P) $\rightarrow_L$ ⟨*Eff* f,P'⟩ **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-eqvt supp-fun-app supp-fun-app-eqvt* **by** *fastforce*
with *1* **have** $bn\ (Act\ (p \cdot \alpha)) \sharp* AC\ (f,\ F,\ \langle f\rangle P)$
by (*auto simp add*: *bn-eqvt fresh-star-def fresh-def supp-Pair*)
with *valid* **obtain** *P''* **where** *trans'*: $AC\ (f,\ F,\ \langle f\rangle P) \rightarrow_L \langle Act\ (p \cdot \alpha), P''\rangle$ **and** *valid'*: *L-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-eqvt fresh-star-def fresh-def supp-Pair*)
with *trans'* **obtain** *P'* **where** *P''*: $P'' = EF\ (L\ (p \cdot \alpha,\ F,\ f),\ P')$ **and** *trans''*: $\langle f\rangle P \rightarrow \langle p \cdot \alpha, P'\rangle$
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-eqvt L-transform-eqvt permute-minus-cancel*(*2*))
with *P''* ‹$p \cdot F = F$› ‹$p \cdot f = f$› **have** *L-transform.valid* $(EF\ (L\ (\alpha,\ F,\ f),\ -\ p \cdot P'))$ (*L-transform x*)
by *simp* (*metis pemute-minus-self permute-minus-cancel*(*1*))
then have *FL-valid P'* $(p \cdot x)$
using *Act.hyps*(*4*) **by** (*metis FL-valid-eqvt permute-minus-cancel*(*1*))

with *trans''* **and** *eq* **show** *FL-valid P* (*FL-Formula.Act f* α *x*)
by (*metis FL-valid-Act*)
qed
qed

end

17.6 Bisimilarity in the *L*-transform

context *effect-nominal-ts*
begin

interpretation *L-transform*: *nominal-ts* $(\vdash_L)$ $(\rightarrow_L)$
by *unfold-locales* (*fact L-satisfies-eqvt*, *fact L-transition-eqvt*)

notation *L-transform.bisimilar* (**infix** $\sim\cdot_L$ *100*)

F/L-bisimilarity is equivalent to bisimilarity in the L-transform.

inductive *L-bisimilar* :: ('*state*,'*effect*) *L-state* ⇒ ('*state*,'*effect*) *L-state* ⇒ *bool*
where
$P \sim\cdot[F]\ Q \Longrightarrow$ *L-bisimilar* $(EF\ (F,P))\ (EF\ (F,Q))$

| $P \sim\cdot[F]\ Q \Longrightarrow f \in_{fs} F \Longrightarrow$ *L-bisimilar* (*AC* (*f*, *F*, ⟨*f*⟩*P*)) (*AC* (*f*, *F*, ⟨*f*⟩*Q*))

lemma *L-bisimilar-is-bisimulation*: *L-transform.is-bisimulation L-bisimilar*
unfolding *L-transform.is-bisimulation-def*
proof
show *symp L-bisimilar*
by (*metis FL-bisimilar-symp L-bisimilar.cases L-bisimilar.intros symp-def*)
next
have $\forall P_L\ Q_L$. *L-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-bisimilar* $P_L\ Q_L \longrightarrow (\forall \alpha_L\ P_L'.\ bn\ \alpha_L \sharp* Q_L \longrightarrow P_L \rightarrow_L \langle \alpha_L, P_L' \rangle \longrightarrow (\exists Q_L'.\ Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \wedge$ *L-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* $\alpha_L \sharp* Q_L$ **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*: $P \sim\cdot[F]\ Q$
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*, ⟨*f*⟩*P*)) (*AC* (*f*, *F*, ⟨*f*⟩*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 ‹$\bigwedge Q_L'.\ Q_L \rightarrow_L \langle \alpha_L, Q_L' \rangle \Longrightarrow$ *L-bisimilar* $P_L'\ Q_L' \Longrightarrow$ *thesis*›
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*: $P \sim\cdot[F]\ Q$ **and** *effect*: $f \in_{fs} F$
have *finite* (*supp* (⟨*f*⟩*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* $\alpha \sharp* (\langle f \rangle Q, F, f)$
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'*: $P' \sim\cdot[L\ (\alpha, F, f)]\ Q'$
by (*metis FL-bisimilar-simulation-step*)
from *fresh* **have** *bn* $\alpha \sharp* (F, f)$

```
          by (meson fresh-PairD(2) fresh-star-def)
        with Q_L and trans-Q have trans-Q_L: Q_L →_L ⟨Act α, EF (L (α,F,f),
Q'⟩
          by (metis L-transition.simps(1))

       from α_L P_L' obtain p where p: (α_L,P_L') = p · (Act α, EF (L (α,F,f),
P')) and supp-p: supp p ⊆ bn α ∪ bn α_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 ♯* (⟨f⟩Q, F, f)
            unfolding fresh-star-def by (metis (no-types, hide-lams) Un-iff
fresh-Pair fresh-def subsetCE supp-AC)
       then have p-fQ: p · ⟨f⟩Q = ⟨f⟩Q and p-Ff: p · (F,f) = (F,f)
         by (simp add: fresh-star-def perm-supp-eq)+
       from p and p-Ff have α_L = Act (p · α) and P_L' = EF (L (p · α, F,
f), p · P')
         by auto

       moreover from Q_L and p-fQ and p-Ff have p · Q_L = Q_L
         by simp
       with trans-Q_L have Q_L →_L p · ⟨Act α, EF (L (α,F,f), Q')⟩
         by (metis L-transform.transition-eqvt)
       then have Q_L →_L ⟨Act (p · α), EF (L (p · α, F, f), p · Q')⟩
         using p-Ff by simp

       moreover from p-Ff have p · F = F and p · f = f
         by simp+
       with bisim' have (p · P') ∼·[L (p · α, F, f)] (p · Q')
         by (metis FL-bisimilar-eqvt L-eqvt')
       then have L-bisimilar (EF (L (p · α, F, f), p · P')) (EF (L (p · α, F,
f), p · Q'))
         by (metis L-bisimilar.intros(1))

       ultimately show thesis
         using ‹⋀Q_L'. Q_L →_L ⟨α_L,Q_L'⟩ ⟹ L-bisimilar P_L' Q_L' ⟹ thesis›
by blast
     qed
    then show ∃ Q_L'. Q_L →_L ⟨α_L,Q_L'⟩ ∧ L-bisimilar P_L' Q_L'
     by auto
   qed
  ultimately show ?S ∧ ?T
   by metis
 qed

 definition invL-FL-bisimilar :: 'effect first ⇒ 'state ⇒ 'state ⇒ bool where
  invL-FL-bisimilar F P Q ≡ EF (F,P) ∼·_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 $\forall P\ Q.$ *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*) $\sharp*$ $EF\ (F,Q)$
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 $\forall P\ Q.$ *invL-FL-bisimilar F P Q* $\longrightarrow$ $(\forall f.\ f \in_{fs} F \longrightarrow (\forall \alpha\ P'.$ *bn* $\alpha\ \sharp*\ (\langle f\rangle Q,\ F,\ f) \longrightarrow$ $\langle f\rangle P \rightarrow \langle \alpha,P'\rangle \longrightarrow (\exists Q'.\ \langle f\rangle Q \rightarrow \langle \alpha,Q'\rangle \wedge$ *invL-FL-bisimilar* $(L\ (\alpha,\ F,\ f))$ $P'\ Q'))))$ (**is** *?T*)
proof (*clarify*)
fix $P\ Q\ f\ \alpha\ P'$
assume *bisim*: *invL-FL-bisimilar F P Q* **and** *effect*: $f \in_{fs} F$ **and** *fresh*: *bn* $\alpha\ \sharp*\ (\langle f\rangle Q,\ F,\ f)$ **and** *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 *bn* (*Eff f*) $\sharp*$ $EF\ (F,Q)$
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** $trans_L$: $EF\ (F,Q) \rightarrow_L \langle Eff\ f,\ Q_L{'}\rangle$ **and** *L-bisim*: $AC\ (f,\ F,\ \langle f\rangle P) \sim_L Q_L{'}$
by (*metis L-transform.bisimilar-simulation-step*)
from $trans_L$ **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-bisim* **and** Q_L' **have** AC $(f$, F, $\langle f\rangle P) \sim\cdot_L$ AC $(f$, F, $\langle f\rangle Q)$
by *metis*
moreover from *fresh* **have** *bn* $(Act$ $\alpha) \sharp* AC$ $(f$, F, $\langle f\rangle Q)$
by (*simp add*: *fresh-def fresh-star-def supp-Pair*)
moreover from *fresh* **have** *bn* $\alpha \sharp* (F$, $f)$
by (*simp add*: *fresh-star-Pair*)
with *trans* **have** AC $(f$, F, $\langle f\rangle P) \rightarrow_L \langle Act$ α, EF $(L$ $(\alpha$,F,$f)$, $P'\rangle\rangle$
by (*metis L-transition.simps*(1))
ultimately obtain Q_L'' **where** $trans_L'$: AC $(f$, F, $\langle f\rangle Q) \rightarrow_L \langle Act$ α, $Q_L''\rangle$ **and** *L-bisim'*: EF $(L$ $(\alpha$,F,$f)$, $P') \sim\cdot_L$ Q_L''
by (*metis L-transform.bisimilar-simulation-step*)

have *finite* (*supp* $(\langle f\rangle Q$, F, $f))$
by (*fact finite-supp*)
with $trans_L'$ **obtain** α' Q' **where** *trans'*: $\langle f\rangle Q \rightarrow \langle \alpha'$,$Q'\rangle$ **and** *alpha*: $\langle Act$ α :: $('act$,$'effect)$ *L-action*, $Q_L''\rangle = \langle Act$ α', EF $(L$ $(\alpha'$,F,$f)$, $Q')\rangle$ **and** *fresh'*: *bn* $\alpha' \sharp* (\langle f\rangle Q$, F,$f)$
by (*metis L-transition-AC-strong*)

from *alpha* **obtain** p **where** p: $(Act$ α :: $('act$,$'effect)$ *L-action*, $Q_L'') = p$ $\cdot$ $(Act$ α', EF $(L$ $(\alpha'$,F,$f)$, $Q'))$ **and** *supp-p*: *supp* $p \subseteq$ *bn* $\alpha \cup$ *bn* α'
by (*metis Un-commute bn-L-action.simps*(1) *residual-eq-iff-perm-renaming*)
from *supp-p* **and** *fresh* **and** *fresh'* **have** *supp* $p \sharp* (\langle f\rangle Q$, F,$f)$
unfolding *fresh-star-def* **by** (*metis* (*no-types*, *hide-lams*) *Un-iff subsetCE*)
then have *p-fQ*: $p \cdot \langle f\rangle Q = \langle f\rangle Q$ **and** *p-F*: $p \cdot F = F$ **and** *p-f*: $p \cdot f = f$
by (*simp add*: *fresh-star-def perm-supp-eq*)+
from p **and** *p-F* **and** *p-f* **have** *p-α'*: $p \cdot \alpha' = \alpha$ **and** Q_L'': $Q_L'' = EF$ $(L$ $(p \cdot \alpha'$, F, $f)$, $p \cdot Q')$
by *auto*

from *trans'* **and** *p-fQ* **and** *p-α'* **have** $\langle f\rangle Q \rightarrow \langle \alpha$, $p \cdot Q'\rangle$
by (*metis transition-eqvt'*)
moreover from *L-bisim'* **and** Q_L'' **and** *p-α'* **have** *invL-FL-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-FL-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-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-bisimulation L-transform.bisimilar-def*)
next
assume *EF* (*F*, *P*) $\sim\cdot_L$ *EF* (*F*, *Q*)
then have *invL-FL-bisimilar F P Q*
by (*metis invL-FL-bisimilar-def*)
then show *P* $\sim\cdot[F]$ *Q*
by (*metis invL-FL-bisimilar-is-L-bisimulation FL-bisimilar-def*)
qed

end

The following (alternative) proof of the "←" 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 ⟦*indexed-nominal-ts TYPE*(*?'idx*) *?satisfies ?transition*; *nominal-ts.bisimilar ?satisfies ?transition ?P ?Q*⟧ ⟹ *indexed-nominal-ts.logically-equivalent TYPE*(*?'idx*) *?satisfies ?transition ?P ?Q* does not actually make use of the cardinality assumptions provided by indexed nominal transition systems.

locale *L-transform-indexed-effect-nominal-ts* = *indexed-effect-nominal-ts L satisfies transition effect-apply*
for *L* :: ('*act*::*bn*) × ('*effect*::*fs*) *fs-set* × '*effect* ⇒ '*effect fs-set*
and *satisfies* :: '*state*::*fs* ⇒ '*pred*::*fs* ⇒ *bool* (**infix** ⊢ *70*)
and *transition* :: '*state* ⇒ ('*act*,'*state*) *residual* ⇒ *bool* (**infix** → *70*)
and *effect-apply* :: '*effect* ⇒ '*state* ⇒ '*state* (⟨-⟩- [*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* (⊢$_L$) (→$_L$)
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** $\sim\cdot_L$ *100*)

theorem *EF* (*F*,*P*) $\sim\cdot_L$ *EF*(*F*,*Q*) ⟶ *P* $\sim\cdot[F]$ *Q*
proof
assume *EF* (*F*, *P*) $\sim\cdot_L$ *EF* (*F*, *Q*)

then have *L-transform.logically-equivalent* (*EF* (*F*, *P*)) (*EF* (*F*, *Q*))
by (*fact L-transform.bisimilarity-implies-equivalence*)
with *FL-valid-iff-valid-L-transform* **have** *FL-logically-equivalent F P Q*
using *FL-logically-equivalent-def L-transform.logically-equivalent-def* **by** *blast*
then show *P* ∼·[*F*] *Q*
by (*fact FL-equivalence-implies-bisimilarity*)
qed

end

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

18 Nominal Transition Systems and Bisimulations with Unobservable Transitions

18.1 Nominal transition systems with unobservable transitions

locale *weak-nominal-ts* = *nominal-ts satisfies transition*
for *satisfies* :: *'state::fs* ⇒ *'pred::fs* ⇒ *bool* (**infix** ⊢ *70*)
and *transition* :: *'state* ⇒ (*'act::bn*,*'state*) *residual* ⇒ *bool* (**infix** → *70*) +
fixes τ :: *'act*
assumes *tau-eqvt* [*eqvt*]: *p* · τ = τ
begin

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

lemma *bn-tau-fresh* [*simp*]: *bn* τ ♯* *P*
by (*simp add*: *fresh-star-def*)

inductive *tau-transition* :: *'state* ⇒ *'state* ⇒ *bool* (**infix** ⇒ *70*) **where**
tau-refl [*simp*]: *P* ⇒ *P*
| *tau-step*: ⟦ *P* → ⟨τ, *P'*⟩; *P'* ⇒ *P''* ⟧ ⟹ *P* ⇒ *P''*

definition *observable-transition* :: *'state* ⇒ *'act* ⇒ *'state* ⇒ *bool* (-/ ⇒{-}/ - [*70*, *70*, *71*] *71*) **where**
P ⇒{α} *P'* ≡ ∃ *Q Q'*. *P* ⇒ *Q* ∧ *Q* → ⟨α, *Q'*⟩ ∧ *Q'* ⇒ *P'*

definition *weak-transition* :: *'state* ⇒ *'act* ⇒ *'state* ⇒ *bool* (-/ ⇒⟨-⟩/ - [*70*, *70*, *71*] *71*) **where**
P ⇒⟨α⟩ *P'* ≡ *if* α = τ *then P* ⇒ *P' else P* ⇒{α} *P'*

The transition relations defined above are equivariant.

lemma *tau-transition-eqvt* :
assumes $P \Rightarrow P'$ **shows** $p \cdot P \Rightarrow p \cdot P'$
using *assms* **proof** (*induction*)
case (*tau-refl P*) **show** *?case*
by (*fact tau-transition.tau-refl*)
next
case (*tau-step P P' P''*)
from ‹$P \rightarrow \langle\tau,P'\rangle$› **have** $p \cdot P \rightarrow \langle\tau,p \cdot P'\rangle$
using *tau-eqvt transition-eqvt'* **by** *fastforce*
with ‹$p \cdot P' \Rightarrow p \cdot P''$› **show** *?case*
using *tau-transition.tau-step* **by** *blast*
qed

lemma *observable-transition-eqvt* :
assumes $P \Rightarrow\{\alpha\} P'$ **shows** $p \cdot P \Rightarrow\{p \cdot \alpha\} p \cdot P'$
using *assms* **unfolding** *observable-transition-def* **by** (*metis transition-eqvt' tau-transition-eqvt*)

lemma *weak-transition-eqvt* :
assumes $P \Rightarrow\langle\alpha\rangle P'$ **shows** $p \cdot P \Rightarrow\langle p \cdot \alpha\rangle p \cdot P'$
using *assms* **unfolding** *weak-transition-def* **by** (*metis* (*full-types*) *observable-transition-eqvt permute-minus-cancel*(*2*) *tau-eqvt tau-transition-eqvt*)

Additional lemmas about ($\Rightarrow$), *observable-transition* and *weak-transition*.

lemma *tau-transition-trans*:
assumes $P \Rightarrow Q$ **and** $Q \Rightarrow R$
shows $P \Rightarrow R$
using *assms* **by** (*induction*, *auto simp add*: *tau-step*)

lemma *observable-transitionI*:
assumes $P \Rightarrow Q$ **and** $Q \rightarrow \langle\alpha,\ Q'\rangle$ **and** $Q' \Rightarrow P'$
shows $P \Rightarrow\{\alpha\} P'$
using *assms observable-transition-def* **by** *blast*

lemma *observable-transition-stepI* [*simp*]:
assumes $P \rightarrow \langle\alpha,\ P'\rangle$
shows $P \Rightarrow\{\alpha\} P'$
using *assms observable-transitionI tau-refl* **by** *blast*

lemma *observable-tau-transition*:
assumes $P \Rightarrow\{\tau\} P'$
shows $P \Rightarrow P'$
proof −
from ‹$P \Rightarrow\{\tau\} P'$› **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

18.2 Weak bisimulations

context *weak-nominal-ts*
begin

definition *is-weak-bisimulation* :: (*'state* ⇒ *'state* ⇒ *bool*) ⇒ *bool* **where**
is-weak-bisimulation R ≡
symp R ∧
— weak static implication
(∀ *P Q φ*. *R P Q* ∧ *P* ⊢ *φ* ⟶ (∃ *Q'*. *Q* ⇒ *Q'* ∧ *R P Q'* ∧ *Q'* ⊢ *φ*)) ∧
— weak simulation
(∀ *P Q*. *R P Q* ⟶ (∀ *α P'*. *bn α* ♯* *Q* ⟶ *P* → ⟨*α*,*P'*⟩ ⟶ (∃ *Q'*. *Q* ⇒⟨*α*⟩ *Q'* ∧ *R P' Q'*)))

definition *weakly-bisimilar* :: *'state* ⇒ *'state* ⇒ *bool* (**infix** ≈· *100*) **where**
P ≈· *Q* ≡ ∃ *R*. *is-weak-bisimulation R* ∧ *R P Q*

(≈·) is an equivariant equivalence relation.

lemma *is-weak-bisimulation-eqvt* :
assumes *is-weak-bisimulation R* **shows** *is-weak-bisimulation* (*p* ∙ *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*: $\forall P\ Q.\ R\ P\ Q \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha\ \sharp*\ Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'.\ Q \Rightarrow\langle \alpha \rangle\ Q' \wedge R\ P'\ Q'))$
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 $\forall P\ Q.\ (p \cdot R)\ P\ Q \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha\ \sharp*\ Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'.\ Q \Rightarrow\langle \alpha \rangle\ Q' \wedge (p \cdot R)\ P'\ Q'))$ (**is** *?U*)
proof (*clarify*)
fix $P\ Q\ \alpha\ P'$
assume $*$: $(p \cdot R)\ P\ Q$ **and** $**$: $bn\ \alpha\ \sharp*\ Q$ **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 $bn\ (-p \cdot \alpha)\ \sharp*\ (-p \cdot Q)$
using $**$ **by** (*metis bn-eqvt fresh-star-permute-iff*)
moreover have $-p \cdot P \rightarrow \langle -p \cdot \alpha, -p \cdot P' \rangle$
using $***$ **by** (*metis transition-eqvt'*)
ultimately obtain Q' **where** T: $-p \cdot Q \Rightarrow\langle -p \cdot \alpha \rangle\ Q'$ **and** R: $R\ (-p \cdot P')\ Q'$
using *3* **by** *metis*
from T **have** $Q \Rightarrow\langle \alpha \rangle\ (p \cdot Q')$
by (*metis permute-minus-cancel*(*1*) *weak-transition-eqvt*)
moreover from R **have** $(p \cdot R)\ P'\ (p \cdot Q')$
by (*metis eqvt-apply eqvt-lambda permute-bool-def unpermute-def*)
ultimately show $\exists Q'.\ Q \Rightarrow\langle \alpha \rangle\ Q' \wedge (p \cdot R)\ P'\ Q'$
by *metis*
qed

ultimately show *?S* ∧ *?T* ∧ *?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* ∧ *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* ∧ *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* (≈·)
by (*fact weakly-bisimilar-symp*)
next
show $(\forall P\ Q\ \varphi.\ P \approx\cdot Q \wedge P \vdash \varphi \longrightarrow (\exists Q'.\ Q \Rightarrow Q' \wedge P \approx\cdot Q' \wedge Q' \vdash \varphi)) \wedge$
$(\forall P\ Q.\ P \approx\cdot Q \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha \sharp* Q \longrightarrow P \rightarrow \langle \alpha, P' \rangle \longrightarrow (\exists Q'.\ Q \Rightarrow \langle \alpha \rangle$
$Q' \wedge P' \approx\cdot Q')))$
by (*auto simp add*: *is-weak-bisimulation-def weakly-bisimilar-def*) *blast+*
qed

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

assumes $P \approx\cdot Q$ **and** $P \Rightarrow P'$
obtains Q' **where** $Q \Rightarrow Q'$ **and** $P' \approx\cdot Q'$
using ‹$P \Rightarrow P'$› ‹$P \approx\cdot Q$› **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 ‹$P \rightarrow \langle\tau,P''\rangle$› **and** ‹$P \approx\cdot Q$› **obtain** Q'' **where** $Q \Rightarrow Q''$ **and** $P'' \approx\cdot Q''$
by (*metis bn-tau-fresh is-weak-bisimulation-def weak-transition-def weakly-bisimilar-is-weak-bisimulation*)
then show *?case*
using *tau-step.hyps*(*3*) *tau-step.prems*(*1*) **by** (*metis tau-transition-trans*)
qed

lemma *weakly-bisimilar-weak-simulation-step*:
assumes $P \approx\cdot Q$ **and** *bn* $\alpha \sharp* Q$ **and** $P \Rightarrow\langle\alpha\rangle P'$
obtains Q' **where** $Q \Rightarrow\langle\alpha\rangle Q'$ **and** $P' \approx\cdot Q'$
proof (*cases* $\alpha = \tau$)
case *True* **with** ‹$P \approx\cdot Q$› **and** ‹$P \Rightarrow\langle\alpha\rangle P'$› **and** *that* **show** *?thesis*
using *weak-transition-tau-iff weakly-bisimilar-tau-simulation-step* **by** *force*
next
case *False* **with** ‹$P \Rightarrow\langle\alpha\rangle P'$› **have** $P \Rightarrow\{\alpha\} P'$
by *simp*
then obtain *P1 P2* **where** *tauP*: $P \Rightarrow P1$ **and** *trans*: $P1 \rightarrow \langle\alpha,P2\rangle$ **and** *tauP2*: $P2 \Rightarrow P'$
using *observable-transition-def* **by** *blast*
from ‹$P \approx\cdot Q$› **and** *tauP* **obtain** *Q1* **where** *tauQ*: $Q \Rightarrow Q1$ **and** *P1Q1*: $P1 \approx\cdot Q1$
by (*metis weakly-bisimilar-tau-simulation-step*)

— rename $\langle\alpha,P2\rangle$ to avoid *Q1*, without touching Q
obtain p **where** *1*: $(p \cdot bn\ \alpha) \sharp* Q1$ **and** *2*: *supp* $(\langle\alpha,P2\rangle, Q) \sharp* p$
proof (*rule at-set-avoiding2*[*of bn* α *Q1* ($\langle\alpha,P2\rangle$, Q), *THEN exE*])
show *finite* (*bn* α) **by** (*fact bn-finite*)
next
show *finite* (*supp Q1*) **by** (*fact finite-supp*)
next
show *finite* (*supp* ($\langle\alpha,P2\rangle$, Q)) **by** (*simp add*: *finite-supp supp-Pair*)
next
show *bn* $\alpha \sharp* (\langle\alpha,P2\rangle, Q)$ **using** ‹*bn* $\alpha \sharp* Q$› **by** (*simp add*: *fresh-star-Pair*)
qed *metis*
from *2* **have** *3*: *supp* $\langle\alpha,P2\rangle \sharp* p$ **and** *4*: *supp* $Q \sharp* p$
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 ⇒ Q' and P'Q': (p • P') ≈· Q'
    by (metis weakly-bisimilar-tau-simulation-step)

  from tauQ and trans' and tauQ2 have Q ⇒⟨p • α⟩ Q'
    by (rule weak-transition-weakI)
  with 4 have Q ⇒⟨α⟩ (-p • Q')
    by (metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqvt)
  moreover from P'Q' have P' ≈· (-p • 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 ≈· Q and QR: Q ≈· 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 ≈· Q and Q ≈· R
        by blast
      then have R ≈· Q and Q ≈· P
        by (metis weakly-bisimilar-symp sympE)+
      then show ?bisim R P
        by blast
    qed
  moreover have ∀ P Q φ. ?bisim P Q ∧ P ⊢ φ ⟶ (∃ Q'. Q ⇒ Q' ∧ ?bisim
P Q' ∧ Q' ⊢ φ)
    proof (clarify)
      fix P Q φ R
      assume phi: P ⊢ φ and PR: P ≈· R and RQ: R ≈· Q
      from PR and phi obtain R' where R ⇒ R' and P ≈· R' and ∗: R' ⊢ φ
        using weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by
force
      from RQ and ‹R ⇒ R'› obtain Q' where Q ⇒ Q' and R' ≈· Q'
        by (metis weakly-bisimilar-tau-simulation-step)
      from ‹R' ≈· Q'› and ∗ obtain Q'' where Q' ⇒ Q'' and R' ≈· Q'' and
∗∗: Q'' ⊢ φ
        using weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by
force
      from ‹Q ⇒ Q'› and ‹Q' ⇒ Q''› have Q ⇒ Q''
        by (fact tau-transition-trans)
      moreover from ‹P ≈· R'› and ‹R' ≈· Q''› have ?bisim P Q''
        by blast
      ultimately show ∃ Q'. Q ⇒ Q' ∧ ?bisim P Q' ∧ Q' ⊢ φ
        using ∗∗ by metis
```

```
    qed
  moreover have ∀ P Q. ?bisim P Q ⟶ (∀ α P'. bn α ♯* Q ⟶ P → ⟨α,P'⟩
⟶ (∃ Q'. Q ⇒⟨α⟩ Q' ∧ ?bisim P' Q'))
    proof (clarify)
      fix P Q R α P'
      assume PR: P ≈· R and RQ: R ≈· Q and fresh: bn α ♯* Q and trans: P
→ ⟨α,P'⟩
      — rename ⟨α,P'⟩ to avoid R, without touching Q
      obtain p where 1: (p · bn α) ♯* R and 2: supp (⟨α,P'⟩, Q) ♯* p
        proof (rule at-set-avoiding2[of bn α R (⟨α,P'⟩, Q), THEN exE])
          show finite (bn α) by (fact bn-finite)
        next
          show finite (supp R) by (fact finite-supp)
        next
          show finite (supp (⟨α,P'⟩, Q)) by (simp add: finite-supp supp-Pair)
        next
          show bn α ♯* (⟨α,P'⟩, Q) by (simp add: fresh fresh-star-Pair)
        qed metis
      from 2 have 3: supp ⟨α,P'⟩ ♯* p and 4: supp Q ♯* p
        by (simp add: fresh-star-Un supp-Pair)+
      from 3 have ⟨p · α, p · P'⟩ = ⟨α,P'⟩
        using supp-perm-eq by fastforce
      with trans obtain pR' where 5: R ⇒⟨p · α⟩ pR' and 6: (p · P') ≈· pR'
            using PR 1 by (metis bn-eqvt weakly-bisimilar-is-weak-bisimulation
is-weak-bisimulation-def)
      from fresh and 4 have bn (p · α) ♯* Q
        by (metis bn-eqvt fresh-star-permute-iff supp-perm-eq)
      then obtain pQ' where 7: Q ⇒⟨p · α⟩ pQ' and 8: pR' ≈· pQ'
        using RQ 5 by (metis weakly-bisimilar-weak-simulation-step)
      from 7 have Q ⇒⟨α⟩ (−p · pQ')
     using 4 by (metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqvt)
      moreover from 6 and 8 have ?bisim P' (−p · pQ')
     by (metis (no-types, hide-lams) weakly-bisimilar-eqvt permute-minus-cancel(2)
relcompp.simps)
      ultimately show ∃ Q'. Q ⇒⟨α⟩ Q' ∧ ?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 ≈· R
    unfolding weakly-bisimilar-def by meson
qed

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

end
```

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

19 Weak Formulas

19.1 Disjunction

definition *Disj* :: ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula set*['*idx*] ⇒ ('*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*) ⊆ *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* · *Disj xset* = *Disj* (*p* · *xset*)
using *assms* **unfolding** *Disj-def* **by** *simp*

lemma *Disj-eq-iff* [*simp*]:
assumes *finite* (*supp xset1*) **and** *finite* (*supp xset2*)
shows *Disj xset1* = *Disj xset2* ⟷ *xset1* = *xset2*
using *assms* **unfolding** *Disj-def* **by** (*metis Conj-eq-iff Not-eq-iff bset.inj-map-strong finite-supp-map-bset-Not*)

19.2 Lemmas about α-equivalence involving τ

context *weak-nominal-ts*
begin

lemma *Act-tau-eq-iff* [*simp*]:
Act τ *x1* = *Act* α *x2* ⟷ α = τ ∧ *x2* = *x1*
(**is** *?l* ⟷ *?r*)

proof
assume *?l*
then obtain *p* **where** *p-α*: $p \cdot \tau = \alpha$ **and** *p-x*: $p \cdot x1 = x2$ **and** *fresh*: (*supp x1 − bn τ*) $\sharp*$ *p*
by (*metis Act-eq-iff-perm*)
from *p-α* **have** $\alpha = \tau$
by (*metis tau-eqvt*)
moreover from *fresh* **and** *p-x* **have** *x2 = x1*
by (*simp add*: *supp-perm-eq*)
ultimately show *?r* **..**
next
assume *?r* **then show** *?l*
by *simp*
qed

end

19.3 Weak action modality

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

Also, we use τ in our definition of weak formulas.

locale *indexed-weak-nominal-ts = weak-nominal-ts satisfies transition*
for *satisfies* :: *'state::fs ⇒ 'pred::fs ⇒ bool* (**infix** ⊢ *70*)
and *transition* :: *'state ⇒ ('act::bn,'state) residual ⇒ bool* (**infix** → *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*| ≤*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 ⇒ nat ⇒* (*'idx,'pred,'act*) *formula*
where
tau-steps x 0 = *x*
| *tau-steps x* (*Suc n*) = *Act τ* (*tau-steps x n*)

lemma *tau-steps-eqvt* [*eqvt*]:

```
    p · tau-steps x n = tau-steps (p · x) (p · n)
    by (induct n) (simp-all add: permute-nat-def tau-eqvt)

  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 ⟷ n = 0
  proof
    assume x = tau-steps x n then show n = 0
      proof (induct n arbitrary: x)
        case 0 show ?case ..
      next
        case (Suc n)
        then have x = Act τ (tau-steps x n)
          by simp
        then show Suc n = 0
          proof (induct x)
            case (Act α x)
            then have x = tau-steps (Act τ 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 = tau-steps x n
      by simp
  qed

  definition weak-tau-modality :: ('idx,'pred::fs,'act::bn) formula ⇒ ('idx,'pred,'act)
formula
    where
      weak-tau-modality x ≡ Disj (map-bset (tau-steps x) (Abs-bset UNIV))

  lemma finite-supp-map-bset-tau-steps [simp]:
    finite (supp (map-bset (tau-steps x) (Abs-bset UNIV :: nat set['idx])))
  proof −
    have eqvt map-bset and eqvt tau-steps
      by (simp add: eqvtI)+
    then have supp (map-bset (tau-steps x)) ⊆ supp x
      using supp-fun-eqvt supp-fun-app supp-fun-app-eqvt by blast
    moreover have supp (Abs-bset UNIV :: nat set['idx]) = {}
      by (simp add: eqvtI supp-fun-eqvt)
   ultimately have supp (map-bset (tau-steps x) (Abs-bset UNIV :: nat set['idx]))
⊆ supp x
      using supp-fun-app by blast
    then show ?thesis
```

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

lemma *weak-tau-modality-eqvt* [*eqvt*]:
p ∙ *weak-tau-modality x* = *weak-tau-modality* (*p* ∙ *x*)
unfolding *weak-tau-modality-def* **by** (*simp add*: *map-bset-eqvt*)

lemma *weak-tau-modality-eq-iff* [*simp*]:
weak-tau-modality x = *weak-tau-modality y* ⟷ *x* = *y*
proof
assume *weak-tau-modality x* = *weak-tau-modality y*
then have *map-bset* (*tau-steps x*) (*Abs-bset UNIV* :: - *set*['*idx*]) = *map-bset* (*tau-steps y*) (*Abs-bset UNIV*)
unfolding *weak-tau-modality-def* **by** *simp*
with *card-idx-nat*′ **have** *range* (*tau-steps x*) = *range* (*tau-steps y*)
(**is** *?X* = *?Y*)
by (*metis Abs-bset-inverse*′ *map-bset.rep-eq*)
then have *x* ∈ *range* (*tau-steps y*) **and** *y* ∈ *range* (*tau-steps x*)
by (*metis range-eqI tau-steps.simps*(*1*))+
then obtain *nx ny* **where** *x*: *x* = *tau-steps y nx* **and** *y*: *y* = *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* = *weak-tau-modality y*
by *simp*
qed

lemma *supp-weak-tau-modality* [*simp*]:
supp (*weak-tau-modality x*) = *supp x*
unfolding *supp-def* **by** *simp*

lemma *Act-weak-tau-modality-eq-iff* [*simp*]:
Act α1 (*weak-tau-modality x1*) = *Act α2* (*weak-tau-modality x2*) ⟷ *Act α1 x1* = *Act α2 x2*
by (*simp add*: *Act-eq-iff-perm*)

definition *weak-action-modality* :: '*act* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* ⇒ ('*idx*,'*pred*,'*act*) *formula* (⟨⟨-⟩⟩-)
where
⟨⟨*α*⟩⟩*x* ≡ *if α* = *τ then weak-tau-modality x else weak-tau-modality* (*Act α* (*weak-tau-modality x*))

lemma *weak-action-modality-eqvt* [*eqvt*]:
p ∙ (⟨⟨*α*⟩⟩*x*) = ⟨⟨*p* ∙ *α*⟩⟩(*p* ∙ *x*)
by (*simp add*: *weak-action-modality-def*)

lemma *weak-action-modality-tau*:
$(\langle\langle\tau\rangle\rangle x) = \textit{weak-tau-modality}\ x$
unfolding *weak-action-modality-def* **by** *simp*

lemma *weak-action-modality-not-tau*:
assumes $\alpha \neq \tau$
shows $(\langle\langle\alpha\rangle\rangle x) = \textit{weak-tau-modality}\ (\textit{Act}\ \alpha\ (\textit{weak-tau-modality}\ x))$
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 $\langle\langle\tau\rangle\rangle \textit{Act}\ \alpha\ (\textit{weak-tau-modality}\ x) = \langle\langle\alpha\rangle\rangle x$ by definition, but clearly not $\textit{Act}\ \tau\ (\textit{Act}\ \alpha\ (\textit{weak-tau-modality}\ x)) = \textit{Act}\ \alpha\ x$.

lemma *weak-action-modality-eq*:
assumes *Act α1 x1* = *Act α2 x2*
shows $(\langle\langle\alpha 1\rangle\rangle x1) = (\langle\langle\alpha 2\rangle\rangle x2)$
proof (*cases* $\alpha 1 = \tau$)
case *True*
with *assms* **have** $\alpha 2 = \alpha 1 \wedge x2 = x1$
by (*metis Act-tau-eq-iff*)
then show *?thesis*
by *simp*
next
case *False*
from *assms* **obtain** *p* **where** *1*: *supp x1* − *bn α1* = *supp x2* − *bn α2* **and** *2*: (*supp x1* − *bn α1*) ♯* *p*
and *3*: *p* ∙ *x1* = *x2* **and** *4*: *supp α1* − *bn α1* = *supp α2* − *bn α2* **and** *5*: (*supp α1* − *bn α1*) ♯* *p*
and *6*: *p* ∙ *α1* = *α2*
by (*metis Act-eq-iff-perm*)
from *1* **have** *supp* (*weak-tau-modality x1*) − *bn α1* = *supp* (*weak-tau-modality x2*) − *bn α2*
by (*metis supp-weak-tau-modality*)
moreover from *2* **have** (*supp* (*weak-tau-modality x1*) − *bn α1*) ♯* *p*
by (*metis supp-weak-tau-modality*)
moreover from *3* **have** *p* ∙ *weak-tau-modality x1* = *weak-tau-modality x2*
by (*metis weak-tau-modality-eqvt*)
ultimately have *Act α1* (*weak-tau-modality x1*) = *Act α2* (*weak-tau-modality x2*)
using *4* **and** *5* **and** *6* **and** *Act-eq-iff-perm* **by** *blast*
moreover from ‹*α1* ≠ *τ*› **and** *assms* **have** *α2* ≠ *τ*
by (*metis Act-tau-eq-iff*)
ultimately show *?thesis*
using ‹*α1* ≠ *τ*› **by** (*simp add*: *weak-action-modality-not-tau*)
qed

19.4 Weak formulas

inductive *weak-formula* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*) *formula* ⇒ *bool*

where
wf-Conj: *finite* (*supp xset*) $\Longrightarrow$ ($\bigwedge x.\ x \in$ *set-bset xset* $\Longrightarrow$ *weak-formula x*) $\Longrightarrow$ *weak-formula* (*Conj xset*)
| *wf-Not*: *weak-formula x* $\Longrightarrow$ *weak-formula* (*Not x*)
| *wf-Act*: *weak-formula x* $\Longrightarrow$ *weak-formula* ($\langle\langle\alpha\rangle\rangle x$)
| *wf-Pred*: *weak-formula x* $\Longrightarrow$ *weak-formula* ($\langle\langle\tau\rangle\rangle$(*Conj* (*binsert* (*Pred* φ) (*bsingleton x*))))

lemma *finite-supp-wf-Pred* [*simp*]: *finite* (*supp* (*binsert* (*Pred* φ) (*bsingleton x*)))
proof −
have *supp* (*bsingleton x*) $\subseteq$ *supp x*
by (*simp add*: *eqvtI supp-fun-app-eqvt*)
moreover have *eqvt binsert*
by (*simp add*: *eqvtI*)
ultimately have *supp* (*binsert* (*Pred* φ) (*bsingleton x*)) $\subseteq$ *supp* $\varphi \cup$ *supp x*
using *supp-fun-app supp-fun-app-eqvt* **by** *fastforce*
then show *?thesis*
by (*metis finite-UnI finite-supp rev-finite-subset*)
qed

weak-formula is equivariant.

lemma *weak-formula-eqvt* [*eqvt*]: *weak-formula x* $\Longrightarrow$ *weak-formula* ($p \cdot x$)
proof (*induct rule*: *weak-formula.induct*)
case (*wf-Conj xset*) **then show** *?case*
by *simp* (*metis* (*no-types*, *lifting*) *imageE permute-finite permute-set-eq-image set-bset-eqvt supp-eqvt weak-formula.wf-Conj*)
next
case (*wf-Not x*) **then show** *?case*
by (*simp add*: *weak-formula.wf-Not*)
next
case (*wf-Act x* α) **then show** *?case*
by (*simp add*: *weak-formula.wf-Act*)
next
case (*wf-Pred x* φ) **then show** *?case*
by (*simp add*: *tau-eqvt weak-action-modality-eqvt weak-formula.wf-Pred*)
qed

end

end
theory *Weak-Validity*
imports
Weak-Formula
Validity
begin

20 Weak Validity

Weak formulas are a subset of (strong) formulas, and the definition of validity is simply taken from the latter. Here we prove some useful lemmas about the validity of weak modalities. These are similar to corresponding lemmas about the validity of the (strong) action modality.

context *nominal-ts*
begin

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

end

context *indexed-weak-nominal-ts*
begin

lemma *valid-weak-tau-modality-iff-tau-steps*:
$P \models \mathit{weak\text{-}tau\text{-}modality}\ x \longleftrightarrow (\exists\, n.\ P \models \mathit{tau\text{-}steps}\ x\ n)$
unfolding *weak-tau-modality-def* **by** (*auto simp add*: *map-bset.rep-eq*)

lemma *tau-steps-iff-tau-transition*:
$(\exists\, n.\ P \models \mathit{tau\text{-}steps}\ x\ n) \longleftrightarrow (\exists\, P'.\ P \Rightarrow P' \wedge P' \models x)$
proof
assume $\exists\, n.\ P \models \mathit{tau\text{-}steps}\ x\ n$
then obtain n **where** $P \models \mathit{tau\text{-}steps}\ x\ n$
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** $P' \models \mathit{tau\text{-}steps}\ x\ n$
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 $\exists\, n.\ P \models \mathit{tau\text{-}steps}\ x\ n$
proof (*induct*)
case (*tau-refl P*) **then have** $P \models \mathit{tau\text{-}steps}\ x\ 0$

```
      by simp
    then show ?case
      by meson
  next
    case (tau-step P P' P'')
    then obtain n where P' ⊨ tau-steps x n
      by meson
    with ‹P → ⟨τ,P'⟩› have P ⊨ tau-steps x (Suc n)
      by (auto simp add: valid-Act-fresh[OF bn-tau-fresh])
    then show ?case
      by meson
  qed
qed

lemma valid-weak-tau-modality:
  P ⊨ weak-tau-modality x ⟷ (∃ P'. P ⇒ P' ∧ P' ⊨ x)
by (metis valid-weak-tau-modality-iff-tau-steps tau-steps-iff-tau-transition)

lemma valid-weak-action-modality:
  P ⊨ (⟨⟨α⟩⟩x) ⟷ (∃ α' x' P'. Act α x = Act α' x' ∧ P ⇒⟨α'⟩ P' ∧ P' ⊨ x')
  (is ?l ⟷ ?r)
proof (cases α = τ)
  case True show ?thesis
  proof
    assume ?l
    with ‹α = τ› obtain P' where trans: P ⇒ P' and valid: P' ⊨ x
      by (metis valid-weak-tau-modality weak-action-modality-tau)
    from trans have P ⇒⟨τ⟩ P'
      by simp
    with ‹α = τ› and valid show ?r
      by blast
  next
    assume ?r
    then obtain α' x' P' where eq: Act α x = Act α' x' and trans: P ⇒⟨α'⟩
P' and valid: P' ⊨ x'
      by blast
    from eq have α' = τ ∧ x' = x
      using ‹α = τ› by simp
    with trans and valid have P ⇒ P' and P' ⊨ x
      by simp+
    with ‹α = τ› show ?l
      by (metis valid-weak-tau-modality weak-action-modality-tau)
  qed
next
  case False show ?thesis
  proof
    assume ?l
      with ‹α ≠ τ› obtain Q where trans: P ⇒ Q and valid: Q ⊨ Act α
(weak-tau-modality x)
```

by (*metis valid-weak-tau-modality weak-action-modality-not-tau*)
from *valid* **obtain** α' x' Q' **where** *eq*: $Act\ \alpha\ (weak\text{-}tau\text{-}modality\ x) = Act\ \alpha'\ x'$ **and** *trans'*: $Q \rightarrow \langle \alpha', Q' \rangle$ **and** *valid'*: $Q' \models x'$
by (*metis valid-Act*)

from *eq* **obtain** p **where** *p-α*: $\alpha' = p \cdot \alpha$ **and** *p-x*: $x' = p \cdot weak\text{-}tau\text{-}modality\ x$
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** $Q' \models weak\text{-}tau\text{-}modality\ (p \cdot x)$
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 α' 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 ‹$\alpha \neq \tau$› **have** α': $\alpha' \neq \tau$
using *eq* **by** (*metis Act-tau-eq-iff*)
with *trans* **obtain** Q Q' **where** *trans'*: $P \Rightarrow Q$ **and** *trans''*: $Q \rightarrow \langle \alpha', Q' \rangle$ **and** *trans'''*: $Q' \Rightarrow P'$
by (*meson observable-transition-def weak-transition-def*)
from *trans'''* **and** *valid* **have** $Q' \models weak\text{-}tau\text{-}modality\ x'$
by (*metis valid-weak-tau-modality*)
with *trans''* **have** $Q \models Act\ \alpha'\ (weak\text{-}tau\text{-}modality\ x')$
by (*metis valid-Act*)
with *trans'* **and** α' **have** $P \models \langle \langle \alpha' \rangle \rangle x'$
by (*metis valid-weak-tau-modality weak-action-modality-not-tau*)
moreover from *eq* **have** $(\langle \langle \alpha \rangle \rangle x) = (\langle \langle \alpha' \rangle \rangle x')$
by (*metis weak-action-modality-eq*)
ultimately show *?l*
by *simp*
qed
qed

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

lemma *valid-weak-action-modality-strong*:
assumes *finite* (*supp X*)
shows $P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \Rightarrow \langle \alpha' \rangle\ P' \wedge P' \models x' \wedge bn\ \alpha' \sharp* X)$
proof

```
    assume P ⊨ ⟨⟨α⟩⟩x
    then obtain α' x' P' where eq: Act α x = Act α' x' and trans: P ⇒⟨α'⟩ P'
and valid: P' ⊨ x'
      by (metis valid-weak-action-modality)
    show ∃ α' x' P'. Act α x = Act α' x' ∧ P ⇒⟨α'⟩ P' ∧ P' ⊨ x' ∧ bn α' ♯* X
      proof (cases α' = τ)
        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 ⇒ Q and trans'': Q → ⟨α', Q'⟩
and trans''': Q' ⇒ P'
          by (metis weak-transition-def observable-transition-def)
        have finite (bn α')
          by (fact bn-finite)
        moreover note ‹finite (supp X)›
        moreover have finite (supp (Act α' x', ⟨α',Q'⟩))
        by (metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair)
        moreover have bn α' ♯* (Act α' x', ⟨α',Q'⟩)
        by (auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair)
         ultimately obtain p where fresh-X: (p • bn α') ♯* X and supp (Act α'
x', ⟨α',Q'⟩) ♯* p
          by (metis at-set-avoiding2)
        then have supp (Act α' x') ♯* p and supp ⟨α',Q'⟩ ♯* p
          by (metis fresh-star-Un supp-Pair)+
         then have 1: Act (p • α') (p • x') = Act α' x' and 2: ⟨p • α', p • Q'⟩ =
⟨α',Q'⟩
          by (metis Act-eqvt supp-perm-eq, metis abs-residual-pair-eqvt supp-perm-eq)
        from trans' and trans'' and trans''' have P ⇒⟨p • α'⟩ (p • 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 ∃ α' x' P'. Act α x = Act α' x' ∧ P ⇒⟨α'⟩ P' ∧ P' ⊨ x' ∧ bn α' ♯*
X
    then show P ⊨ ⟨⟨α⟩⟩x
      by (metis valid-weak-action-modality)
  qed

  lemma valid-weak-action-modality-fresh:
    assumes bn α ♯* P
    shows P ⊨ (⟨⟨α⟩⟩x) ⟷ (∃ P'. P ⇒⟨α⟩ P' ∧ P' ⊨ x)
  proof
    assume P ⊨ ⟨⟨α⟩⟩x

    moreover have finite (supp P)
      by (fact finite-supp)
```

ultimately obtain $\alpha'\ x'\ P'$ **where**
eq: $Act\ \alpha\ x = Act\ \alpha'\ x'$ **and** *trans*: $P \Rightarrow\langle\alpha'\rangle\ P'$ **and** *valid*: $P' \models x'$ **and** *fresh*: $bn\ \alpha' \sharp* P$
by (*metis valid-weak-action-modality-strong*)

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

from *assms* **and** *fresh* **have** $(bn\ \alpha \cup p \cdot bn\ \alpha) \sharp* P$
using *p-α* **by** (*metis bn-eqvt fresh-star-Un*)
then have $supp\ p \sharp* P$
using *supp-p* **by** (*metis fresh-star-def subset-eq*)
then have *p-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-P p-α* **by** (*metis permute-minus-cancel(1) weak-transition-eqvt*)
moreover from *valid* **have** $-p \cdot P' \models x$
using *p-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

21 Weak Logical Equivalence

context *indexed-weak-nominal-ts*
begin

Two states are weakly logically equivalent if they validate the same weak formulas.

definition *weakly-logically-equivalent* :: *'state* ⇒ *'state* ⇒ *bool* **where**
weakly-logically-equivalent P Q ≡ (∀ *x*::(*'idx*,*'pred*,*'act*) *formula*. *weak-formula x* ⟶ *P* ⊨ *x* ⟷ *Q* ⊨ *x*)

notation *weakly-logically-equivalent* (**infix** ≡· *50*)

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

end

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

22 Weak Bisimilarity Implies Weak Logical Equivalence

context *indexed-weak-nominal-ts*
begin

lemma *weak-bisimilarity-implies-weak-equivalence-Act*:
assumes $\bigwedge P\ Q.\ P \approx\cdot Q \Longrightarrow P \models x \longleftrightarrow Q \models x$
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 ‹$P \models \langle\langle\alpha\rangle\rangle x$› **obtain** $\alpha'\ x'\ P'$ **where** *eq*: $Act\ \alpha\ x = Act\ \alpha'\ x'$ **and** *trans*: $P \Rightarrow\langle\alpha'\rangle\ P'$ **and** *valid*: $P' \models x'$ **and** *fresh*: *bn* $\alpha' \sharp* Q$
by (*metis valid-weak-action-modality-strong*)

from ‹$P \approx\cdot Q$› **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 ‹$\bigwedge P\ Q.\ P \approx\cdot Q \Longrightarrow P \models x \longleftrightarrow Q \models x$› **by** *metis*
with *px* **have** $Q' \models x'$
by (*metis permute-minus-cancel*(*1*) *valid-eqvt*)

with *eq* **and** *trans'* **show** $Q \models \langle\langle\alpha\rangle\rangle x$
unfolding *valid-weak-action-modality* **by** *metis*
qed

lemma *weak-bisimilarity-implies-weak-equivalence-Pred*:
assumes $\bigwedge P\ Q.\ P \approx\cdot Q \Longrightarrow P \models x \longleftrightarrow Q \models x$
and $P \approx\cdot Q$
— not needed: and *weak-formula x*
and $P \models \langle\langle\tau\rangle\rangle(Conj\ (binsert\ (Pred\ \varphi)\ (bsingleton\ x)))$
shows $Q \models \langle\langle\tau\rangle\rangle(Conj\ (binsert\ (Pred\ \varphi)\ (bsingleton\ x)))$
proof −
let *?c* = $Conj\ (binsert\ (Pred\ \varphi)\ (bsingleton\ x))$

from ‹$P \models \langle\langle\tau\rangle\rangle$ *?c*› **obtain** P' **where** *trans*: $P \Rightarrow P'$ **and** *valid*: $P' \models$ *?c*
using *valid-weak-action-modality* **by** *auto*

have $bn\ \tau\ \sharp*\ Q$
by (*simp add*: *fresh-star-def*)
with ‹$P \approx\cdot Q$› **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 ‹$\bigwedge P\ Q.\ P \approx\cdot Q \Longrightarrow P \models x \longleftrightarrow Q \models x$› **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

```
≡· Q
  proof −
  {
    fix x :: ('idx, 'pred, 'act) formula
    assume weak-formula x
      then have ⋀P Q. P ≈· Q ⟹ P ⊨ x ⟷ Q ⊨ 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 α) then show ?case
      by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Act
sympE)
    next
      case (wf-Pred x φ) then show ?case
      by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Pred
sympE)
      qed
    }
    with assms show ?thesis
      unfolding weakly-logically-equivalent-def by simp
  qed

end

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

23 Weak Logical Equivalence Implies Weak Bisimilarity

```
context indexed-weak-nominal-ts
begin

  definition distinguishing-formula :: ('idx, 'pred, 'act) formula ⇒ 'state ⇒ 'state
⇒ bool
    (- distinguishes - from - [100,100,100] 100)
  where
    x distinguishes P from Q ≡ P ⊨ x ∧ ¬ Q ⊨ x

  lemma distinguishing-formula-eqvt [eqvt]:
    assumes x distinguishes P from Q shows (p · x) distinguishes (p · P) from
```

$(p \cdot Q)$
using *assms* **unfolding** *distinguishing-formula-def*
by (*metis permute-minus-cancel*(*2*) *valid-eqvt*)

lemma *weakly-equivalent-iff-not-distinguished*: $(P \equiv\cdot Q) \longleftrightarrow \neg(\exists\, x.$ *weak-formula* $x \wedge x$ *distinguishes* P *from* $Q)$
by (*meson distinguishing-formula-def weakly-logically-equivalent-def valid-Not wf-Not*)

There exists a distinguishing weak formula for P and Q whose support is contained in *supp* P.

lemma *distinguished-bounded-support*:
assumes *weak-formula* x **and** x *distinguishes* P *from* Q
obtains y **where** *weak-formula* y **and** *supp* $y \subseteq$ *supp* P **and** y *distinguishes* P *from* Q
proof −
let *?B* $= \{p \cdot x | p.$ *supp* $P \sharp* p\}$
have *supp* P *supports ?B*
unfolding *supports-def* **proof** (*clarify*)
fix a b
assume a: $a \notin$ *supp* P **and** b: $b \notin$ *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*: *supp* $P \sharp* p$
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** *supp* $P \sharp*$ *?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*: *supp* $P \sharp* p$
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** *supp* $P \sharp*$ *?q*
by (*metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps*(*3*))
ultimately show $x' \in (a \rightleftharpoons b) \cdot$ *?B*
using *mem-permute-iff* **by** *blast*
qed
ultimately show $(a \rightleftharpoons b) \cdot$ *?B* $=$ *?B* **..**
qed

```
    then have supp-B-subset-supp-P: supp ?B ⊆ 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 • x
        using Abs-bset-inverse mem-Collect-eq by auto
      then show weak-formula x'
        using ‹weak-formula x› by (metis weak-formula-eqvt)
    qed
     moreover from finite-supp-B and card-B and supp-B-subset-supp-P have
supp ?y ⊆ supp P
      by simp
    moreover have ?y distinguishes P from Q
      unfolding distinguishing-formula-def proof
        from assms show P ⊨ ?y
         by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
supp-perm-eq valid-eqvt)
      next
        from assms show ¬ Q ⊨ ?y
         by (auto simp add: card-B finite-supp-B) (metis distinguishing-formula-def
permute-zero fresh-star-zero)
      qed
    ultimately show ?thesis ..
  qed

 lemma 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 φ assume P ≡· Q and P ⊢ φ
      then have ∃ Q'. Q ⇒ Q' ∧ P ≡· Q' ∧ Q' ⊢ φ
```

```
    proof −
      {
        let ?Q′ = {Q′. Q ⇒ Q′ ∧ Q′ ⊢ φ}
        assume ∀ Q′∈?Q′. ¬ P ≡· Q′
         then have ∀ Q′∈?Q′. ∃ x :: ('idx, 'pred, 'act) formula. weak-formula x
∧ x distinguishes P from Q′
          by (metis weakly-equivalent-iff-not-distinguished)
         then have ∀ Q′∈?Q′. ∃ x :: ('idx, 'pred, 'act) formula. weak-formula x
∧ supp x ⊆ supp P ∧ x distinguishes P from Q′
          by (metis distinguished-bounded-support)
        then obtain f :: 'state ⇒ ('idx, 'pred, 'act) formula where
           ∗: ∀ Q′∈?Q′. weak-formula (f Q′) ∧ supp (f Q′) ⊆ supp P ∧ (f Q′)
distinguishes P from Q′
          by metis
        have supp P supports (f ‘ ?Q′)
          unfolding supports-def proof (clarify)
            fix a b
            assume a: a ∉ supp P and b: b ∉ supp P
            have (a ⇌ b) · (f ‘ ?Q′) ⊆ f ‘ ?Q′
            proof
              fix x
              assume x ∈ (a ⇌ b) · (f ‘ ?Q′)
              then have (a ⇌ b) · x ∈ f ‘ ?Q′
                by (metis (full-types) mem-permute-iff permute-swap-cancel)
               then obtain Q′ where 1: (a ⇌ b) · x = f Q′ and 2: Q ⇒ Q′ ∧
Q′ ⊢ φ
                by auto
              with ∗ and a and b have a ∉ supp (f Q′) and b ∉ supp (f Q′)
                by auto
              with 1 have x = f Q′
                by (metis permute-swap-cancel supp-supports supports-def)
              with 2 show x ∈ f ‘ ?Q′
                by simp
            qed
            moreover have f ‘ ?Q′ ⊆ (a ⇌ b) · (f ‘ ?Q′)
            proof
              fix x
              assume x ∈ f ‘ ?Q′
              then obtain Q′ where 1: x = f Q′ and 2: Q ⇒ Q′ ∧ Q′ ⊢ φ
                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*|
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 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 distinguishing-formula-def by blast
        qed
       ultimately have False
        by (metis ‹P ≡· Q› weakly-logically-equivalent-def)
      }
      then show ?thesis
       by blast
    qed
  }
  moreover — weak simulation
  {
   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. weak-formula x
∧ x distinguishes P' from Q'
        by (metis weakly-equivalent-iff-not-distinguished)
       then have ∀ Q'∈?Q'. ∃ x :: ('idx, 'pred, 'act) formula. weak-formula x
∧ supp x ⊆ supp P' ∧ x distinguishes P' from Q'
        by (metis distinguished-bounded-support)
      then obtain f :: 'state ⇒ ('idx, 'pred, 'act) formula where
        ∗: ∀ Q'∈?Q'. weak-formula (f Q') ∧ supp (f Q') ⊆ supp P' ∧ (f Q')
distinguishes P' from Q'
        by metis
      have supp P' supports (f ‘ ?Q')
        unfolding supports-def proof (clarify)
         fix a b
         assume a: a ♯ supp P' and b: b ♯ supp P'
         have (a ⇌ b) • (f ‘ ?Q') ⊆ f ‘ ?Q'
         proof
          fix x
          assume x ∈ (a ⇌ b) • (f ‘ ?Q')
          then obtain Q' where 1: x = (a ⇌ b) • f Q' and 2: Q ⇒⟨α⟩ Q'
          by auto (metis (no-types, lifting) imageE image-eqvt mem-Collect-eq
permute-set-eq-image)
          with ∗ and a and b have a ♯ supp (f Q') and b ♯ 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 ∈ f ‘ ?Q'
```

```
      by simp
    qed
    moreover have f ‘ ?Q′ ⊆ (a ⇌ b) • (f ‘ ?Q′)
    proof
      fix x
      assume x ∈ f ‘ ?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 distinguishing-formula-def mem-Collect-eq)
            }
            then show P' ⊨ ?y
              by (simp add: finite-supp-image card-image)
          qed
        moreover have ¬ Q ⊨ ⟨⟨α⟩⟩?y
          proof
            assume Q ⊨ ⟨⟨α⟩⟩?y
            then obtain Q' where 1: Q ⇒⟨α⟩ Q' and 2: Q' ⊨ ?y
              using ‹bn α ♯∗ Q› by (metis valid-weak-action-modality-fresh)
            from 2 have ⋀Q''. Q ⇒⟨α⟩ Q'' ⟶ Q' ⊨ f Q''
              by (simp add: finite-supp-image card-image)
            with 1 and ∗ show False
              using distinguishing-formula-def by blast
          qed
        ultimately have False
          by (metis ‹P ≡· Q› 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 ≡· Q shows P
≈· Q
using assms by (metis weakly-bisimilar-def weak-equivalence-is-weak-bisimulation)

end

end
```

References

[1] J. Parrow, J. Borgström, L. Eriksson, R. Gutkovas, and T. Weber. Modal logics for nominal transition systems. In L. Aceto and D. de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPIcs*, pages 198–211. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.