Modal Logics for Nominal Transition Systems

Tjark Weber et al.

April 20, 2020

Abstract

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

Contents

1 Bounded Sets Equipped With a Permutation Action ................................................................. 4

2 Lemmas about Well-Foundedness and Permutations ............................................................... 5
   2.1 Hull and well-foundedness ................................................................. 5

3 Residuals ....................................................................................................................................... 7
   3.1 Binding names .............................................................................. 7
   3.2 Raw residuals and α-equivalence .................................................. 7
   3.3 Residuals ...................................................................................... 8
   3.4 Notation for pairs as residuals ...................................................... 9
   3.5 Support of residuals ..................................................................... 9
   3.6 Equality between residuals ........................................................ 9
   3.7 Strong induction ......................................................................... 10
   3.8 Other lemmas .............................................................................. 11

4 Nominal Transition Systems and Bisimulations ........................................................................ 12
   4.1 Basic Lemmas ............................................................................. 12
   4.2 Nominal transition systems ........................................................ 12
   4.3 Bisimulations ............................................................................. 12

5 Infinitary Formulas .................................................................................................................... 16
   5.1 Infinitely branching trees ............................................................ 16
   5.2 Trees modulo α-equivalence ......................................................... 19
   5.3 Constructors for trees modulo α-equivalence ............................... 33
   5.4 Induction over trees modulo α-equivalence .................................. 36
   5.5 Hereditarily finitely supported trees .......................................... 36
5.6 Infinitary formulas ........................................... 38
5.7 Constructors for infinitary formulas ....................... 40
5.8 Induction over infinitary formulas ........................ 44
5.9 Strong induction over infinitary formulas ............... 45

6 Validity .................................................. 46
6.1 Validity for infinitely branching trees .................... 48
6.2 Validity for trees modulo $\alpha$-equivalence ............ 51
6.3 Validity for infinitary formulas .......................... 52

7 (Strong) Logical Equivalence ............................... 54

8 Bisimilarity Implies Logical Equivalence .................. 55

9 Logical Equivalence Implies Bisimilarity .................. 56

10 Disjunction ............................................. 60

11 Expressive Completeness .................................. 61
11.1 Distinguishing formulas .................................. 61
11.2 Characteristic formulas .................................. 65
11.3 Expressive completeness .................................. 68

12 Finitely Supported Sets .................................... 69

13 Nominal Transition Systems with Effects and $F/L$-Bisimilarity 70
13.1 Nominal transition systems with effects .................. 70
13.2 $L$-bisimulations and $F/L$-bisimilarity ................. 71

14 Infinitary Formulas With Effects .......................... 76
14.1 Infinitely branching trees .................................. 76
14.2 Trees modulo $\alpha$-equivalence .......................... 78
14.3 Constructors for trees modulo $\alpha$-equivalence .......... 93
14.4 Induction over trees modulo $\alpha$-equivalence ........... 96
14.5 Hereditarily finitely supported trees ...................... 97
14.6 Infinitary formulas ...................................... 98
14.7 Constructors for infinitary formulas ..................... 100
14.8 $F/L$-formulas .......................................... 104
14.9 Induction over infinitary formulas ....................... 105
14.10 Strong induction over infinitary formulas ............... 105

15 Validity With Effects ...................................... 105
15.1 Validity for infinitely branching trees ................. 107
15.2 Validity for trees modulo $\alpha$-equivalence ............. 110
15.3 Validity for infinitary formulas ......................... 111
theory Nominal-Bounded-Set
imports
  Nominal2
begin

1 Bounded Sets Equipped With a Permutation Action

Additional lemmas about bounded sets.

interpretation bset-lifting

lemma Abs-bset-inverse [simp]:
  assumes \(|A| < \circ \text{natLeq} + c \mid \text{UNIV} :: 'k set\)
  shows \((\text{set-bset} \ (\text{Abs-bset} A :: 'a set[\{k\}])) = A\)
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 :: \(\text{perm} \Rightarrow 'a \text{set} \Rightarrow 'a \text{set}\)

proof
  fix \(p\) and \(A :: 'a \text{set}\)
  have \(|p \cdot A| \leq |A|\) by (simp add: permute-set-eq-image)
  also assume \(|A| < \circ \text{natLeq} + c \mid \text{UNIV} :: 'b set\)
  finally show \(|p \cdot A| < \circ \text{natLeq} + c \mid \text{UNIV} :: 'b set\).
qed

instance
by standard (transfer, simp)+

end

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

lemma supp-Abs-bset [simp]:
  assumes \(|A| < \circ \text{natLeq} + c \mid \text{UNIV} :: 'k set\)
  shows \((\text{supp} \ (\text{Abs-bset} A :: 'a::pt set[\{k\}])) = \text{supp} A\)
proof
  from assms have \(\bigwedge p. \ (\text{Abs-bset} A :: 'a::pt set[\{k\}]) \neq \text{Abs-bset} A \iff p \cdot A\)
≠ A
then show ?thesis
  unfolding supp-def by simp
qed

lemma map-bset-permute: p · B = map-bset (permute p) B
by transfer (auto simp add: image-def permute-set-def)

lemma set-bset-eqvt [eqvt]:
  p · set-bset B = set-bset (p · B)
by transfer simp

lemma map-bset-eqvt [eqvt]:
  p · map-bset f B = map-bset (p · f) (p · B)
by transfer simp

lemma bempty-eqvt [eqvt]: p · bempty = bempty
by transfer simp

lemma binsert-eqvt [eqvt]: p · (binsert x B) = binsert (p · x) (p · B)
by transfer simp

lemma bsingleton-eqvt [eqvt]: p · bsingleton x = bsingleton (p · 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 ≡ {(x, y). x < y}

lemma less-bool-rel-iff [simp]:
  (a, b) ∈ less-bool-rel ↔ ¬ a ∧ 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 wfUNIV1)

2.1 Hull and well-foundedness

inductive-set hull-rel where
\[(p \cdot x, x) \in \text{hull-rel}\]

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

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

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

**lemma** hull-rel-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  
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 \((x, y) \in R\)  
solves \((p \cdot x, y) \in\) hull-rel O R  
using assms by (metis hull-rel.relcompI relcompI)

**lemma** hull-rel-relcomp-trivialI [simp]:  
assumes \((x, y) \in R\)  
solves \((x, y) \in\) hull-rel O R  
using assms by (metis hull-rel-relcompI permute-zero)
theory Residual
imports
  Nominal2
begin

3 Residuals

3.1 Binding names

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

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

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

3.2 Raw residuals and $\alpha$-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 $\langle \alpha_1, P_1 \rangle \langle \alpha_2, P_2 \rangle \leftrightarrow [bn \alpha_1]set. (\alpha_1, P_1) = [bn \alpha_2]set. (\alpha_2, P_2)$

$\alpha$-equivalence is equivariant.

lemma alpha-residual-eqvt [$eqvt$]:
  assumes alpha-residual $r_1$ $r_2$
  shows alpha-residual $(p \cdot r_1) (p \cdot r_2)$
using assms by (cases $r_1$, cases $r_2$) (simp, metis Pair-eqvt $bn$-eqvt permute-Abs-set)

$\alpha$-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 $\alpha$-equivalence.

quotient-type

$\langle \text{act}, \text{state} \rangle$ residual = $\langle \text{act::bn} \times \text{state::pt} \rangle /$ alpha-residual
by (fact alpha-residual-equivp)

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

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

The permutation operation is lifted from raw residuals.

instantiation residual :: (bn,pt) pt
begin

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

instance
proof
  fix res :: (\_,\_) residual
  show $\theta \cdot res = res$
    by transfer (metis alpha-residual-equivp equivp-reflp permute-zero)
next
  fix p q :: perm and res :: (\_,\_) residual
  show $(p + q) \cdot res = p \cdot q \cdot res$
    by transfer (metis alpha-residual-equivp equivp-reflp permute-plus)
qed

end

The abstraction function from raw residuals to residuals is equivariant. The representation function is equivariant modulo $\alpha$-equivalence.

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

lemma alpha-residual-permute-rep-commute [simp]: alpha-residual (p $\cdot$ rep-residual res) (rep-residual (p $\cdot$ res))
by (metis residual.abs-eq-iff residual-abs-rep permute-residual.abs-eq)
3.4 Notation for pairs as residuals

**abbreviation**  
\( \text{abs-residual-pair} :: \{\text{act::bn} \Rightarrow \text{'state::pt} \Rightarrow (\text{'act,'state}) \text{ residual} \} \)

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

**lemma**  
\( \text{abs-residual-pair-eqvt [simp]: p \cdot \langle \alpha, P \rangle = \langle p \cdot \alpha, p \cdot P \rangle} \)

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

3.5 Support of residuals

We only consider finitely supported states now.

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

**proof**

- have \( \text{supp} (\alpha, P) = \text{supp} ([\text{bn} \alpha]\text{set.} (\alpha, P)) \)
  by (simp add: residual.abs-eq-iff bn-eqvt)
- then show \( \text{thesis} \) by (simp add: supp-Abs)
  qed

**lemma**  
\( \text{bn-abs-residual-fresh [simp]: bn} \alpha ♯ (\langle \alpha, P :: \text{fs} \rangle) \)

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

**lemma**  
\( \text{finite-suppp-abs-residual-pair [simp]: finite} (\text{supp} (\alpha, P :: \text{fs})) \)

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

3.6 Equality between residuals

**lemma**  
\( \text{residual-eq-iff-perm:} \langle \alpha1, P1 \rangle = \langle \alpha2, P2 \rangle \iff (\exists p. \supp (\alpha1, P1) - \text{bn} \alpha1 = \supp (\alpha2, P2) - \text{bn} \alpha2 \land (\supp (\alpha1, P1) - \text{bn} \alpha1) ♯ p \land p \cdot (\alpha1, P1) = (\alpha2, P2) \land p \cdot \text{bn} \alpha1 = \text{bn} \alpha2) \)

(is \( ?l \iff ?r \))

**proof**

- assume \( ?l \)
  then have \( [\text{bn} \alpha1]\text{set.} (\alpha1, P1) = [\text{bn} \alpha2]\text{set.} (\alpha2, P2) \)
    by (simp add: residual.abs-eq-iff)
  then obtain \( p \) where \( (\text{bn} \alpha1, (\alpha1, P1)) \approx\text{set} ((=)) \supp p (\text{bn} \alpha2, (\alpha2, P2)) \)
    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 \( (\text{bn} \alpha1, (\alpha1, P1)) \approx\text{set} ((=)) \supp p (\text{bn} \alpha2, (\alpha2, P2)) \)
    using alpha-set.simps by blast
  then have \( [\text{bn} \alpha1]\text{set.} (\alpha1, P1) = [\text{bn} \alpha2]\text{set.} (\alpha2, P2) \)
    using Abs-eq-iff(1) by blast
  then show \( ?l \)
    by (simp add: residual.abs-eq-iff)
  qed
lemma residual-eq-iff-perm-renaming: \((\alpha_1, P_1) = (\alpha_2, P_2) \iff (\exists p. \text{supp}(\alpha_1, P_1) - \text{bn} \alpha_1 = \text{supp}(\alpha_2, P_2) - \text{bn} \alpha_2 \land (\text{supp}(\alpha_1, P_1) - \text{bn} \alpha_1) \# p \land p \cdot (\alpha_1, P_1) = (\alpha_2, P_2) \land p \cdot \text{bn} \alpha_1 = \text{bn} \alpha_2 \land \text{supp} p \subseteq \text{bn} \alpha_1 \cup p \cdot \text{bn} \alpha_1)\)

(is ?l \iff ?r)

proof

assume ?l

then obtain \(p\) where \(p: \text{supp}(\alpha_1, P_1) - \text{bn} \alpha_1 = \text{supp}(\alpha_2, P_2) - \text{bn} \alpha_2 \land (\text{supp}(\alpha_1, P_1) - \text{bn} \alpha_1) \# p \land p \cdot (\alpha_1, P_1) = (\alpha_2, P_2) \land p \cdot \text{bn} \alpha_1 = \text{bn} \alpha_2\)

by (metis residual-eq-iff-perm)

moreover obtain \(q\) where \(q-p: \forall b \in \text{bn} \alpha_1. q \cdot b = p \cdot b\) and \(\text{supp}-q: \text{supp} q \subseteq \text{bn} \alpha_1 \cup p \cdot \text{bn} \alpha_1\)

by (metis set-renaming-perm2)

have \(\text{supp} q \subseteq \text{supp} p\)

proof

fix \(a\)

assume \(*: a \in \text{supp} q\) then show \(a \in \text{supp} p\)

proof (cases \(a \in \text{bn} \alpha_1\))

  case True then show \(?\text{thesis}\)

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

next

  case False then have \(a \in p \cdot \text{bn} \alpha_1\)

    using \(*: \text{supp}-q\) using UnE subsetCE by blast

    with \(\text{false have} p \cdot a \neq a\)

    by (metis mem-permute-iff)

    then show \(?\text{thesis}\)

    using fresh-def fresh-perm by blast

qed

with \(p\) have \((\text{supp}(\alpha_1, P_1) - \text{bn} \alpha_1) \# q\)

by (meson fresh-def fresh-star-def subset-Iff)

moreover with \(p\) and \(q-p\) have \(\forall a. a \in \text{supp} \alpha_1 \implies q \cdot a = p \cdot a\) and \(\forall a. a \in \text{supp} P_1 \implies q \cdot a = p \cdot a\)

by (metis Diff-Iff fresh-perm fresh-star-def UnCI supp-Pair+)

then have \(q \cdot \alpha_1 = p \cdot \alpha_1\) and \(q \cdot P_1 = p \cdot P_1\)

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

ultimately show \(?r\)

using \(\text{supp}-q\) by (metis Pair-eqvt bn-eqvt)

next

assume \(?r\) then show \(?l\)

by (meson residual-eq-iff-perm)

qed

3.7 Strong induction

lemma residual-strong-induct:

assumes \(\forall (\text{act}::'act::\text{bn}) (\text{state}::'\text{state}::\text{fs}) (c::'a::\text{fs}). \text{bn} \text{act} \# c \implies P \cdot c (\text{act}, \text{state})\)

shows \(P \cdot c\) residual

proof (rule residual.abs-induct, clarify)
fix \textit{act} :: 'act\ and \textit{state} :: 'state

obtain \( p \) where 1: \((p \cdot \textit{bn}\ \textit{act}) \# c \) and 2: \( \text{supp}\ (\textit{act},\textit{state}) \# p \)

\begin{proof}
\begin{enumerate}
\item show \( \text{finite}\ (\textit{bn}\ \textit{act})\) by (fact \textit{bn}-finite)
\item show \( \text{finite}\ (\text{supp}\ c)\) by (fact \text{finite-supp})
\item show \( \text{finite}\ (\text{supp}\ \langle\textit{act},\textit{state}\rangle)\) by (fact \text{finite-supp-abs-residual-pair})
\item show \( \text{bn}\ \text{act} \# \langle\textit{act},\textit{state}\rangle\) by (fact \textit{bn-abs-residual-fresh})
\end{enumerate}
\end{proof}

from 2 have \( \langle p \cdot \textit{act}, p \cdot \textit{state}\rangle = \langle\textit{act},\textit{state}\rangle\)

\begin{proof}
\item unfolding \text{supp-perm-eq} by \text{fastforce}
\item using \text{assms 1} by (\text{metis bn-eqvt})
\end{proof}

\subsection*{3.8 Other lemmas}

\textbf{lemma residual-empty-bn-eq-iff}:

\begin{enumerate}
\item \textbf{assumes} \( \text{bn}\ \alpha_1 = \{\}\)
\item \textbf{shows} \( \langle\alpha_1,\textit{P}_1\rangle = \langle\alpha_2,\textit{P}_2\rangle \iff \alpha_1 = \alpha_2 \land P_1 = P_2\)
\end{enumerate}

\begin{proof}
\begin{enumerate}
\item \textbf{assume} \( \langle\alpha_1,\textit{P}_1\rangle = \langle\alpha_2,\textit{P}_2\rangle\)
\item \textbf{with assms have} \( [\{\}]\cdot\text{set.}\ (\alpha_1,\textit{P}_1) = [\text{bn}\ \alpha_2]\cdot\text{set.}\ (\alpha_2,\textit{P}_2)\)
\item \textbf{by} (simp add: residual.abs-eq-iff)
\item \textbf{then obtain} \( p \) where \( [\{\}], (\alpha_1,\textit{P}_1) \approx\text{set.}\ ((=))\cdot\text{supp}\ p\ (\text{bn}\ \alpha_2, (\alpha_2,\textit{P}_2))\)
\item \textbf{using} \text{Abs-eq-iff}(1) by \text{blast}
\item \textbf{then show} \( \alpha_1 = \alpha_2 \land P_1 = P_2\)
\item \textbf{unfolding} \text{alpha-set} \textbf{using} \text{supp-perm-eq by} \text{fastforce}
\end{enumerate}
\end{proof}

\textbf{qed}

\begin{proof}
\begin{enumerate}
\item \textbf{The following lemma is not about residuals, but we have no better place for it.}
\item \textbf{lemma set-bounded-supp}:
\item \textbf{assumes} \( \text{finite}\ S\) and \( \bigwedge x. x \in X \implies \text{supp}\ x \subseteq S\)
\item \textbf{shows} \( \text{supp}\ X \subseteq S\)
\item \textbf{proof} –
\item \textbf{have} \( S\ \text{supports}\ X\)
\item \textbf{unfolding} \text{supports-def} \textbf{proof} (clarify)
\item \textbf{fix} \( a\ b\)
\item \textbf{assume} \( a: \ a \notin S\) and \( b: \ b \notin S\)
\{ \text{fix} \ x \text{ assume} \ x \in X \text{ then have} \( (a \Rightarrow b) \cdot x = x\)
\item \textbf{using} \( a\ b\ \bigwedge x. x \in X \implies \text{supp}\ x \subseteq S\) by (meson \text{fresh-def subsetCE})
\end{enumerate}
\end{proof}
4 Nominal Transition Systems and Bisimulations

4.1 Basic Lemmas

lemma symp-eqvt [eqvt]:
  assumes symp R shows symp (p · R)
using assms unfolding symp-def by (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 ⊩ ϕ =⇒ p · P ⊩ p · ϕ
  and transition-eqvt [eqvt]: P → α Q =⇒ p · P → p · α Q
begin

  lemma transition-eqvt ':
    assumes P → ⟨α, Q⟩ shows p · P → ⟨p · α, p · Q⟩
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 is-bisimulation R
  ⟨P, Q⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)

proof (clarify)
  fix P Q ϕ
  assume *: (p ∘ R) P Q and **: P ⊢ ϕ
  from * have R (−p ∘ P) (−p ∘ Q)
  ⟨P, Q⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)

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)
  ⟨P, Q⟩ by (clarify)
  ⟨P, Q’⟩ by (clarify)

moreover have bn (−p ∘ α) *∗ (−p ∘ Q)
  ⟨P, Q’⟩ by (clarify)

moreover have −p ∘ P → (−p ∘ α, −p ∘ P’)
  ⟨P, Q’⟩ by (clarify)

moreover have −p ∘ P’ → (−p ∘ α, −p ∘ P’)
  ⟨P, Q’⟩ by (clarify)

ultimately obtain Q’ where T: −p ∘ Q → ⟨−p ∘ α,Q’⟩ and R: R (−p ∘ P’)

proof (clarify)
  from T have Q → ⟨α, p ∘ Q’⟩
  ⟨P, Q’⟩ by (clarify)

moreover from R have (p ∘ R) P’ (p ∘ Q’)
  ⟨P, Q’⟩ by (clarify)

ultimately show ∃ Q’ Q → ⟨α,Q’⟩ ∧ (p ∘ R) P’ Q’
  ⟨P, Q’⟩ by (clarify)

qed

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

lemma bisimilar-eqvt :

13
assumes $P \sim Q$ shows $(p \cdot P) \sim (p \cdot Q)$

proof
from assms obtain $R$ where $*: is\text{-}bisimulation R \land R P Q$
  unfolding is\text{-}bisimulation-def ..
then have is\text{-}bisimulation $(p \cdot R)$
  by (simp add: is\text{-}bisimulation-eqvt)
moreover from $*$ have $(p \cdot R) (p \cdot P) (p \cdot Q)$
  by (metis eqvt\text{-}apply permute\text{-}boolI)
ultimately show $(p \cdot P) \sim (p \cdot Q)$
  unfolding is\text{-}bisimulation-def by auto
qeds

lemma bisimilar\text{-}reflp: reflp bisimilar
proof (rule reflpI)
  fix $x$
  have is\text{-}bisimulation $(=)$
    unfolding is\text{-}bisimulation-def by (simp add: symp-def)
  then show $x \sim x$
    unfolding bisimilar-def by auto
qeds

lemma bisimilar\text{-}symp: symp bisimilar
proof (rule sympI)
  fix $P Q$
  assume $P \sim Q$
  then obtain $R$ where $*: is\text{-}bisimulation R \land R P Q$
    unfolding is\text{-}bisimulation-def ..
  then have $R Q P$
    unfolding is\text{-}bisimulation-def by (simp add: symp-def)
  with $*$ show $Q \sim P$
    unfolding is\text{-}bisimulation-def by auto
qeds

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

lemma bisimilar\text{-}transp: transp bisimilar
proof (rule transpI)
  fix $P Q R$
  assume $PQ: P \sim Q$ and $QR: Q \sim R$
  let $?\text{bisim} = bisimilar OO bisimilar$
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 \( \forall P Q. ?\text{bisim} P Q \rightarrow (\forall \varphi. P \vdash \varphi \rightarrow Q \vdash \varphi) \)
using bisimilar-is-bisimulation is-bisimulation-def by auto
moreover have \( \forall P Q. ?\text{bisim} P Q \rightarrow (\forall \alpha P', bn \alpha \sharp* Q \rightarrow P \rightarrow (\alpha,P') \rightarrow (\exists Q', Q \rightarrow (\alpha,Q') \land ?\text{bisim} P') \)
proof (clarify)
  fix P R Q \alpha P'
  assume PR: P ~ R and RQ: R ~ Q fresh: bn \alpha \sharp* Q and trans: P \rightarrow (\alpha,P')
  — rename \((\alpha,P')\) to avoid R, without touching Q
  obtain p where 1: \((p \cdot bn \alpha) \sharp* R \land 2: \text{supp} ((\alpha,P'), Q) \sharp* p
  proof (rule at-set-avoiding2[of bn \alpha R ((\alpha,P'), Q), THEN exE])
    show finite \((bn \alpha)\) by (fact bn-finite)
  next
    show finite \((\text{supp}\ R)\) by (fact finite-supp)
  next
    show finite \((\text{supp} ((\alpha,P'), Q))\) by (simp add: finite-supp supp-Pair)
  next
    show bn \alpha \sharp* ((\alpha,P'), Q) by (simp add: fresh-star-Pair)
  qed metis
from 2 have 3: \(\text{supp} (\alpha,P') \sharp* p\) and 4: \(\text{supp} Q \sharp* p\)
  by (simp add: fresh-star-Un supp-Pair)+
from 3 have \(p \cdot \alpha, p \cdot P' = (\alpha,P')\)
  using supp-perm-eq by fastforce
then obtain pR' where 5: \(R \rightarrow (p \cdot \alpha, pR')\) and 6: \(p \cdot P' \sim pR'\)
  using PR trans 1 by (metis (mono-tags, lifting) bisimilar-is-bisimulation bn-eqvt is-bisimulation-def)
from fresh and 4 have bn \((p \cdot \alpha) \sharp* Q\)
  by (metis bn-eqvt fresh-star-permute-iff supp-perm-eq)
then obtain pQ' where 7: \(Q \rightarrow (p \cdot \alpha, pQ')\) and 8: \(pR' \sim pQ'\)
  using RQ 5 by (metis (full-types) bisimilar-is-bisimulation is-bisimulation-def)
from 7 have \(Q \rightarrow (\alpha, \neg p \cdot pQ')\)
  using 4 by (metis permute-minus-cancel(2) supp-perm-eq transition-eqvt')
moreover from 6 and 8 have \(?\text{bisim} P' (\neg p \cdot pQ')\)
  by (metis (no-types, hide-lams) bisimilar-eqvt permute-minus-cancel(2) relcompp.simps)
  ultimately show \(\exists Q'. Q \rightarrow (\alpha,Q') \land ?\text{bisim} P' Q'\)
  by metis
ultimately have \( \text{is-bisimulation} \ ?\text{bisim} \)
unfolding \( \text{is-bisimulation-def} \) by \text{metis}
moreover have \( ?\text{bisim} P R \)
using \( PQ QR \) by \text{blast}
ultimately show \( P \sim R \)
unfolding \( \text{bisimilar-def} \) by \text{meson}

\lemmas{bisimilar-equivp}{equivp \ bisimilar}
\text{by} (\text{metis bisimilar-reflp bisimilar-symp bisimilar-transp equivp-reflp-symp-transp})

\lemmas{bisimilar-simulation-step}{assumes} \( P \sim Q \) and \( bn \ \alpha \# \star Q \) and \( P \rightarrow \langle \alpha, P' \rangle \)
\obtain \( Q' \) where \( Q \rightarrow \langle \alpha, Q' \rangle \) and \( P' \sim Q' \)
\text{using} \( \text{assms by} \) (\text{metis (poly-guards-query) bisimilar-is-bisimulation is-bisimulation-def})

\end{theory}

\section{5 Infinitary Formulas}

\subsection{5.1 Infinitely branching trees}

First, we define a type of trees, with a constructor \( \text{tConj} \) 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.

\begindatatype{\langle 'idx,'pred,'act \rangle \ Tree =}
\text{tConj} \ (\langle 'idx,'pred,'act \rangle \ Tree \ \text{set['idx]} \ — \ \text{potentially infinite sets of trees} \\
| \text{tNot} \ (\langle 'idx,'pred,'act \rangle \ Tree} \\
| \text{tPred} 'pred \\
| \text{tAct} 'act \ (\langle 'idx,'pred,'act \rangle \ Tree}
\enddatatype

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.

\begindatatypes{Tree-wf :: \langle 'idx,'pred,'act \rangle \ Tree \ \text{rel} \ \text{where} \\
t \in \ \text{set-set tset} \implies (t, \text{tConj tset}) \in \text{Tree-wf}
We define a permutation operation on the type of trees.

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

primrec permute-Tree :: perm ⇒ (─,─,─) Tree ⇒ (─,─,─) Tree where
  p · (tConj tset) = tConj (map-bset (permute p) tset) — neat trick to get around the fact that tset is not of permutation type yet
| p · (tNot t) = tNot (p · t)
| p · (tPred ϕ) = tPred (p · ϕ)
| p · (tAct α t) = tAct (p · α) (p · t)

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

\textbf{qed}

\textbf{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.

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

\textbf{declare} \textit{permute-Tree.simps(1) [simp del]}

The relation \textit{Tree-wf} is equivariant.

\textbf{lemma} \textit{Tree-wf-eqvt-aux}:
\textbf{assumes} $(t1, t2) \in \text{Tree-wf}$ \textbf{shows} $(p \cdot t1, p \cdot t2) \in \text{Tree-wf}$
\textbf{using} \textit{asms\ proof} (induction rule: \textit{Tree-wf.induct})
\textbf{fix} $t :: (a,b,c) \text{ Tree}$ \textbf{and} $tset :: (a,b,c) \text{ Tree set}$
\textbf{assume} $t \in \text{set-bset tset}$ \textbf{then show} $(p \cdot t, p \cdot tConj tset) \in \text{Tree-wf}$
\textbf{by} (metis \textit{Tree-wf.intros(1) mem-permute-iff permute-Tree-tConj set-bset-eqvt})

\textbf{next}
\textbf{fix} $t :: (a,b,c) \text{ Tree}$
\textbf{show} $(p \cdot t, p \cdot tNot t) \in \text{Tree-wf}$
\textbf{by} (metis \textit{Tree-wf.intros(2) permute-Tree.simps(2)})

\textbf{next}
\textbf{fix} $t :: (a,b,c) \text{ Tree}$ \textbf{and} $\alpha$
\textbf{show} $(p \cdot t, p \cdot tAct \alpha t) \in \text{Tree-wf}$
\textbf{by} (metis \textit{Tree-wf.intros(3) permute-Tree.simps(4)})

\textbf{qed}

\textbf{lemma} \textit{Tree-wf-eqvt} [eqvt, simp]: $p \cdot \text{Tree-wf} = \text{Tree-wf}$
\textbf{proof}
\textbf{show} $p \cdot \text{Tree-wf} \subseteq \text{Tree-wf}$
\textbf{by} (auto simp add: \textit{permute-set-def}) (rule \textit{Tree-wf-eqvt-aux})

\textbf{next}
\textbf{show} $\text{Tree-wf} \subseteq p \cdot \text{Tree-wf}$
\textbf{by} (auto simp add: \textit{permute-set-def}) (metis \textit{Tree-wf-eqvt-aux permute-minus-cancel(1)})

\textbf{qed}

\textbf{lemma} \textit{Tree-wf-eqvt'}: eqvt \textit{Tree-wf}
\textbf{by} (metis \textit{Tree-wf-eqvt eqvtI})

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

\textbf{lemma} \textit{supp-tConj [simp]}: $\text{supp} \ (tConj tset) = \text{supp} \ tset$
\textbf{unfolding} \textit{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 α ∪ supp t
unfolding supp-def by (simp add: Collect-imp-eq Collect-neg-eq)

5.2 Trees modulo α-equivalence

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

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

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

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

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

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

lemma supp-rel-cong [fundef-cong]:
  [ x=x'; y=y'; a b. R ((a ⊛ b) · x') x' ←→ R' ((a ⊛ b) · 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'; a b. a ∈ set-bset x' ⇒ b ∈ set-bset y' ⇒ R a b ←→ 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' · x') y' ←→ R' (p' · 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) \iff 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 Quotient3-Tree)

lemma permute-rep-abs-Tree_p:
  obtains p where rep-Tree_p (abs-Tree_p t) = p \cdot t
by (metis Quotient3-Tree_p)

lift-definition Tree-wf_p :: ('idx,'pred::pt,'act::bn) Tree_p rel is
  Tree-wf .

lemma Tree-wf_pI [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 (abs-Tree_p a, abs-Tree_p b) \in Tree-wf_p
using assms by (metis (Tree-wf_pI permute-zero)

lemma Tree-wf_pE:
  assumes (a_p, b_p) \in Tree-wf_p
obtains a b where a_p = abs-Tree_p a and b_p = abs-Tree_p b and (a,b) \in 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-equiv wf-Tree-wf wf-hull-rel-recomp wf-inv-image)

next
  show Tree-wf_p \subseteq 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) \in 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) \in Tree-wf
      by (rule Tree-wf_pE)
    from 1 obtain p where 4: rep-Tree_p a_p = p \cdot a
      by (metis permute-rep-abs-Tree_p)
    from 2 obtain q where 5: rep-Tree_p b_p = q \cdot b
      by (metis permute-rep-abs-Tree_p)
have \((p \cdot a, q \cdot a) \in \text{hull-rel}\)
by \((\text{metis hull-rel.simps permute-minus-cancel(2) permute-plus})\)
moreover from 3 have \((q \cdot a, q \cdot b) \in \text{Tree-wf}\)
by \((\text{rule Tree-wf-eqvt-aux})\)
ultimately show \((a_p, b_p) \in \text{inv-image (hull-rel O Tree-wf)}\) \text{rep-Tree}_p
using 4 5 by auto
qed

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

Here it comes . . .

\text{function (sequential)}
alpha-Tree :: \langle idx::pred::pt::act::bn \rangle \text{ Tree} \Rightarrow \langle idx::pred::act \rangle \text{ Tree} \Rightarrow \text{bool (infix =}_\text{a} 50\rangle \text{ where}
\begin{align*}
\alpha & = (\_\alpha) \\
\alpha\_\text{tConj} & = t\text{Conj tset}_1 = _\alpha t\text{Conj tset}_2 \iff \text{rel-bset Alpha-Tree tset}_1 tset_2 \\
\alpha\_\text{tNot} & = t\text{Not} t1 = _\alpha t\text{Not} t2 \iff t1 = _\alpha t2 \\
\alpha\_\text{tPred} & = t\text{Pred} \varphi_1 = _\alpha t\text{Pred} \varphi_2 \iff \varphi_1 = \varphi_2 \\
\text{— the action may have binding names}
\alpha\_\text{tAct} & = t\text{Act} \alpha t_1 = _\alpha t\text{Act} \alpha_2 t_2 \iff \\
(\exists p. (\text{bn} \_\alpha t_1, t_1) \Rightarrow \text{set Alpha-Tree (supp-rel Alpha-Tree)} \ p (\text{bn} \_\alpha_2 t_2) \land (\text{bn} \_\alpha_1, t_1) \Rightarrow \text{set (}=(i)\text{ supp} \ p (\text{bn} \_\alpha_2, \_\alpha_2)) \\
\alpha\_\text{other} & = _\alpha \_\alpha \iff \text{False} \\
\text{— 254 subgoals (!)}
\end{align*}
by \text{pat-completeness auto}

\text{termination proof}
\begin{align*}
\text{let } \_R & = \text{inv-image (max-ext Tree-wf}_p \text{) alpha-Tree-termination} \\
\text{show } \_\text{wf} \_R \\
& \text{by } (\text{metis max-ext-wf wf-Tree-wf}_p \text{ wf-inv-image})
\end{align*}
\text{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.

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

\text{lemma alpha-Tree-induct\[case-names tConj tNot tPred tAct\text{, consumes 1]:}
\begin{align*}
& \text{assumes } t1 = _\alpha t2 \\
& \text{and } \bigwedge tset_1 tset_2. (\bigwedge a \_b. a \in \text{set-bset tset}_1 \Rightarrow b \in \text{set-bset tset}_2 \Rightarrow a = _\alpha b \\
& \Rightarrow P a \_b) \Rightarrow \text{rel-bset } (_\alpha) tset_1 tset_2 \Rightarrow P (tConj tset_1) (tConj tset_2) \\
& \text{and } \bigwedge t1 t2. t1 = _\alpha t2 \Rightarrow P t1 t2 \Rightarrow P (t\text{Not} t1) (t\text{Not} t2)
\end{align*}
and \( \land \varphi. \ P (t_{\text{Pred}} \varphi) (t_{\text{Pred}} \varphi) \)

and \( \land (p \cdot t) =_a t2 \implies P (p \cdot t1) t2 \)

\( (\land a \ b. ((a = b) \cdot t) =_a t1 \implies P ((a = b) \cdot t1) t1 \) \implies (\land a \ b. ((a = b) \cdot t2) =_a t2 \implies P ((a = b) \cdot t2) t2 \)

\( (\exists p. (bn \alpha1, t1) \approx set (=_a) p (bn \alpha2, t2) \land (bn \alpha1, \alpha t) \approx set (= p) (supp-rel (=_a)) p (bn \alpha2, t2) \) \implies P (t_{\text{Act}} \alpha1 t1) (t_{\text{Act}} \alpha2 t2)

shows \( P t1 t2 \)

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

alpha-equivalence is equivariant.

lemma alpha-Tree-eqvt-aux:

assumes \( \land a \ b. (a = b) \cdot t =_a t \leftarrow p \cdot (a = b) \cdot t =_a p \cdot t \)

shows \( p \cdot \text{supp-rel} (=_a) t = \text{supp-rel} (=_a) (p \cdot t) \)

proof –

\[
\begin{align*}
\text{fix } a \\
\text{let } ?B = \{ b. \neg ((a = b) \cdot t) =_a t \} \\
\text{let } ?pB = \{ b. \neg ((p \cdot a = b) \cdot p \cdot t) =_a (p \cdot t) \} \\
\{ \\
\text{assume finite } ?B \\
\text{moreover have inj-on (unpermute } p) ?pB \\
\text{by (simp add: inj-on-def unpermute-def) } \\
\text{moreover have unpermute } p \cdot ?pB \subseteq ?B \\
\text{using assms by auto (metis (erased, lifting) eqvt-bound permute-eqvt swap-eqvt) } \\
\text{ultimately have finite } ?pB \\
\text{by (metis inj-on-finite) } \\
\}
\text{moreover} \\
\{ \\
\text{assume finite } ?pB \\
\text{moreover have inj-on (permute } p) ?B \\
\text{by (simp add: inj-on-def) } \\
\text{moreover have permute } p \cdot ?B \subseteq ?pB \\
\text{using assms by auto (metis (erased, lifting) permute-eqvt swap-eqvt) } \\
\text{ultimately have finite } ?B \\
\text{by (metis inj-on-finite) } \\
\}
\text{ultimately have infinite } ?B \iff \text{infinite } ?pB \\
\text{by auto } \\
\}
\text{then show } ?thesis \\
\text{by (auto simp add: supp-rel-def permute-set-def) (metis eqvt-bound)}
\] qed

lemma alpha-Tree-eqvt': \( t1 =_a t2 \leftarrow p \cdot t1 =_a p \cdot t2 \)

proof (induction t1 t2 rule: alpha-Tree-induct')

case (alpha-tConj tset1 tset2) show ?case
proof

assume *: tConj tset1 =_α tConj tset2

{ fix x
  assume x ∈ set-bset (p · tset1)
  then obtain x’ where 1: x’ ∈ set-bset tset1 and 2: x = p · x’
  by (metis imageE permute-bset.rep-eq permute-set-eq-image)
  from 1 obtain y’ where 3: y’ ∈ set-bset tset2 and 4: x’ =_α y’
  using * by (metis (mono-tags, lifting) Formula.alpha-tConj rel-bset.rep-eq
rel-set-def)
  from 3 have p · y’ ∈ set-bset (p · tset2)
  by (metis mem-permute-iff set-bset-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 Formula.alpha-tConj permute-Tree-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 x =_α p · 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 ..
}

ultimately show p · tConj tset1 =_α p · tConj tset2
by (simp add: rel-bset-def rel-set-def)

next
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' =ₐ 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 =ₐ y
    using alpha-tConj.IH by blast
  with 4 have ∃ x ∈ set-bset tset1. x =ₐ y ..
}
ultimately show tConj tset1 =ₐ 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 (=ₐ) t1 = supp-rel (=ₐ) (p · t1)
    by (rule alpha-Tree-eqvt-aux)
  from alpha-tAct.IH(3) have t2: p · supp-rel (=ₐ) t2 = supp-rel (=ₐ) (p · t2)
    by (rule alpha-Tree-eqvt-aux)
show ?case
proof
  assume tAct α1 t1 =ₐ tAct α2 t2
  then obtain q where 1: (bn α1, t1) ≈set (=ₐ) (supp-rel (=ₐ)) 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 (=ₐ) (p · t1) = bn (p · α1) = supp-rel
    (=ₐ) (p · t2) = bn (p · α2)
    by (metis Diff-eqvt alpha-set bn-eqvt)
  moreover from 1 and t1 have (supp-rel (=ₐ) (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 · q · t1 =ₐ p · t2
    by (simp add: alpha-set)
  moreover from 2 have p · q · −p · bn (p · α1) = bn (p · α2)
    by (simp add: alpha-set bn-eqvt)
  ultimately have (bn (p · α1), p · t1) ≈set (=ₐ) (supp-rel (=ₐ)) (p + q − p) (bn (p · α2), p · t2)
    by (simp add: alpha-set)
  moreover from 2 have (bn (p · α1), p · α1) ≈set (=) supp (p + q − p) (bn
    (p · α2), p · 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 · tAct α1 t1 =ₐ p · tAct α2 t2
    by auto

24
next

assume \( p \cdot tAct \alpha \, t_1 =_\alpha p \cdot tAct \alpha \, t_2 \)
then obtain \( q \) where 1: \( (bn \, (p \cdot \alpha \, 1), p \cdot t_1) \approx set \ (=_\alpha) \) \( q \) \( (bn \, (p \cdot \alpha \, 2), p \cdot t_2) \) and 2: \( (bn \, (p \cdot \alpha \, 1), p \cdot \alpha \, 1) \approx set \ (=_\alpha) \) \( supp \, q \) \( (bn \, (p \cdot \alpha \, 2), p \cdot \alpha \, 2) \)

by auto

\{
from 1 and \( t_1 \) and \( t_2 \) have \( supp \ (=_\alpha) \) \( t_1 - bn \, \alpha \, 1 = supp \ (=_\alpha) \) \( t_2 - bn \, \alpha \, 2 \)
   by (metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff)
moreover with 1 and \( t_2 \) have \( (supp \ (=_\alpha) \) \( t_1 - bn \, \alpha \, 1) \times^* (\sim p + q + p) \)
   by (auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting) Diff1 bn-eqvt mem-permute-iff permute-minus-cancel(2))
moreover from 1 have \( -p \cdot q \cdot p \cdot t_1 =_\alpha t_2 \)
   using alpha-tAct.IH(1) by (simp add: alpha-set) (metis (no-types, lifting) permute-eqgt 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 \( (bn \, \alpha \, 1, t_1) \approx set \ (=_\alpha) \) \( (supp \ (=_\alpha) \) \( -p + q + p) \) \( (bn \, \alpha \, 2, \alpha \, 2) \)
   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 \( (supp \, \alpha \, 1 - bn \, \alpha \, 1) \times^* (\sim p + q + p) \)
   by (auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting) Diff1 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 \( (bn \, \alpha \, 1, \alpha \, 1) \approx set \ (=_\alpha) \) \( supp \ (=_\alpha) \) \( -p + q + p) \) \( (bn \, \alpha \, 2, \alpha \, 2) \)
   by (simp add: alpha-set)
\}
ultimately show \( tAct \, \alpha \, t_1 =_\alpha tAct \, \alpha \, t_2 \)
by auto
qed

lemma alpha-Tree-eqvt [eqvt]: \( t_1 =_\alpha t_2 \implies p \cdot t_1 =_\alpha p \cdot t_2 \)
by (metis alpha-Tree-eqvt)

(\( =_\alpha \)) is an equivalence relation.

lemma alpha-Tree-reflp: reflp alpha-Tree
proof (rule reflpI)

25
fix \( t :: (\alpha', \beta', \gamma') \) Tree

proof (induction \( t \))
\begin{align*}
\text{case } t \text{Conj then show } ?\text{case by (metis alpha-tConj rel-bset.rep-eq rel-setI)} \\
\text{next}
\end{align*}

\begin{align*}
\text{case } t \text{Not then show } ?\text{case by (metis alpha-tNot)} \\
\text{next}
\end{align*}

\begin{align*}
\text{case } t \text{Pred then show } ?\text{case by (metis alpha-tPred)} \\
\text{next}
\end{align*}

\begin{align*}
\text{case } t \text{Act then show } ?\text{case by (metis (mono-tags) alpha-tAct alpha-refl (1))} \\
\text{qed}
\end{align*}

qed

lemma alpha-Tree-symp: symp alpha-Tree
proof (rule sympI)
fix \( x, y :: (\alpha', \beta', \gamma') \) Tree
assume \( x =_\alpha y \) then show \( y =_\alpha x \)
proof (induction \( x, y \) rule: alpha-Tree-induct)
\begin{align*}
\text{case } t \text{Conj then show } ?\text{case by (simp add: rel-bset-def rel-set-def Ball-def Bex-def) metis} \\
\text{next}
\end{align*}

\begin{align*}
\text{case } (t \text{Act } \alpha_1 \ t_1 \ \alpha_2 \ t_2) \\
\text{then obtain } p \ where \ (bn \ \alpha_1, \ t_1) \approx set (=_\alpha) (supp-rel (=_\alpha)) p \ (bn \ \alpha_2, \ t_2) \\
\land (bn \ \alpha_1, \ \alpha_1) \approx set (=_\alpha) supp p \ (bn \ \alpha_2, \ \alpha_2) \\
\text{by auto} \\
\text{then have } (bn \ \alpha_2, \ t_2) \approx set (=_\alpha) (supp-rel (=_\alpha)) \neg p \ (bn \ \alpha_1, \ t_1) \land (bn \ \alpha_2, \ \alpha_2) \approx set (=_\alpha) supp \neg p \ (bn \ \alpha_1, \ \alpha_1) \\
\text{using } t \text{Act.IH by (metis (mono-tags, lifting) alpha-Tree-eqvt alpha-sym (1) permute-minus-cancel (2))} \\
\text{then show } ?\text{case} \\
\text{by auto} \\
\text{qed simp-all}
\end{align*}

qed

lemma alpha-Tree-transp: transp alpha-Tree
proof (rule transpI)
fix \( x, y, z :: (\alpha', \beta', \gamma') \) Tree
assume \( x =_\alpha y \) and \( y =_\alpha z \) then show \( x =_\alpha z \)
proof (induction \( x, y \) arbitrary; \( z \) rule: alpha-Tree-induct)
\begin{align*}
\text{case } (t \text{Conj } \text{tset-x } \text{tset-y}) \text{ show } ?\text{case} \\
\text{proof (cases } z) \\
\text{fix } \text{tset-z} \\
\text{assume } z :: z =_\alpha t \text{Conj } \text{tset-z} \\
\text{have } \text{rel-bset } (=_\alpha) \text{ tset-x } \text{tset-z} \\
\text{unfolding } \text{rel-bset.rep-eq rel-set-def Ball-def Bex-def} \\
\text{proof} \\
\text{show } \forall \ x'. \ x' \in \text{set-bset } \text{tset-x} \to (\exists z'. \ z' \in \text{set-bset } \text{tset-z } x' =_\alpha z') \\
\text{proof (rule allI, rule impI)}
\end{align*}

26
fix $x'$ assume $1: x' \in \text{set-bset tset-x}$
then obtain $y'$ where $2: y' \in \text{set-bset tset-y}$ and $3: x'=_{\alpha} y'$
by (metis rel-bset.rep-eq rel-set-def tConj.hyps)
from $2$ obtain $z'$ where $4: z' \in \text{set-bset tset-z}$ and $5: y'=_{\alpha} z'$
by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prems $z$)
from $1 \ 2 \ 3 \ 5$ have $x'=_{\alpha} z'$
by (rule (tConj.IH))
with $4$ show $\exists z'. z' \in \text{set-bset tset-z} \land x'=_{\alpha} z'$
by auto qed
next
show $\forall z'. z' \in \text{set-bset tset-z} \rightarrow (\exists x', x' \in \text{set-bset tset-x} \land x'=_{\alpha} z')$
proof (rule allI, rule impI)
fix $z'$ assume $1: z' \in \text{set-bset tset-z}$
then obtain $y'$ where $2: y' \in \text{set-bset tset-y}$ and $3: y'=_{\alpha} z'$
by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prems $z$)
from $2$ obtain $x'$ where $4: x' \in \text{set-bset tset-x}$ and $5: x'=_{\alpha} y'$
by (rule (tConj.IH))
with $4 \ 2 \ 5 \ 3$ have $x'=_{\alpha} z'$
by (rule (tConj.IH))
with $4$ show $\exists x'. x' \in \text{set-bset tset-x} \land x'=_{\alpha} z'$
by auto qed
next
case $tNot$ then show $?case$
by (cases $z$) simp-all
next
case $tPred$ then show $?case$
by simp
next
case $(\text{tAct} \ \alpha \ 1 \ t1 \ \alpha2 \ t2)$ show $?case$
proof (cases $z$)
fix $\alpha \ t$
assume $z: z = \text{tAct} \ \alpha \ t$
obtain $p$ where $1: (bn \ \alpha1, \ t1) \approx \text{set} \ (=_\alpha) \ (\text{supp-rel} \ (=_\alpha)) \ p \ (bn \ \alpha2, \ t2) \land (bn \ \alpha1, \ \alpha1) \approx \text{set} \ (=_\alpha) \ \text{supp} \ p \ (bn \ \alpha2, \ \alpha2)$
using $\text{tAct.hyps}$ by auto
obtain $q$ where $2: (bn \ \alpha2, \ t2) \approx \text{set} \ (=_\alpha) \ (\text{supp-rel} \ (=_\alpha)) \ q \ (bn \ \alpha, \ t) \land (bn \ \alpha2, \ \alpha2) \approx \text{set} \ (=_\alpha) \ \text{supp} \ q \ (bn \ \alpha, \ \alpha)$
using $\text{tAct.prems}$ $z$ by auto
have $(bn \ \alpha1, \ t1) \approx \text{set} \ (=_\alpha) \ (\text{supp-rel} \ (=_\alpha)) \ (q + p) \ (bn \ \alpha, \ t)$
proof =
have $(\text{supp-rel} \ (=_\alpha)) \ t1 - bn \ \alpha1 = \text{supp-rel} \ (=_\alpha) \ t - bn \ \alpha$
using $1$ and $2$ by (metis alpha-set)
moreover have $(\text{supp-rel} \ (=_\alpha)) \ t1 - bn \ \alpha1 \ \sharp* \ (q + p)$
using 1 and 2 by \((\text{metis alpha-set fresh-star-plus})\)
moreover have \((q + p) \cdot t_1 =_\alpha t\)
using 1 and 2 and \(t\text{Act.IH}\) by \((\text{metis (no-types, lifting) alpha-Tree-eqvt alpha-set permute-minus-cancel(1) permute-plus})\)
moreover have \((q + p) \cdot bn\ a_1 = bn\ a\)
using 1 and 2 by \((\text{metis alpha-set permute-plus})\)
ultimately show \(?\text{thesis}\)
by \((\text{metis alpha-set})\)
\qedsymbol
moreover have \((bn\ a_1, a_1) \approxset (=) \supp (q + p) (bn\ a, a)\)
using 1 and 2 by \((\text{metis mono-tags alpha-trans(1) permute-plus})\)
ultimately show \(t\text{Act} a_1 t_1 =_\alpha z\)
using \(z\) by \text{auto}
\qedsymbol
\qedsymbol

\textbf{lemma} alpha-Tree-equivp: equivp alpha-Tree
by \((\text{auto intro: equivpI alpha-Tree-reflp alpha-Tree-symp alpha-Tree-transp})\)

\textit{alpha-equivalent trees have the same support modulo alpha-equivalence.}

\textbf{lemma} alpha-Tree-supp-rel:
\textbf{assumes} \(t_1 =_\alpha t_2\)
\textbf{shows} \(\supp-rel (=) t_1 = \supp-rel (=) t_2\)
using \(\text{assms}\) proof \((\text{induction rule: alpha-Tree-induct})\)
case \((t\text{Conj} t\text{set}_1 t\text{set}_2)\)
have \(\text{sym}: \forall x\ y. \ rel-bset (=) x\ y \iff \ rel-bset (=) y\ x\)
by \((\text{meson alpha-Tree-symp bset.rel-symp sympE})\)
\{
fix \(a\ b\)
from \(t\text{Conj}.\text{hyps}\) have \(*: \ rel-bset (=) ((a \iff b) \cdot t\text{set}_1) ((a \iff b) \cdot t\text{set}_2)\)
by \((\text{metis alpha-tConj alpha-Tree-eqvt permute-Tree-tConj})\)
have \(\rel-bset (=) ((a \iff b) \cdot t\text{set}_1) t\text{set}_1 \iff \ rel-bset (=) ((a \iff b) \cdot t\text{set}_2)\)
t\text{set}_2
by \((\text{rule iffI}) (\text{metis * alpha-Tree-transp bset.rel-transp sym tConj.hyps transpE})\)+
\}
then show \(?\text{case}\)
by \((\text{simp add: supp-rel-def})\)
next
case \(t\text{Not}\) then show \(?\text{case}\)
by \((\text{simp add: supp-rel-def})\)
next
case \((t\text{Act} a_1 t_1 a_2 t_2)\)
\{
fix \(a\ b\)
have \(t\text{Act} a_1 t_1 =_\alpha t\text{Act} a_2 t_2\)
using \(t\text{Act}.\text{hyps}\) by \text{simp}
then have \((a \iff b) \cdot t\text{Act} a_1 t_1 =_\alpha t\text{Act} a_1 t_1 \iff (a \iff b) \cdot t\text{Act} a_2 t_2 =_\alpha\)
\[ tAct \alpha t1 \alpha t2 \]

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 \( \alpha \)-equivalence.

**lemma** alpha-Tree-tAct:

assumes \( t1 =_\alpha t2 \)

shows \( tAct \alpha t1 =_\alpha tAct \alpha t2 \)

**proof** –

have \( (bn \alpha, t1) \approx (set (=\alpha) (supp-rel (=\alpha))) \theta (bn \alpha, t2) \)

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

moreover have \( (bn \alpha, \alpha) \approx (set (=) \supp \theta (bn \alpha, \alpha) \)

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

ultimately show \( \text{thesis} \)
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-supp-def by simp

**lemma** supp-rel-tPred [simp]: \( \supp-rel (=\alpha) (tPred \varphi) = \supp \varphi \)

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

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

**lemma** infinite-mono: \( \text{infinite } S \implies (\forall x. x \in S \implies x \in T) \implies \text{infinite } T \)
by (metis infinite-super subsetI)

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

assumes \( \text{finite } \supp-rel (=\alpha) t \)

shows \( \supp-rel (=\alpha) (tAct \alpha t) = \supp \alpha \cup \supp-rel (=\alpha) (t - bn \alpha) \)

**proof**

show \( \supp \alpha \cup \supp-rel (=\alpha) (t - bn \alpha) \subseteq \supp-rel (=\alpha) (tAct \alpha t) \)

**proof**

fix \( x \)

assume \( x \in \supp \alpha \cup \supp-rel (=\alpha) (t - bn \alpha) \)

moreover

{ assume \( x1: x \in \supp \alpha \) and \( x2: x \notin bn \alpha \)

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

**unfolding** supp-def ..
then have infinite \(\{b. (x \simeq b) \cdot \alpha \neq \alpha\} - \text{supp } \alpha\)
by (simp add: finite-supp)
moreover
\{
\begin{align*}
&\text{fix } b \\
&\text{assume } b \in \{b. (x \simeq b) \cdot \alpha \neq \alpha\} - \text{supp } \alpha \\
&\text{then have } b1: (x \simeq b) \cdot \alpha \neq \alpha \text{ and } b2: b \not\in \text{supp } \alpha - bn \alpha \\
&\quad \text{by simp+}
\end{align*}
\}
from b1 have sort-of \(x = \text{sort-of } b\)
using swap-different-sorts by fastforce
then have \((x \simeq b) \cdot (\text{supp } \alpha - bn \alpha) \neq \text{supp } \alpha - bn \alpha\)
using b2 x1 x2 by (simp add: swap-set-in)
then have \(b \in \{b. \neg (x \simeq b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}\)
by (auto simp add: alpha-set Diff-eqvt bn-eqvt)
ultimately have infinite \(\{b. \neg (x \simeq b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}\)
by (rule infinite-mono)
then have \(x \in \text{supp-rel } (=_{\alpha}) (tAct \alpha t)\)
unfolding supp-rel-def ..
\}
moreover
\{
\begin{align*}
&\text{assume } x1: x \in \text{supp-rel } (=_{\alpha}) t \text{ and } x2: x \not\in bn \alpha \\
&\text{from } x1 \text{ have infinite } \{b. \neg (x \simeq b) \cdot t =_{\alpha} t\}
\end{align*}
\}
unfolding supp-rel-def ..
then have infinite \(\{b. \neg (x \simeq b) \cdot t =_{\alpha} t\} - \text{supp-rel } (=_{\alpha}) t\)
using assms by simp
moreover
\{
\begin{align*}
&\text{fix } b \\
&\text{assume } b \in \{b. \neg (x \simeq b) \cdot t =_{\alpha} t\} - \text{supp-rel } (=_{\alpha}) t \\
&\text{then have } b1: \neg (x \simeq b) \cdot t =_{\alpha} t \text{ and } b2: b \not\in \text{supp-rel } (=_{\alpha}) t - bn \alpha \\
&\quad \text{by simp+}
\end{align*}
\}
from b1 have \((x \simeq b) \cdot t \neq t\)
by (metis alpha-Tree-reflp reflpE)
then have sort-of \(x = \text{sort-of } b\)
using swap-different-sorts by fastforce
then have \((x \simeq b) \cdot (\text{supp-rel } (=_{\alpha}) t - bn \alpha) \neq \text{supp-rel } (=_{\alpha}) t - bn \alpha\)
using b2 x1 x2 by (simp add: swap-set-in)
then have \(\text{supp-rel } (=_{\alpha}) ((x \simeq b) \cdot t) - bn ((x \simeq b) \cdot \alpha) \neq \text{supp-rel } (=_{\alpha}) t - bn \alpha\)
by (simp add: Diff-eqvt bn-eqvt)
then have \(b \in \{b. \neg (x \simeq b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}\)
by (simp add: alpha-set)
ultimately have infinite \(\{b. \neg (x \simeq b) \cdot tAct \alpha t =_{\alpha} tAct \alpha t\}\)
by (rule infinite-mono)
then have \(x \in \text{supp-rel } (=_{\alpha}) (tAct \alpha t)\)
unfolding supp-rel-def ..
ultimately show \( x \in \text{supp-rel} \ (\alpha t \ t) \)
by auto
qed

next
show \( \text{supp-rel} \ (\alpha t \ t) \subseteq \text{supp} \alpha \cup \text{supp-rel} \ (\alpha t \ t) \)
proof
fix \( x \)
assume \( x \in \text{supp-rel} \ (\alpha t \ t) \)
then have \( \ast: \text{infinite} \ \{ \ b. \ \neg (x \equiv b) \cdot t \equiv \alpha t \ \alpha t \ t \} \)
using alpha-Tree-tAct by force

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

ultimately show \( x \in \text{supp} \alpha \cup \text{supp-rel} (=_\alpha) \cdot t - bn \alpha \)
by blast
qed

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

\[ \text{quotient-type} \]
\[
\begin{align*}
\left( \text{`idx'}, \text{`pred',`act} \right) \text{Tree}_\alpha &= \left( \text{`idx'}, \text{`pred::pt',`act::bn} \right) \text{Tree} / \text{alpha-Tree} \\
\text{by} \ (\text{fact alpha-Tree-equivp})
\end{align*}
\]

\text{lemma Tree}_\alpha \cdot \text{abs-rep} \left[ \text{simp} \right]: \text{abs-Tree}_\alpha (\text{rep-Tree}_\alpha t_\alpha) = t_\alpha
by (metis Quotient-Tree_\alpha` Quotient-abs-rep)

\text{lemma Tree}_\alpha \cdot \text{rep-abs} \left[ \text{simp} \right]: \text{rep-Tree}_\alpha (\text{abs-Tree}_\alpha t) =_\alpha t
by (metis Tree_\alpha \cdot \text{abs-eq-iff} Tree_\alpha \cdot \text{abs-rep})

The permutation operation is lifted from trees.

\text{instantiation} Tree_\alpha \:: (\text{type, pt, bn}) \cdot \text{pt}
begin

\text{lift-definition} \text{permute-Tree}_\alpha \:: \text{perm} \Rightarrow \left( \text{'a'}, \text{`b',`c} \right) \text{Tree}_\alpha \Rightarrow \left( \text{'a'}, \text{`b',`c} \right) \text{Tree}_\alpha
is \text{permute}
by (\text{fact alpha-Tree-equivt})

\text{instance}
proof
fix \( t_\alpha \:: \left( \cdot, \cdot, \cdot \right) \text{Tree}_\alpha \\
show \( \theta \cdot t_\alpha = t_\alpha \)
by transfer (metis alpha-Tree-equivp equivp-reflp permute-zero)
next
fix \( p \cdot q \:: \text{perm} \) and \( t_\alpha \:: \left( \cdot, \cdot, \cdot \right) \text{Tree}_\alpha \\
show \( (p + q) \cdot t_\alpha = p \cdot q \cdot t_\alpha \)
by transfer (metis alpha-Tree-equivp equivp-reflp permute-plus)
qed

end

The abstraction function from trees to trees modulo \( \alpha \)-equivalence is equivariant. The representation function is equivariant modulo \( \alpha \)-equivalence.

\text{lemmas} \text{permute-Tree}_\alpha \cdot \text{abs-eq} \left[ \text{eqvt, simp} \right]

\text{lemma} \text{alpha-Tree-permute-rep-commute} \left[ \text{simp} \right]: p \cdot \text{rep-Tree}_\alpha t_\alpha =_\alpha \text{rep-Tree}_\alpha (p \cdot t_\alpha)
5.3 Constructors for trees modulo $\alpha$-equivalence

The constructors are lifted from trees.

**lift-definition** $\text{Conj}_\alpha :: (\text{'idx}, \text{'pred}, \text{'act}) \rightarrow \text{Tree}_\alpha$ is $\text{tConj}$ by simp

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

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

**lift-definition** $\text{Not}_\alpha :: (\text{'pred}) \rightarrow \text{Tree}_\alpha$ is $\text{tNot}$ by simp

**lift-definition** $\text{Pred}_\alpha :: (\text{'pred}) \rightarrow \text{Tree}_\alpha$ is $\text{tPred}$

**lift-definition** $\text{Act}_\alpha :: (\text{'act}) \rightarrow \text{Tree}_\alpha \Rightarrow (\text{'idx}, \text{'pred}, \text{'act}) \rightarrow \text{Tree}_\alpha$ is $\text{tAct}$ by (fact $\alpha\cdot\text{Tree.-TAct}$)

The lifted constructors are equivariant.

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

**proof**

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

33
moreover

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

lemma Not α-eqvt [eqvt, simp]: p · Not α tα = Not α (p · tα)
by (induct tα) (simp add: Not α.abs-eq)

lemma Pred α-eqvt [eqvt, simp]: p · Pred α ϕ = Pred α (p · ϕ)
by (simp add: Pred α.abs-eq)

lemma Act α-eqvt [eqvt, simp]: p · Act α α tα = Act α (p · α) (p · tα)
by (induct tα) (simp add: Act α.abs-eq)
The lifted constructors are injective (except for Act α).

lemma Conj α-eq-iff [simp]: Conj α tset1 α = Conj α tset2 α ←→ tset1 α = tset2 α
proof
  assume Conj α tset1 α = Conj α tset2 α
  then have t Conj (map-bset rep-Tree α tset1 α) =α t Conj (map-bset rep-Tree α tset2 α)
    by (metis Conj α.def′ Tree α.abs-eq-iff)
  then have rel-bset (=α) (map-bset rep-Tree α tset1 α) (map-bset rep-Tree α tset2 α)
    by (auto elim: alpha-Tree.cases)
  then show tset1 α = tset2 α
    using Quotient-Tree α Quotient-rel-abs2 bset-lifting.bset-quot-map map-bset-abs-rep-Tree α
    by fastforce
qed (fact arg-cong)

lemma Not α-eq-iff [simp]: Not α t1 α = Not α t2 α ←→ t1 α = t2 α
proof
  assume Not α t1 α = Not α t2 α

34
then have \( \text{tNot} (\text{rep-Tree}_\alpha t1_\alpha) = \alpha \text{tNot} (\text{rep-Tree}_\alpha t2_\alpha) \)
by (metis \( \text{Not}_\alpha \text{abs-eq Tree}_\alpha.\text{abs-eq-iff Tree}_\alpha\text{-abs-rep} \))
then have \( \text{rep-Tree}_\alpha t1_\alpha = \alpha \text{rep-Tree}_\alpha t2_\alpha \)
using alpha-Tree.\cases by auto
then show \( t1_\alpha = t2_\alpha \)
by (metis Tree_\alpha.\text{abs-eq-iff Tree}_\alpha\text{-abs-rep} )

next
assume \( t1_\alpha = t2_\alpha \)
then show \( \text{Not}_\alpha t1_\alpha = \text{Not}_\alpha t2_\alpha \)
by simp
qed

lemma Pred_\alpha\text{-eq-iff} \[ simp \]: \( \text{Pred}_\alpha \varphi_1 = \text{Pred}_\alpha \varphi_2 \leftrightarrow \varphi_1 = \varphi_2 \)
proof
assume \( \text{Pred}_\alpha \varphi_1 = \text{Pred}_\alpha \varphi_2 \)
then have \( (t\text{Pred}_\varphi_1 :: (d, b, e) \text{Tree}) =_\alpha t\text{Pred}_\varphi_2 \) — note the unrelated type
by (metis \( \text{Pred}_\alpha \text{abs-eq Tree}_\alpha.\text{abs-eq-iff} \))
then show \( \varphi_1 = \varphi_2 \)
using alpha-Tree.\cases by auto
next
assume \( \varphi_1 = \varphi_2 \)
then show \( \text{Pred}_\alpha \varphi_1 = \text{Pred}_\alpha \varphi_2 \)
by simp
qed

lemma Act_\alpha\text{-eq-iff} : \( \text{Act}_\alpha \alpha t_1 = \text{Act}_\alpha \alpha t_2 \leftrightarrow t\text{Act}_\alpha \alpha t_1 =_\alpha t\text{Act}_\alpha \alpha t_2 \)
by (metis \( \text{Act}_\alpha \text{abs-eq Tree}_\alpha.\text{abs-eq-iff Tree}_\alpha\text{-abs-rep} \))

The lifted constructors are free (except for \( \text{Act}_\alpha \)).

lemma Tree_\alpha\text{-free} \[ simp \]:
shows \( \text{Conj}_\alpha t\text{set}_\alpha \neq \text{Not}_\alpha t_\alpha \)
and \( \text{Conj}_\alpha t\text{set}_\alpha \neq \text{Pred}_\alpha \varphi \)
and \( \text{Conj}_\alpha t\text{set}_\alpha \neq \text{Act}_\alpha \alpha t_\alpha \)
and \( \text{Not}_\alpha t_\alpha \neq \text{Pred}_\alpha \varphi \)
and \( \text{Not}_\alpha t_\alpha \neq \text{Act}_\alpha \alpha t_\alpha \)
and \( \text{Pred}_\alpha \varphi \neq \text{Act}_\alpha \alpha t_\alpha \)
by (simp add: Conj_\alpha\text{-def Not_\alpha\text{-def Pred_\alpha\text{-def Act_\alpha\text{-def Tree_\alpha\text{-abs-eq-iff}}}})+)

The following lemmas describe the support of constructed trees modulo \( \alpha \)-equivalence.

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

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

lemma supp-Not_\alpha \[ simp \]: \( \text{supp} (\text{Not}_\alpha t_\alpha) = \text{supp} t_\alpha \)
unfolding supp-def \by simp
lemma supp-Pred\(_\alpha\) [simp]: 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-eq Tree\(_\alpha\).rep-abs alpha-Tree-supp-rel supp-alpha-supp-rel supp-rel-tAct)

5.4 Induction over trees modulo \(\alpha\)-equivalence

lemma Tree\(_\alpha\)-induct [case-names Conj\(_\alpha\) Not\(_\alpha\) Pred\(_\alpha\) Act\(_\alpha\) Env\(_\alpha\), induct type: Tree\(_\alpha\)]:
  fixes t\(_\alpha\)
  assumes \(\Lambda tset\(_\alpha\). (\W x. x \in set-bset tset\(_\alpha\) \Longrightarrow P x) \Longrightarrow P (Conj\(_\alpha\) tset\(_\alpha\))\)
  and \(\Lambda t\(_\alpha\). P t\(_\alpha\) \Longrightarrow P (Not\(_\alpha\) t\(_\alpha\))\)
  and \(\Lambda pred. P (Pred\(_\alpha\) pred)\)
  and \(\Lambda 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-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 \(\alpha\)-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 \(\alpha\)-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.

\[
\text{inductive hereditarily-fs :: } (\text{'idx,'pred::fs,'act::bn}) \text{ Tree}_\alpha \Rightarrow \text{ bool where}
\]

\[
\begin{align*}
\text{Conj}_\alpha: \text{ finite (supp tset}_\alpha) \implies (\forall t_\alpha. t_\alpha \in \text{ set-bset tset}_\alpha \implies \text{ hereditarily-fs t}_\alpha) \\
\implies \text{ hereditarily-fs (Conj}_\alpha \text{ tset}_\alpha) \\
| \text{ Not}_\alpha: \text{ hereditarily-fs t}_\alpha \implies \text{ hereditarily-fs (Not}_\alpha t_\alpha) \\
| \text{ Pred}_\alpha: \text{ hereditarily-fs t}_\alpha \implies \text{ hereditarily-fs (Pred}_\alpha \varphi) \\
| \text{ Act}_\alpha: \text{ hereditarily-fs t}_\alpha \implies \text{ hereditarily-fs (Act}_\alpha \alpha t_\alpha)
\end{align*}
\]

hereditarily-fs is equivariant.

\textbf{lemma} hereditarily-fs-eqvt [eqvt]:
\begin{itemize}
\item \textbf{assumes} hereditarily-fs t_\alpha
\item \textbf{shows} hereditarily-fs (p \cdot t_\alpha)
\item \textbf{using} assms \textbf{proof} (induction rule: hereditarily-fs.induct)
\item \textbf{case Conj}_\alpha \textbf{ then show } ?case
\item \hspace{1em} 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)
\item \textbf{next}
\item \textbf{case Not}_\alpha \textbf{ then show } ?case
\item \hspace{1em} by (metis Not_\alpha-eqvt hereditarily-fs Perd_\alpha)
\item \textbf{next}
\item \textbf{case Pred}_\alpha \textbf{ then show } ?case
\item \hspace{1em} by (metis Pred_\alpha-eqvt hereditarily-fs Pred_\alpha)
\item \textbf{next}
\item \textbf{case Act}_\alpha \textbf{ then show } ?case
\item \hspace{1em} by (metis Act_\alpha-eqvt hereditarily-fs Act_\alpha)
\item \textbf{qed}
\end{itemize}

\text{lemma} hereditarily-fs-alpha-renaming:
\begin{itemize}
\item \textbf{assumes} Act_\alpha \alpha t_\alpha = Act_\alpha \alpha' t_\alpha'
\item \textbf{shows} hereditarily-fs t_\alpha \iff hereditarily-fs t_\alpha'
\item \textbf{proof}
\item \hspace{1em} \textbf{assume} hereditarily-fs t_\alpha
\item \hspace{1em} \textbf{then show} hereditarily-fs t_\alpha'
\item \hspace{1em} \textbf{using} assms \textbf{by} (auto simp add: Act_\alpha-def Tree_\alpha.abs-eq-iff alphas) \textbf{metis Tree}_\alpha.abs-eq-iff Tree_\alpha.abs-rep hereditarily-fs-eqvt permute-Tree_\alpha.abs-eq)
\item \textbf{next}
\item \hspace{1em} \textbf{assume} hereditarily-fs t_\alpha'
\item \hspace{1em} \textbf{then show} hereditarily-fs t_\alpha
\item \hspace{1em} \textbf{using} assms \textbf{by} (auto simp add: Act_\alpha-def Tree_\alpha.abs-eq-iff alphas) \textbf{metis Tree}_\alpha.abs-eq-iff Tree_\alpha.abs-rep hereditarily-fs-eqvt permute-Tree_\alpha.abs-eq permute-minus-cancel(2))
\item \textbf{qed}
\end{itemize}

Hereditarily finitely supported trees have finite support.
Lemma hereditarily-fs-implies-finite-supp:
  Assumes hereditarily-fs \( t_\alpha \)
  Shows finite \((\text{supp } t_\alpha)\)
  Using assms by (induction rule: hereditarily-fs.induct) (simp-all add: finite-supp)

5.6 Infinitary formulas

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

Typedef \( ('\text{id}', '\text{pred}': \text{fs}, '\text{act}': \text{bn}) \ \text{formula} = \{ t_\alpha::('\text{id}', '\text{pred}', '\text{act}) \ \text{Tree}_\alpha. \ \text{hereditarily-fs} \}
\) 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 \( \alpha \)-equivalence to the sub-type of formulas.

Setup-lifting type-definition-formula

Lemma Abs-formula-inverse [simp]:
  Assumes hereditarily-fs \( t_\alpha \)
  Shows \( \text{Rep-formula} (\text{Abs-formula } t_\alpha) = t_\alpha \)
  Using assms by (metis Abs-formula-inverse mem-Collect-eq)

Lemma Rep-formula' [simp]: hereditarily-fs \( \text{(Rep-formula } x) \)
by (metis Rep-formula mem-Collect-eq)

Now we lift the permutation operation.

Instantiation formula :: (type, \text{fs}, \text{bn}) pt
begin

Lift-definition permute-formula :: perm \( \Rightarrow ('\text{a}', '\text{b}', 'c) \ \text{formula} \Rightarrow ('\text{a}', '\text{b}', 'c) \ \text{formula} \)
is permute
by (fact hereditarily-fs-eqvt)

Instance
by standard (transfer, simp)+
end

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

Lemma Abs-formula-eqvt [simp]:
  Assumes hereditarily-fs \( t_\alpha \)
  Shows \( p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula} (p \cdot t_\alpha) \)
by (metis assms eq-onp-same-args permute-formula.abs-eq)

Lemma supp-Abs-formula [simp]:
  Assumes hereditarily-fs \( t_\alpha \)
  Shows \( \text{supp} (\text{Abs-formula } t_\alpha) = \text{supp } t_\alpha \)
proof -
{
  fix p :: perm
  have p · Abs-formula tα = Abs-formula (p · tα)
    using assms by (metis Abs-formula-eqvt)
  moreover have hereditarily-fs (p · tα)
    using assms by (metis hereditarily-fs-eqvt)
  ultimately have p · Abs-formula tα = Abs-formula tα ←→ p · tα = tα
    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 \( \forall x y. x \neq y \implies \text{map-bset Rep-formula} x \neq \text{map-bset Rep-formula} y \)
    by (metis inj-eq)
  then have \( \{ b. (a \equiv b) \cdot xset \neq xset\} \subseteq \{ b. (a \equiv b) \cdot \text{map-bset Rep-formula} xset \neq \text{map-bset Rep-formula} xset\} \) (is \(?S \subseteq ?T\))
    by auto
  then have infinite ?S ⇒ 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
  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 $\alpha$-equivalence) to infinitary formulas. Since $\text{Conj}_\alpha$ 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 \texttt{lift\_definition} to define $\text{Conj}$. Instead, theorems about terms of the form $\text{Conj}\ zset$ will usually carry an assumption that $xset$ is finitely supported.

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

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

\textbf{lemma} \texttt{finite-sup-implies-hereditarily-fs-Conj}_\alpha [simp]:

assumes finite (supp $xset$)

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

proof (rule hereditarily-fs $\text{Conj}_\alpha$)

show finite (supp (map-bset $\text{Rep-formula}\ xset$))

using \texttt{assms} by (metis supp-map-bset-Rep-formula)

next

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

then show hereditarily-fs $t_\alpha$

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

qed

\textbf{lemma} \texttt{Conj-rep-eq}:

assumes finite (supp $xset$)

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

using \texttt{assms} unfolding \texttt{Conj\_def} by simp

\textbf{lift-definition} \texttt{Not} :: (\cdot idx,\cdot pred,\cdot act)\ \text{formula}\ \Rightarrow(\cdot idx,\cdot pred::fs,\cdot act::bn)\ \text{formula}\ \text{is}

$\text{Not}_\alpha$

by (fact hereditarily-fs $\text{Not}_\alpha$)

\textbf{lift-definition} $\texttt{Pred} :: \cdot pred \Rightarrow(\cdot idx,\cdot pred::fs,\cdot act::bn)\ \text{formula}\ \text{is}$

$\text{Pred}_\alpha$

by (fact hereditarily-fs $\text{Pred}_\alpha$)

\textbf{lift-definition} $\texttt{Act} :: \cdot act \Rightarrow(\cdot idx,\cdot pred,\cdot act)\ \text{formula}\ \Rightarrow(\cdot idx,\cdot pred::fs,\cdot act::bn)$

\text{formula}\ \text{is}$

$\text{Act}_\alpha$

by (fact hereditarily-fs $\text{Act}_\alpha$)

The lifted constructors are equivariant (in the case of $\text{Conj}$, on finitely supported arguments).

\textbf{lemma} $\texttt{Conj-eqvt}$ [simp]:

assumes finite (supp $xset$)

shows $p \cdot \text{Conj}\ xset = \text{Conj}\ (p \cdot xset)$
using assms unfolding Conj-def by simp

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

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

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

The following lemmas describe the support of constructed formulas.

lemma supp-Conj [simp]:
assumes finite (supp xset)
sows 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 α 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)
sows Conj xset1 = Conj xset2 ←→ xset1 = xset2
using assms
by (metis (erased, hide-lams) Conj-α-eq-iff Conj-rep-eq Rep-formula-inverse injI inj-eq bset.inj-map)

lemma Not-eq-iff [simp]: Not x1 = Not x2 ←→ x1 = x2
by (metis Not.rep-eq Not-α-eq-iff Rep-formula-inverse)

lemma Pred-eq-iff [simp]: Pred ϕ1 = Pred ϕ2 ←→ ϕ1 = ϕ2
by (metis Pred.rep-eq Pred-α-eq-iff)

lemma Act-eq-iff: Act α1 x1 = Act α2 x2 ←→ Act_α α1 (Rep-formula x1) = Act_α 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 ←→
(∃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 obtain p where alpha: (bn α1, rep-Treeα Rew-formula x1)) ⇔ set (=α)
(supp-rel (=α)) p (bn α2, rep-Treeα (Rew-formula x2)) and eq: (bn α1, α1) ⇔ set (=) supp p (bn α2, α2)

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

ultimately show ?r

by (simp add: alpha-set.simps)

next

assume ?r

then 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-Treeα (Rew-formula x1)) ⇔ set (=α) (supp-rel (=α)) p (bn α2, rep-Treeα (Rew-formula x2))


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 α1 x1 = Act α2 x2

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

qed

lemma Act-eq-iff-perm-renaming: Act α1 x1 = Act α2 x2 ↔

(∃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 ∧ supp p ⊆ bn α1 ∪ p · bn α1)

(is ?l ↔ ?r)

proof

assume ?l
then obtain $p$ where $p : \text{supp } x_1 - \text{bn } \alpha_1 = \text{supp } x_2 - \text{bn } \alpha_2 \land (\text{supp } x_1 - \text{bn } \alpha_1) \neq p \land p \cdot x_1 = x_2 \land \text{supp } \alpha_1 - \text{bn } \alpha_1 = \text{supp } \alpha_2 - \text{bn } \alpha_2 \land (\text{supp } \alpha_1 - \text{bn } \alpha_1) \neq p \land p \cdot \alpha_1 = \alpha_2$

by (metis Act-eq-iff-perm)

moreover obtain $q$ where $q : p$ and $q \cdot b \in \text{bn } \alpha_1$. $q \cdot b = p \cdot b$ and $\text{supp } q : \text{supp } q \subseteq \text{bn } \alpha_1 \cup p \cdot \text{bn } \alpha_1$

by (metis set-renaming-perm2)

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

proof

fix $a$

assume $\ast : a \in \text{supp } q$

then show $a \in \text{supp } p$

proof (cases $a \in \text{bn } \alpha_1$)

case True
then show $a \in \text{supp } p$

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

next

case False
then have $a \in p \cdot \text{bn } \alpha_1$

using $\ast$ $q$-p

using UnE subsetCE by blast

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

by (metis mem-permute-iff)

then show $a \in \text{supp } p$

using fresh-def fresh-perm

by blast

qed

qed

with $p$

have $(\text{supp } x_1 - \text{bn } \alpha_1) \neq q$ and $(\text{supp } \alpha_1 - \text{bn } \alpha_1) \neq q$

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

moreover with $p$ and $q$-p have $\forall a. a \in \text{supp } \alpha_1 \Rightarrow q \cdot a = p \cdot a$ and $\forall a. a \in \text{supp } x_1 \Rightarrow q \cdot a = p \cdot a$

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

then have $q \cdot \alpha_1 = p \cdot \alpha_1$ and $q \cdot x_1 = p \cdot x_1$

by (metis supp-perm-perm-eq)

ultimately show $?r$

using $\ast$ $\ast$-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 $\text{Act}$).

lemma Tree-free [simp]:

shows finite $(\text{supp } xset) \Rightarrow \text{Conj } xset \neq \text{Not } x$

and finite $(\text{supp } xset) \Rightarrow \text{Conj } xset \neq \text{Pred } \varphi$

and finite $(\text{supp } xset) \Rightarrow \text{Conj } xset \neq \text{Act } \alpha$

and $\text{Not } x \neq \text{Pred } \varphi$

and $\text{Not } x \neq \text{Act } \alpha$

and $\text{Pred } \varphi \neq \text{Act } \alpha$

proof

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

by (metis Conj-rep-eq Not.rep-eq Tree\_\_free(1))

next

show finite $(\text{supp } xset) \Rightarrow \text{Conj } xset \neq \text{Pred } \varphi$
by (metis Conj-rep-eq Pred.rep-eq Tree\(\alpha\)-free(2))

next
show finite (supp xset) \implies \text{Conj} xset \neq \text{Act} \alpha x
by (metis Conjr-rep-eq Act.rep-eq Tree\(\alpha\)-free(3))

next
show \text{Not} x \neq \text{Pred} \varphi
by (metis Not.rep-eq Pred.rep-eq Tree\(\alpha\)-free(4))

next
show \text{Not} x1 \neq \text{Act} \alpha x2
by (metis Not.rep-eq Act.rep-eq Tree\(\alpha\)-free(5))

next
show \text{Pred} \varphi \neq \text{Act} \alpha x
by (metis Pred.rep-eq Act.rep-eq Tree\(\alpha\)-free(6))

qed

5.8 Induction over infinitary formulas

lemma formula-induct [case-names Conj Not Pred Act, induct type: formula]:
fixes x
assumes \(\forall xset. \text{finite} (\text{supp} xset) \implies (\forall x. x \in \text{set-bset} xset \implies P x) \implies P\)
(Conj xset)
and \(\forall \text{formula}. P \text{ formula} \implies P (\text{Not} \text{ formula})\)
and \(\forall \text{pred}. P \text{ pred} \implies P (\text{Act} \text{ pred formula})\)
shows \(P x\)
proof (induction x)
fix \(t_{\alpha} :: (\alpha, \beta, \gamma) \text{Tree}_{\alpha}\)
assume \(t_{\alpha} \in \{t_{\alpha}. \text{hereditarily-fs} t_{\alpha}\}\)
then have \(\text{hereditarily-fs} t_{\alpha}\)
by simp
then show \(P (\text{Abs-formula} t_{\alpha})\)
proof (induction \(t_{\alpha}\))
case (Conj\(\alpha \) tset\(\alpha\)) show ?case
proof =
let \(?tset = \text{map-bset} \text{Abs-formula} tset\(\alpha\)\nhave \(\forall t_{\alpha}', t_{\alpha}' \in \text{set-bset} tset\(\alpha\) \implies t_{\alpha}' = (\text{Rep-formula} \circ \text{Abs-formula}) t_{\alpha}'\)
by (simp add: Conj\(\alpha\).hyps)
then have \(tset\(\alpha) = \text{map-bset} (\text{Rep-formula} \circ \text{Abs-formula}) tset\(\alpha)\)
by (metis bset.map-cong0 bset.map-id apply)
then have \(*: tset\(\alpha) = \text{map-bset} \text{Rep-formula} ?tset\)
by (metis bset.map-comp)
then have \(\text{Abs-formula} (\text{Conj}_\alpha \text{ tset}_\alpha) = \text{Conj} ?tset\)
by (metis Conjr-def)
moreover from * have \(\text{finite} (\text{supp} ?tset)\)
using Conj\(\alpha\).hyps(1) by (metis supp-map-bset-Rep-formula)
moreover have \((\forall t. t \in \text{set-bset} ?tset \implies P t)\)
using Conj\(\alpha\).IH by (metis imageE map-bset.rep-eq)
ultimately show ?thesis

using assms(1) by metis
qed
next
case Not$_\alpha$ then show ?case
using assms(2) by (metis Formula.Abs-formula-inverse Not.rep-eq Rep-formula-inverse)
next
case Pred$_\alpha$ show ?case
using assms(3) by (metis Pred.abs-eq)
next
case Act$_\alpha$ then show ?case
using assms(4) by (metis Formula.Abs-formula-inverse Act.rep-eq Rep-formula-inverse)
qed

5.9 Strong induction over infinitary formulas

lemma formula-strong-induct-aux:
fixes x
assumes \( \forall xset c. \text{finite} (\text{supp} xset) \implies (\forall x c. x \in \text{set-bset} xset \implies (\forall c. P c x)) \)
implies P c (Conj xset)
and \( \forall \text{formula} c. (\forall c. P c \text{ formula}) \implies P c (\text{Not} \ \text{formula}) \)
and \( \forall \text{pred} c. P c (\text{Pred} \ \text{pred}) \)
and \( \forall \text{act} \ \text{formula} c. \text{bn act} \# c \implies (\forall c. P c \text{ formula}) \implies P c (\text{Act} \ \text{act} \ \text{formula}) \)
shows \( \forall (c :: \alpha) p. P c (p \cdot x) \)
proof (induction x)
case (Conj xset)
moreover then have finite (supp (p \cdot xset))
by (metis permute-finite supp-eqvt)
moreover have \( (\forall x c. x \in \text{set-bset} (p \cdot xset) \implies P c x) \)
using Conj.IH by (metis (full-types) eqvt-bound mem-permute-iff set-bset-eqvt)
ultimately show ?case
using assms(1) by simp
next
case Not then show ?case
using assms(2) by simp
next
case Pred show ?case
using assms(3) by simp
next
case (Act (p \cdot x)) show ?case
proof
— rename bn \((p \cdot \alpha)\) to avoid c, without touching Act \((p \cdot \alpha) (p \cdot x)\)
obtain q where 1: \((q \cdot \text{bn} (p \cdot \alpha)) \# c \) and 2: supp \((p \cdot \alpha) (p \cdot x)\) \# q
proof (rule at-set-avoiding2[of bn (p \cdot \alpha) c Act (p \cdot \alpha) (p \cdot x), THEN exE])
show finite \((bn (p \cdot \alpha))\) by (fact bn-finite)
next
show finite \((\text{supp} c)\) by (fact finite-supp)
next

45
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

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 ≡ {(x,y). x =α y}

lemma alpha-Tree-relI [simp];
  assumes x =α y shows (x,y) ∈ alpha-Tree-rel
using assms unfolding alpha-Tree-rel-def by simp

lemma alpha-Tree-relE;
  assumes (x,y) ∈ alpha-Tree-rel and x =α y ⇒ 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 :: ('d', 'e', 'f) Tree × ('d', 'e', 'f) Tree
  assume x ∈ (hull-rel O Tree-wf) O alpha-Tree-rel
  then obtain x₁ x₂ x₃ x₄ where 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 I and 3 proof (induct rule: Tree-wf.induct)
    — tConj
      fix t and tset :: ('d,'e,'f) Tree set[′d]
      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 =ₐ x₄
        by (rule alpha-Tree-relE)
      then obtain tset' where x₄: x₄ = tConj tset' and rel-bset (=ₐ) tset tset'
        by (cases x₄) simp-all
      with * obtain t' where t': t' ∈ set-bset tset' and t =ₐ t'
        by (metisrel-bset.rep-eq rel-set-def)
      with x₁ have (x₁, p · t') ∈ alpha-Tree-rel
        by (metis Treeₐ.abs-eq-iff alpha-Tree-relI permute-Treeₐ.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 =ₐ x₄
    by (rule alpha-Tree-relE)
  then obtain t' where x₄: x₄ = tNot t' and t =ₐ t'
    by (cases x₄) simp-all
  with x₁ have (x₁, p · t') ∈ alpha-Tree-rel
    by (metis Treeₐ.abs-eq-iff alpha-Tree-relI permute-Treeₐ.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
— $t\text{Act}$

fix $\alpha t$

assume $*$: $(x1,t) \in \text{hull-rel}$ and $**$: $(t\text{Act}\alpha t, x4) \in \alpha\text{-Tree-rel}$

from $*$ obtain $p$ where $x1: x1 = p \cdot t$

using $\text{hull-rel.cases}$ by $\text{blast}$

from $**$ have $t\text{Act}\alpha t =_\alpha x4$

by (rule $\alpha\text{-Tree-relE}$)

then obtain $q t'$ where $x4: x4 = t\text{Act}(q \cdot \alpha t')$ and $q \cdot t =_\alpha t'$

by (cases $x4$) (auto simp add: $\alpha\text{-set}$)

with $x1$ have $(x1, p \cdot q \cdot t') \in \alpha\text{-Tree-rel}$

by (metis $\alpha\text{-Tree-relI}$ $\text{permute-Tree}\alpha\text{-eq}$ $\text{permute-minus-cancel}(1)$)

moreover have $(p \cdot q \cdot t', t') \in \text{hull-rel}$

by (metis $\text{hull-rel.simps}$ $\text{permute-plus}$)

moreover from $x4$ have $(t', x4) \in \text{Tree-wf}$

by (simp add: $\text{Tree-wf.intros}(3)$)

ultimately show $(x1, x4) \in \alpha\text{-Tree-rel} \circ \text{hull-rel} \circ \text{Tree-wf}$

by auto

qed

with $x$ show $x \in \alpha\text{-Tree-rel} \circ \text{hull-rel} \circ \text{Tree-wf}$

by simp

qed

qed

lemma $\alpha\text{-Tree-rel-relcomp-trivialI}$ [simp]:

assumes $(x, y) \in R$

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

using assms unfolding $\alpha\text{-Tree-rel-def}$

by (metis $\text{Tree}\alpha\text{-eq-iff}$ $\text{case-prodI}$ $\text{mem-Collect-eq}$ $\text{relcompI}$ $\text{relcompI}$)

lemma $\alpha\text{-Tree-rel-relcompI}$ [simp]:

assumes $x =_\alpha x'$ and $(x', y) \in R$

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

using assms unfolding $\alpha\text{-Tree-rel-def}$

by (metis $\text{case-prodI}$ $\text{mem-Collect-eq}$ $\text{relcompI}$ $\text{relcompI}$)

6.1 Validity for infinitely branching trees

context $\text{nominal-ts}$

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 $\text{nominal_function}$ because that generates proof obli-
gations where, for formulas of the form $\text{Conj zset}$, the assumption that $zset$
has finite support is missing.

declare $\text{conj-cong}$ [fundef-cong]

function $\text{valid-Tree}$ :: $'\text{state} \Rightarrow (\text{idx, pred, act}) \text{Tree} \Rightarrow \text{bool}$ where

$\text{valid-Tree}\ P\ (1\text{Conj tset}) \iff (\forall t\in\text{set-bset}\ tset.\ \text{valid-Tree}\ P\ t)$
valid-Tree P (tNot t) \iff \neg valid-Tree P t
valid-Tree P (tPred \varphi) \iff P \vdash \varphi
valid-Tree P (tAct \alpha t) \iff (\exists \alpha' t' P'. tAct \alpha t =_{\alpha} tAct \alpha' t' \land P \rightarrow (\alpha' P'))
\land 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 uf ?R
  by (metis uf-alpha-Tree-rel-hull-rel-Tree-wf uf-inv-image)
next
fix P :: 'state and tset :: ('idx,'pred,'act) Tree set['idx] and t
assume t \in set-bset tset then show ((P, t), (P, tConj tset)) \in ?R
  by (simp add: Tree-uf.intros(1))
next
fix P :: 'state and t :: ('idx,'pred,'act) Tree
show ((P, t), (P, tNot t)) \in ?R
  by (simp add: Tree-uf.intros(2))
next
fix P1 P2 :: 'state and \alpha1 \alpha2 :: 'act and t1 t2 :: ('idx,'pred,'act) Tree
assume tAct \alpha1 t1 =_{\alpha} tAct \alpha2 t2
then obtain p where t2 =_{\alpha} p \cdot t1
  by (auto simp add: alphas) (metis alpha-Tree-symp sympE)
then show ((P2, t2), (P1, tAct \alpha1 t1)) \in ?R
  by (simp add: Tree-uf.intros(3))
}
qed

valid-Tree is equivariant.

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

\begin{proof}
\begin{enumerate}
  \item then show \texttt{valid-Tree P (tConj tset)}
    \end{enumerate}
  \end{proof}

\begin{lemma}
\begin{assumptions}
\end{assumptions}
\begin{shows}
\end{shows}
\begin{using}
\end{using}
\end{lemma}

$\alpha$-equivalent trees validate the same states.

\begin{lemma}
\begin{assumptions}
\end{assumptions}
\begin{shows}
\end{shows}
\begin{using}
\end{using}
\end{lemma}

50

\end{document}
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 Tree_α.abs-eq-iff valid-Tree.simps(4))
  qed

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 Tree_α.abs-eq-iff valid-Tree.simps(4))
  qed

qed simp-all

6.2 Validity for trees modulo α-equivalence

lift-definition valid-Tree_α :: 'state ⇒ ('idx,'pred,'act) Tree_α ⇒ bool is valid-Tree
by (fact alpha-Tree-valid-Tree)

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

lemma valid-Tree_α-Conj_α [simp]: valid-Tree_α P (Conj_α tset_α) ←→ (∀ tα∈set-bset tset_α. valid-Tree_α P tα)
  proof
    have valid-Tree P (rep-Tree_α (abs-Tree_α (tConj (map-bset rep-Tree_α tset_α))))
    ←→ valid-Tree P (tConj (map-bset rep-Tree_α tset_α))
      by (metis Tree_α.abs-eq-iff alpha-Tree-valid-Tree)
    then show ?thesis
      by (simp add: valid-Tree_α-def Conj_α-def map-bset.rep-eq)
  qed

lemma valid-Tree_α-Not_α [simp]: valid-Tree_α P (Not_α tα) ←→ ¬ valid-Tree_α P tα
  by transfer simp

lemma valid-Tree_α-Pred_α [simp]: valid-Tree_α P (Pred_α ϕ) ←→ P ⊨ ϕ

51
by transfer simp

lemma valid-Treeα-Act [simp]: valid-Treeα P (Actα α tα) ←→ (∃α' tα' P').
Actα α tα = Actα α' tα' ∧ P ←→ (α',P') ∧ valid-Treeα P' tα'

proof
  assume valid-Treeα P (Actα α tα)
  moreove have Actα α tα = abs-Treeα (tAct α (rep-Treeα tα))
  by (metis Actα.abs-eq Treeα.abs-rep)
  ultimately show ∃α' tα' P'. Actα α tα = Actα α' tα' ∧ P ←→ (α',P') ∧ valid-Treeα P' tα'
  by (metis Actα.abs-eq Treeα.abs-eq iff valid-Treeα.abs-eq)
next
  assume ∃α' tα' P'. Actα α tα = Actα α' tα' ∧ P ←→ (α',P') ∧ valid-Treeα P' tα'
moreove have ∃α' tα'. Actα α tα = abs-Treeα (tAct α (rep-Treeα tα'))
  by (metis Actα.abs-eq Treeα.abs-rep)
  ultimately show valid-Treeα P (Actα α tα)
  by (metis Treeα.abs-eq iff valid-Treeα.abs-eq valid-Treeα.rep-eq)
qed

6.3 Validity for infinitary formulas

lift-definition valid :: 'state ⇒ ('idx,'pred,'act) formula ⇒ bool (infix |= 70) is
valid-Treeα

lemma valid-eqvt :
  assumes P |= x shows (p • P) |= (p • x)
using assms by transfer (metis valid-Treeα-eqvt)

lemma valid-Conj [simp]:
  assumes finite (supp xset)
  shows P |= Conj xset ←→ (∀x ∈ set-bset xset. P |= x)
using assms by (simp add: valid-def Conj-def map-bset.rep-eq)

lemma valid-Not [simp]: P |= Not x ←→ P |= x
by transfer simp

lemma valid-Pred [simp]: P |= Pred φ ←→ P ⊨ φ
by transfer simp

lemma valid-Act: P |= Act α x ←→ (∃α' x' P'. Act α x = Act α' x' ∧ P ←→ (α',P') ∧ P' |= x')
proof
  assume P |= Act α x
  moreover have Rep-formula (Abs-formula (Actα α (Rep-formula x))) = Actα α (Rep-formula x)
  by (metis Act.rep-eq Rep-formula-inverse)
  ultimately show ∃α' x' P'. Act α x = Act α' x' ∧ P ←→ (α',P') ∧ P' |= x'

52
\[
\begin{align*}
\text{by (auto simp add: valid-def Act-def) (metis Abs-formula-inverse Rep-formula' hereditarily-fs-alpha-renaming)}
\end{align*}
\]

\text{next}
\begin{align*}
\text{assume } & \exists \alpha' x' P'. \text{ Act } \alpha x = \text{Act } \alpha' x' \land P \rightarrow \langle \alpha', P' \rangle \land P' \models x'
\end{align*}

\text{then show } P \models \text{Act } \alpha x
\begin{align*}
\text{by (metis Act.rep-eq valid.rep-eq valid.Tree}_{\alpha'}-\text{Act}_{\alpha})
\end{align*}

\text{qed}

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

\text{lemma valid-Act-strong:}
\begin{align*}
\text{assumes } & \text{finite } (\text{supp } X)
\end{align*}

\text{shows } P \models \text{Act } \alpha x \leftrightarrow (\exists \alpha' x' P'. \text{ Act } \alpha x = \text{Act } \alpha' x' \land P \rightarrow \langle \alpha', P' \rangle \land P' \models x' \land \text{bn } \alpha' \sharp X)
\begin{align*}
\text{proof}
\end{align*}

\text{assume } P \models \text{Act } \alpha x
\begin{align*}
\text{then obtain } & \alpha' x' P' \text{ where eq: Act } \alpha x = \text{Act } \alpha' x' \text{ and trans: } P \rightarrow \langle \alpha', P' \rangle
\end{align*}

\text{and valid: } P' \models x'
\begin{align*}
\text{by (metis valid-Act)}
\end{align*}

\text{have finite } (\text{bn } \alpha')
\begin{align*}
\text{by (fact bn-finite)}
\end{align*}

\text{moreover note } (\text{finite } (\text{supp } X))
\begin{align*}
\text{moreover have } & \text{finite } (\text{supp } (\text{Act } \alpha' x', \langle \alpha', P' \rangle))
\end{align*}

\text{by (metis finite-Diff finite-UnI finite-supp supp-abs-residual-pair)}
\begin{align*}
\text{moreover have } & \text{bn } \alpha' \sharp (\text{Act } \alpha' x', \langle \alpha', P' \rangle)
\end{align*}

\text{by (auto simp add: fresh-star-def fresh-def supp-Pair supp-abs-residual-pair)}
\begin{align*}
\text{ultimately obtain } & p \text{ where fresh-X: } (p \cdot \text{bn } \alpha') \sharp X \text{ and supp } (\text{Act } \alpha' x', \langle \alpha', P' \rangle) \sharp p
\end{align*}

\text{by (metis at-set-avoiding2)}
\begin{align*}
\text{then have } & \text{supp } (\text{Act } \alpha' x') \sharp p \text{ and supp } \langle \alpha', P' \rangle \sharp p
\end{align*}

\text{by (metis fresh-star-Un supp-Pair)+}
\begin{align*}
\text{then have } & \text{Act } (p \cdot \alpha) (p \cdot x') = \text{Act } \alpha' x' \text{ and } (p \cdot \alpha', p \cdot P') = \langle \alpha', P' \rangle
\end{align*}

\text{by (metis Act-eqv supp-perm-eq, metis abs-residual-pair-eqv supp-perm-eq)}
\begin{align*}
\text{then show } & \exists \alpha' x' P'. \text{ Act } \alpha x = \text{Act } \alpha' x' \land P \rightarrow \langle \alpha', P' \rangle \land P' \models x' \land \text{bn } \alpha' \sharp X
\end{align*}

\text{using eq and trans and valid and fresh-X by (metis bn-eqv valid-eqv)}
\begin{align*}
\text{next}
\end{align*}

\text{assume } \exists \alpha' x' P'. \text{ Act } \alpha x = \text{Act } \alpha' x' \land P \rightarrow \langle \alpha', P' \rangle \land P' \models x' \land \text{bn } \alpha' \sharp X
\begin{align*}
\text{then show } P \models \text{Act } \alpha x
\end{align*}

\text{by (metis valid-Act)}

\text{qed}

\text{lemma valid-Act-fresh:}
\begin{align*}
\text{assumes } & \text{bn } \alpha \sharp P
\end{align*}

\text{shows } P \models \text{Act } \alpha x \leftrightarrow (\exists P'. P \rightarrow \langle \alpha, P' \rangle \land P' \models x)
\begin{align*}
\text{proof}
\end{align*}

\text{assume } P \models \text{Act } \alpha x
moreover have finite \((\text{supp } P)\) 
by \text{(fact finite-sup)}
ultimately obtain \(\alpha' \ x' P'\) where
\(\text{eq}: \text{Act } \alpha \ x = \text{Act } \alpha' \ x'\) and \(\text{trans}: P \to \langle \alpha', P' \rangle\) and \(\text{valid}: P' \models x'\) and
\(\text{fresh}: \text{bn } \alpha' \sharp\ P\)
by \text{(metis valid-Act-strong)}

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

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

from \(\text{trans}\) have \(P \to \langle \alpha, -p \cdot P \rangle\)
using \(p\)-\(\text{P}\) \(p\)-\(\alpha\) by \text{(metis permute-minus-cancel(1) transition-eqvt)}
moreover from \(\text{valid}\) have \(-p \cdot P' \models x\)
using \(p\)-\(\text{x}\) by \text{(metis permute-minus-cancel(1) valid-eqvt)}
ultimately show \(\exists P'. P \to \langle \alpha, P' \rangle \land P' \models x\)
by \text{meson}
next
assume \(\exists P'. P \to \langle \alpha, P' \rangle \land P' \models x\) then show \(P \models \text{Act } \alpha \ x\)
by \text{(metis valid-Act)}
qed
end

end

theory Logical-Equivalence
imports
Validity
begin

7 (Strong) Logical Equivalence

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

locale indexed-nominal-ts = nominal-ts satisfies transition
for satisfies :: 'state::fs \Rightarrow 'pred::fs \Rightarrow bool (infix \oplus 70)
and transition :: 'state \Rightarrow ('act::bn, 'state) residual \Rightarrow bool (infix \rightarrow 70) +
assumes card-idx-perm: |UNIV::'perm set| <o |UNIV::'idx set|
and card-idx-state: |UNIV::'state set| <o |UNIV::'idx set|
begin
definition logically-equivalent :: 'state ⇒ '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. 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'

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 is-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 is-distinguishing-formula-eqvt :
    assumes x distinguishes P from Q shows (p ∘ x) distinguishes (p ∘ P) from (p ∘ Q)
  using assms unfolding is-distinguishing-formula-def
  by (metis permute-minus-cancel(2) valid-eqvt)

56
lemma equivalent-iff-not-distinguished: \((P =\cdot Q) \iff \neg (\exists x. x \text{ distinguishes } P \text{ from } Q)\)

by (metis (full-types) is-distinguishing-formula-def logically-equivalent-def valid-Not)

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

lemma distinguished-bounded-support:

assumes \(x \text{ distinguishes } P \text{ from } Q\)

obtains \(y\) where \(\text{supp } y \subseteq \text{supp } P\) and \(y \text{ distinguishes } P \text{ from } Q\)

proof

- let \(?B = \{p \cdot x | p. \text{ supp } P \ast* p\}\)

have \(\text{supp } P \text{ supports } ?B\)

unfolding supports-def proof (clarify)

fix \(a\) \(b\)

assume \(a: a \notin \text{supp } P\) and \(b: b \notin \text{supp } P\)

have \((a \iff b) \cdot ?B \subseteq ?B\)

proof

- fix \(x'\)

  assume \(x' \in (a \iff b) \cdot ?B\)

  then obtain \(p\) where \(1: x' = (a \iff b) \cdot p \cdot x\) and \(2: \text{ supp } P \ast* p\)

  by (auto simp add: permute-set-def)

  let \(?q = (a \iff b) + p\)

  from \(1\) have \(x' = ?q \cdot x\)

  by simp

  moreover from \(a\) and \(b\) and \(2\) have \(\text{ supp } P \ast* ?q\)

  by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))

ultimately show \(x' \in ?B\) by blast

qed

moreover have \(?B \subseteq (a \iff b) \cdot ?B\)

proof

- fix \(x'\)

  assume \(x' \in ?B\)

  then obtain \(p\) where \(1: x' = p \cdot x\) and \(2: \text{ supp } P \ast* p\)

  by auto

  let \(?q = (a \iff b) + p\)

  from \(1\) have \(x' = (a \iff b) \cdot ?q \cdot x\)

  by simp

  moreover from \(a\) and \(b\) and \(2\) have \(\text{ supp } P \ast* ?q\)

  by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))

ultimately show \(x' \in (a \iff b) \cdot ?B\)

using mem-permute-iff by blast

qed

ultimately show \((a \iff b) \cdot ?B = ?B\ ..\)

qed

then have \(\text{supp } B\)-subset-supp-P: \(\text{ supp } ?B \subseteq \text{ supp } P\)

by (metis (erased, lifting) finite-supp supp-is-subset)

then have finite-supp-B: \(\text{ finite } (\text{ supp } ?B)\)

using finite-supp rev-finite-subset by blast
have \( ?B \subseteq (\lambda p \cdot p \cdot x) \cdot \text{UNIV} \)
by auto
then have \( |?B| \leq o |\text{UNIV} :: \text{perm set}| \)
by (rule surj-imp-ordLeq)
also have \( |\text{UNIV} :: \text{perm set}| < o |\text{UNIV} :: \text{idx set}| \)
by (metis card-idx-perm)
also have \( |\text{UNIV} :: \text{idx set}| \leq o \text{natLeq} + c |\text{UNIV} :: \text{idx set}| \)
by (metis Cnotzero-UNIV ordLeq-csum2)
finally have \( |\text{UNIV} : \text{perm set}| < o \text{natLeq} + c |\text{UNIV} :: \text{idx set}| \)

let \( ?y = \text{Conj} (\text{Abs-bset} ?B) : (\text{\texttt{idx}}, \text{\texttt{pred}}, \text{\texttt{act}}) \text{ formula} \)
from finite-sup B and card-B and supp-B-subset-supP have supp ?y \( \subseteq \) supp P
by simp
moreover have \( ?y \text{ distinguishes } P \text{ from } Q \)
unfolding is-distinguishing-formula-def proof
from assms show \( P \models ?y \to Q \models \varphi \)
by (metis logically-equivalent-def valid-Pred)
next
from assms show \( \neg Q \models ?y \)
by (metis logically-equivalent-def permute-zero fresh-star-zero)
qed
ultimately show \( ?\text{thesis} \)
qed

lemma equivalence-is-bisimulation: is-bisimulation logically-equivalent
proof –
have symp logically-equivalent
by (metis logically-equivalent-def sympI)
moreover { fix \( P \, Q \, \varphi \) assume \( P = \cdot Q \) then have \( P \models \varphi \to Q \models \varphi \)
by (metis logically-equivalent-def valid-Pred)
}
moreover { fix \( P \, Q \, \alpha \) assume \( P = \cdot Q \) and \( \text{bn} \, \alpha \not\models Q \) and \( P \to \langle \alpha, P' \rangle \)
then have \( \exists Q'. \, Q \to \langle \alpha, Q' \rangle \land P' = \cdot Q' \)
proof –
{ let \( ?Q' = \{ Q'. \, Q \to \langle \alpha, Q' \rangle \} \)
assume \( \forall Q' \in ?Q'. \, P' = \cdot Q' \)
then have \( \forall Q' \in ?Q'. \, \exists x :: (\text{'idx}, \text{'pred}, \text{'act}) \text{ formula. } x \text{ distinguishes } P' \text{ from } Q' \)
by (metis equivalent-iff-not-distinguished)
then have \( \forall Q' \in ?Q'. \, \exists x :: (\text{'idx}, \text{'pred}, \text{'act}) \text{ formula. supp } x \subseteq \text{ supp } P' \land x \text{ distinguishes } P' \text{ from } Q' \)
by (metis distinguished-bounded-support)
}

58
then obtain \( f :: \text{'state} \Rightarrow (\text{'idx, 'pred, 'act}) \) formula where
\*: \( \forall Q' \in Q'. \, \text{supp} (f Q') \subseteq \text{supp} P' \land (f Q') \) distinguishes \( P' \) from \( Q' \)
by metis
have \( \text{supp} (f Q) \subseteq \text{supp} P' \)
by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)
then have finite-supp-image: \( \text{finite} (\text{supp} (f Q')) \)
using finite-supp rev-finite-subset by blast
have \( |f Q'| \leq |\text{UNIV} :: \text{'state set}| \)
by (metis card-of-UNIV card-of-image ordLeq-transitive)
also have \( |\text{UNIV} :: \text{'idx set}| < |\text{UNIV} :: \text{'state set}| \)
using finite-supp rev-finite-subset by blast
have \( \text{card} (\text{supp} (f Q')) \leq |\text{UNIV} :: \text{'idx set}| \)
by (metis Cnotzero-UNIV ordLeq-csum2)
finally have \( \text{card} (\text{supp} (f Q')) \leq |\text{UNIV} :: \text{'idx set}| \)
by (metis ordLeq-csum2)

let \( ?y = \text{Conj} (\text{Abs-bset} (f Q)) :: (\text{'idx, 'pred, 'act}) \) formula
have \( P \models \text{Act} \alpha \ ?y \)
unfolding valid-Act proof (standard+)
show \( P \rightarrow \langle \alpha, P' \rangle \) by fact
next
\{ fix \( Q' \)
assume \( Q \rightarrow \langle \alpha, Q' \rangle \)
with \* have \( P' \models f Q' \)
by (metis is-distinguishing-formula-def mem-Collect-eq)
\}
then show \( P' \models ?y \)
by (simp add: finite-supp-image card-image)
qed
moreover have \( \neg Q \models \text{Act} \alpha \ ?y \)
proof
assume \( Q \models \text{Act} \alpha \ ?y \)
then obtain \( Q' \) where 1: \( Q \rightarrow \langle \alpha, Q' \rangle \) and 2: \( Q' \models ?y \)
using \( \text{bn} \ \alpha \ ?y \) \( Q \) by (metis valid-Act-fresh)
from 2 have \( \forall Q''. Q \rightarrow (\alpha, Q'') \rightarrow Q' \models f Q'' \)
by (simp add: finite-supp-image card-image)
with 1 and \* show False
using is-distinguishing-formula-def by blast
qed
ultimately have False
by (metis \( \langle P = Q \rangle \) logically-equivalent-def)
\}
then show \( \text{thesis} \) by auto
qed
}\)
ultimately show \( \text{thesis} \)
unfolding is-bisimulation-def by metis
qed

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

end

end

theory Disjunction

imports
  Formula
  Validity

begin

10 Disjunction

definition Disj :: ('idx::fs, 'act::bn) formula set['idx] ⇒ ('idx::fs, 'act) formula where
  Disj xset = Not (Conj (map-bset Not xset))

lemma finite-sup-p-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-sup-p-map-bset-Not)

context nominal-ts

begin

lemma valid-Disj [simp]:
  assumes finite (supp xset)
  shows P □ Disj xset ⟷ (∃ x∈set-bset xset. P □ x)

using assms by (simp add: Disj-def map-bset.rep-eq)
end

end

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

11 Expressive Completeness

context indexed-nominal-ts
begin

11.1 Distinguishing formulas

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

Note that this definition uses Hilbert’s choice operator $\varepsilon$, which is not necessarily equivariant. This is immediately remedied by a hull construction.

definition distinguishing-formula :: 'state ⇒ 'state ⇒ ('idx, 'pred, 'act) formula
where
distinguishing-formula P Q ≡ Conj (Abs-bset \{-p · (ε x. supp x ⊆ supp (p · P)) ∧ x distinguishes (p · P) from (p · Q))\}|p. True})

— just an auxiliary lemma that will be useful further below
lemma distinguishing-formula-card-aux:
  \{|\{-p · (ε x. supp x ⊆ supp (p · P)) ∧ x distinguishes (p · P) from (p · Q))\}|p. True} ≤o natLeq +c |UNIV :: 'idx set|
proof
  let ?some = λp. (ε x. supp x ⊆ supp (p · P)) ∧ x distinguishes (p · P) from (p · Q)
  let ?B = \{-p · ?some p|p. True\}
  have ?B ⊆ (λp. -p · ?some p) · 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 show ?thesis .

61
qed

— just an auxiliary lemma that will be useful further below

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

  assumes \( \neg (P \cong Q) \)
  shows \( \supp (\Abs-bset \{-p \cdot (\epsilon x. \supp x \subseteq \supp (p \cdot P) \land x \text{ distinguishes } (p \cdot P) \} \cdot p. \True) :: set[^{idx}] \subseteq \supp P \)

  proof –
  let \( \some = \lambda p. (\epsilon x. \supp x \subseteq \supp (p \cdot P) \land x \text{ distinguishes } (p \cdot P) \) from \( (p \cdot Q) \)\)
  let \( \some p \cdot \True \)

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

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

  from \( \supp-B \) and distinguishing-formula-card-aux show \( \text{?thesis using supp-Abs-bset by blast} \)

qed

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

  assumes \( \neg (P \cong Q) \)
  shows \( p \cdot \text{distinguishing-formula } P \cdot Q = \text{distinguishing-formula } (p \cdot P) \cdot (p \cdot Q) \)

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

  from assms have \( \supp (\Abs-bset ?B :: \cdot \text{set[^{idx}]}) \subseteq \supp P \)

62
by (rule distinguishing-formula-supp-aux)
then have finite (supp (Abs-bset ?B :: set[\'idx\'])))
using finite-supp rev-finite-subset by blast
with distinguishing-formula-card-aux have *: p · Conj (Abs-bset ?B) = Conj
(Abs-bset (p · ?B))
by simp

let ?some' = \lambda p'. (\epsilon x. supp x \subseteq supp (p' · p · P) \wedge x distinguishes (p' · p · P)) from (p' · p · Q))
let ?B' = \{-p' · ?some' p[p']. True\}

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

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

lemma supp-distinguishing-formula:
assumes \neg (P =\cdot Q)
shows supp (distinguishing-formula P Q) \subseteq supp P

63
proof
  let ?some = \( \lambda p. (\epsilon x. \text{supp } x \subseteq \text{supp } (p \cdot P) \land x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q)) \)
  let ?B = \{ - p \cdot ?some p | p. True \}

  from assms have \( \text{supp } (\text{Abs-bset } ?B :: - \text{set}[\text{idx}]) \subseteq \text{supp } P \)
  by (rule distinguishing-formula-supp-aux)

  moreover from this have \( \text{finite } \) (\( \text{supp } (\text{Abs-bset } ?B :: - \text{set}[\text{idx}]) \))
  using finite-supp rev-finite-subset by blast

  ultimately show ?thesis
    unfolding distinguishing-formula-def by simp
  qed

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

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

  note some-distinguishes = this

  \{
    fix \( P' \)
    from assms have \( \text{supp } (\text{Abs-bset } ?B :: - \text{set}[\text{idx}]) \subseteq \text{supp } P \)
    by (rule distinguishing-formula-supp-aux)

    then have \( \text{finite } \) (\( \text{supp } (\text{Abs-bset } ?B :: - \text{set}[\text{idx}]) \))
    using finite-supp rev-finite-subset by blast

    with distinguishing-formula-card-aux have \( P' \models \text{distinguishing-formula } P Q \)
    \( \iff (\forall x \in \text{?B}. P' \models x) \)
    unfolding distinguishing-formula-def by simp
  \}

  note valid-distinguishing-formula = this

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

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

moreover have \( \neg Q \models distinguishing-formula P \ Q \)

ultimately show (distinguishing-formula P \ Q) distinguishes P from Q

11.2 Characteristic formulas

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

definition characteristic-formula :: 'state ⇒ ('idx, 'pred, 'act) formula where
characteristic-formula P ≡ Conj (Abs-bset {distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)})

— just an auxiliary lemma that will be useful further below

lemma characteristic-formula-card-aux:
\[
|\{distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)}| < o \ natLeq + c \ \ |UNIV :: 'idx set|
\]

proof −
let \(?B = \{distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)}\)

have \(?B \subseteq (distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)}\) \cdot UNIV

by auto

then have |?B| ≤ o |UNIV :: 'state set|

by (rule surj-imp-ordLeq)

also have |UNIV :: 'state set| < o |UNIV :: 'idx set|

by (metis card-idx-state)

also have |UNIV :: 'idx set| ≤ o \ natLeq + c \ |UNIV :: 'idx set|

by (metis Cnotzero-UNIV ordLeq-csum2)

finally show \(?thesis\).

qed

— just an auxiliary lemma that will be useful further below

lemma characteristic-formula-supp-aux:

shows supp (Abs-bset {distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)} :: - set[\'idx|]) ⊆ supp P

proof −

let \(?B = \{distinguishing-formula P \ Q\ | \ Q. \ (P =\cdot \ Q)}\)

\{

fix \(x\)

assume \(x \in \ ?B\)

then obtain \(Q\) where \(x = distinguishing-formula P \ Q\) and \(\neg (P =\cdot \ Q)\)

by blast

with supp-distinguishing-formula have supp \(x \subseteq supp \ P\)

by metis

65
note * = this
have supp-B: supp ?B ⊆ supp P
  by (rule set-bounded-sup, fact finite-sup, cut-tac *, blast)

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

qed

lemma characteristic-formula-eqvt [simp]:
  \( p \cdot \text{characteristic-formula } P = \text{characteristic-formula } (p \cdot P) \)

proof -
  let \( ?B = \{ \text{distinguishing-formula } P Q | Q, \neg (P = \cdot Q) \} \)

  have supp (Abs-bset ?B :: set[\( \text{idx} \)]) ⊆ supp P
    by (fact characteristic-formula-sup-aux)
  then have finite (supp (Abs-bset ?B :: set[\( \text{idx} \)]))
    using finite-sup rev-finite-subset by blast
  with characteristic-formula-card-aux have *: p · Conj (Abs-bset ?B) = Conj (Abs-bset (p · ?B))
    by simp

  let ?B’ = \{ \text{distinguishing-formula } (p · P) Q | Q, \neg ((p · P) = \cdot Q) \}

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

66
by blast
from 2 have \( \neg (P \Rightarrow (-p \cdot Q)) \)
  by (metis logically-equivalent-eqvt permute-minus-cancel(1))
moreover from this and 1 have \( x = p \cdot \text{distinguishing-formula} P (-p \cdot Q) \)
  by simp
ultimately have \( x \in p \cdot ?B \)
  using mem-permute-iff by blast
} then show \( ?B' \subseteq p \cdot ?B \)
  by blast
qed

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

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

lemma characteristic-formula-is-characteristic':
  \( Q \models \text{characteristic-formula} P \iff P = \cdot Q \)
proof -
  let \( ?B = \{ \text{distinguishing-formula} P Q|Q. \neg (P = \cdot Q) \}\)
  {
    fix \( P' \)
    have supp (Abs-bset ?B :: - set[\'idx\]) \(\subseteq\) supp \( P \)
      by (fact characteristic-formula-supp-aux)
    then have finite (supp (Abs-bset ?B :: - set[\'idx\]))
      using finite-supp rev-finite-subset by blast
    with characteristic-formula-card-aux have \( P' \models \text{characteristic-formula} P \)
      \(\iff\) (\( \forall x\in?B. P' \models x \) )
      unfolding characteristic-formula-def by simp
  }
  note valid-characteristic-formula = this

show ?thesis
proof
  assume *: \( Q \models \text{characteristic-formula} P \)
  show \( P = \cdot Q \)
  proof (rule ccontr)
    assume \( \neg (P = \cdot Q) \)
    with * show False
      using distinguishing-formula-distinguishes is-distinguishing-formula-def
valid-characteristic-formula by auto
  qed
next
assume $P = Q$
moreover have $P \models \text{characteristic-formula } P$
  using distinguishing-formula-distinguishes is-distinguishing-formula-def
valid-characteristic-formula by auto
ultimately show $Q \models \text{characteristic-formula } P$
  using logically-equivalent-def by blast
qed
qed

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

11.3 Expressive completeness

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

theorem expressive-completeness:
  assumes finite $(\supp S)$
  and $\forall P. P \in S \Longrightarrow P \sim Q \Longrightarrow Q \in S$
  shows $P \models \text{Disj } (\text{Abs-bset } (\text{characteristic-formula } ' S)) \iff P \in S$
proof
  let $?B = \text{characteristic-formula } ' S$
  have $?B \subseteq \text{characteristic-formula } ' \text{UNIV}$
    by auto
  then have $|?B| \leq |\text{UNIV} :: '\text{state set}|$
    by (rule surj-imp-ordLeq)
  also have $|\text{UNIV} :: '\text{state set}| < o |\text{UNIV} :: '\text{idx set}|$
    by (metis card-idx-state)
  also have $|\text{UNIV} :: '\text{idx set}| \leq o n\text{atLeq} + c |\text{UNIV} :: '\text{idx set}|$
    by (metis Cnotzero-UNIV ordLeq-csum2)
  finally have $\text{card-B} : |?B| < o n\text{atLeq} + c |\text{UNIV} :: '\text{idx set}|$. 
  have eqvt image and eqvt characteristic-formula
    by (simp add: eqvtI)+
  then have supp $?B \subseteq \supp S$
    using supp-fun-eqvt supp-fun-app supp-fun-app-eqvt by blast
  with card-?B have supp (Abs-bset $?B :: - set['idx]) \subseteq \supp S$
    using supp-Abs-bset by blast
  with $(\text{finite } (\supp S))$ have finite $(\supp (\text{Abs-bset } ?B :: - set['idx]))$
    using finite-supp rev-finite-subset by blast
  with card-?B have $P \models \text{Disj } (\text{Abs-bset } (\text{characteristic-formula } ' S)) \iff (\exists x \in ?B. P \models x)$
    by simp
  with $\forall P Q. P \in S \Longrightarrow P \sim Q \Longrightarrow Q \in S$ show ?thesis
    using characteristic-formula-is-characteristic characteristic-formula-is-characteristic'
12 Finitely Supported Sets

We define the type of finitely supported sets (over some permutation type 'a).
Note that we cannot more generally define the (sub-)type of finitely sup-
ported 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

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

13.1 Nominal transition systems with effects

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

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 \( \text{first} \)) 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

13.2 L-bisimulations and F/L-bisimilarity

context effect-nominal-ts
begin

definition is-L-bisimulation:: ('effect first ⇒ 'state ⇒ 'state ⇒ bool) ⇒ bool
where
is-L-bisimulation R ≡
  ∀ F. symp (R F) ∧
  (∀ P Q. R F P Q → (∀ f ∈_fs F → (∀ ϕ. ⟨f⟩P ⊢ ϕ → ⟨f⟩Q ⊢ ϕ))) ∧
  (∀ P Q. R F P Q → (∀ 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 · R) F) (is ?S)
    using assms unfolding is-L-bisimulation-def by (metis eqvt-lambda symp-eqvt)
moreover have \( \forall P, Q. \ (p \cdot R) F P Q \rightarrow (\forall f. f \in_{fs} F \rightarrow (\forall \varphi. (f)P \vdash \varphi) \rightarrow (f)Q \vdash \varphi) \) (is ?T)

proof (clarify)

fix \( PQ \) \( f \varphi \)
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 \( R (\neg p \cdot F) (\neg p \cdot P) (\neg p \cdot Q) \)
  by (simp add: eqvt-lambda permute-bool-def unpermute-def)
moreover have \( (\neg p \cdot f) \in_{fs} (\neg p \cdot F) \)
  using effect by simp
moreover have \( (\neg p \cdot f)(\neg p \cdot P) \vdash \neg p \cdot \varphi \)
  using satisfies by (metis effect-apply-eqvt satisfies-eqvt)
ultimately have \( (\neg p \cdot f)(\neg p \cdot Q) \vdash \neg p \cdot \varphi \)
  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 \( \forall P, Q. \ (p \cdot R) F P Q \rightarrow (\forall f. f \in_{fs} F \rightarrow (\forall \alpha. P, bn \alpha \sharp^* \langle f \rangle Q, F, f \rightarrow (f)P \rightarrow \langle \alpha,P \rangle \rightarrow (\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha,Q \rangle \land (p \cdot R) (L (\alpha, F, f)) (P' Q')) ) \) (is ?U)

proof (clarify)

fix \( PQ \) \( f \alpha P' \)
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: \( (f)P \rightarrow \langle \alpha,P \rangle \)
from \( pR \) have \( R (\neg p \cdot F) (\neg p \cdot P) (\neg p \cdot Q) \)
  by (simp add: eqvt-lambda permute-bool-def unpermute-def)
moreover have \( (\neg p \cdot f) \in_{fs} (\neg p \cdot F) \)
  using effect by simp
moreover have \( bn (\neg p \cdot \alpha) \sharp^* (\neg p \cdot f)(\neg p \cdot Q), (\neg p \cdot F, (\neg p \cdot f)) \)
  using fresh by (metis (full-types) effect-apply-eqvt' bn-eqvt fresh-star-Pair fresh-star-permute-iff)
moreover have \( (\neg p \cdot f)(\neg p \cdot P) \rightarrow (\neg p \cdot \alpha, (\neg p \cdot P') \)
  using trans by (metis effect-apply-eqvt' transition-eqvt)
ultimately obtain \( Q' \) where \( T: (\neg p \cdot f)(\neg p \cdot Q) \rightarrow (\neg p \cdot \alpha, Q') \) and \( R: (L (\neg p \cdot \alpha, (\neg p \cdot F, (\neg p \cdot f))) (\neg p \cdot P') Q' \)
  using assms unfolding is-L-bisimulation-def by meson
from \( T \) have \( \langle f \rangle Q \rightarrow (\alpha, P') \)
  by (metis (no-types, lifting) effect-apply-eqvt' abs-residual-pair-eqvt permute-minus-cancel(1) transition-eqvt)
moreover from \( R \) have \( (p \cdot R) (p \cdot L (\neg p \cdot \alpha, (\neg p \cdot F, (\neg p \cdot f))) (p \cdot p' (\neg p \cdot P')) (\neg p \cdot Q')) \)
  by (metis permute-bolI 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 \( \exists Q'. \ (f)Q \rightarrow \langle \alpha, Q' \rangle \land (p \cdot R) (L (\alpha, F, f)) P' Q' \)
  by metis
qed
lemma FL-bisimilar-eqvt:
assumes \( P \sim_{\sim \cdot} [F] Q \)
shows \( (p \cdot P) \sim_{\sim \cdot} [p \cdot F] (p \cdot Q) \)
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 \((\lambda \cdot (=))\)
    unfolding is-L-bisimulation-def by (simp add: symp-def)
  then show \( x \sim_{\sim \cdot} [F] x \)
    unfolding FL-bisimilar-def by auto
qed

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

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

lemma FL-bisimilar-simulation-step:
  assumes \( P \sim_{\sim \cdot} [F] Q \) and \( f \in_{fs} F \) and \( bn \alpha \exists^* ((f)Q, F, f) \) and \( (f)P \rightarrow \langle \alpha, P' \rangle \)
  obtains \( Q' \) where \( (f)Q \rightarrow \langle \alpha, Q' \rangle \) and \( P' \sim_{\sim \cdot} [L (\alpha, F, f)] Q' \)
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 F Q and QR: Q \sim F R
  let \( ?FL-bisim = \lambda F. (FL-bisimilar F) OO (FL-bisimilar F) \)
  have \(~ F. symp (\( ?FL-bisim F \))\)
  proof (rule sympI)
    fix F P R
    assume \( ?FL-bisim F P R \)
    then obtain Q where P \sim F Q and Q \sim F R
    by blast
    then have R \sim F P
    by (metis FL-bisimilar-symp sympE)
    moreover have \( \forall f. (f) P \Rightarrow (\forall \varphi. \langle f \rangle P \vdash \varphi \Rightarrow (\forall f. (f) Q \vdash \varphi)) \)
    using FL-bisimilar-is-L-bisimulation is-L-bisimulation-def by auto
    qed (clarify)

    moreover have \(~ F. \forall P Q. ?FL-bisim F P Q \Rightarrow (\forall f. f \in F \Rightarrow F \Rightarrow (\forall f. f \vdash \varphi)) \)
    using FL-bisimilar-is-L-bisimulation is-L-bisimulation-def by auto
  qed

  proof (rule at-set-avoiding2) of bn α ((\( f \)) 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 ((\( f \)) 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 (\( p \cdot \alpha, p \cdot P' \)) = (\( \alpha, P' \))
  using supp-perm-eq by fastforce
  then obtain p R' where 5: (\( f \)) R \Rightarrow (\( p \cdot \alpha, p R' \)) and 6: (\( p \cdot P' \) \sim [L (\( p \cdot \alpha, F, f) \)) P R'

  qed
using PR effect trans 1 by (metis FL-bisimilar-simulation-step bn-eqvt)
from fresh and 4 have bn \((p \cdot \alpha) \not\sim (f)Q, F, f)\)
by (metis bn-eqvt fresh-star-permute-iff supp-perm-eq)
then obtain \(pQ'\) where \(7: (f)Q \rightarrow (p \cdot \alpha, pQ')\) and \(8: pR' \sim \{L (p \cdot \alpha, F, f)\}\)
using RQ effect 5 by (metis FL-bisimilar-simulation-step)
from 4 have supp \(((f)Q) \not\sim p\)
by (simp add: fresh-star-Un supp-Pair)
with \(7\) have \((f)Q \rightarrow \{\alpha, p \cdot pQ'\}\)
by (metis relcompp.relcompI)
moreover from \(6\) and \(8\) have ?FL-bisim \((L (p \cdot \alpha, F, f)) (p \cdot P') pQ'\)
by (metis FL-bisimilar-eqvt)
than have ?FL-bisim \((p \cdot L (p \cdot \alpha, F, f)) (p \cdot p) (p \cdot pQ')\)
using FL-bisimilar-eqvt by blast
then have ?FL-bisim \((L (\alpha, p \cdot F, f)) P' (p \cdot pQ')\)
by (simp add: L-eqvI)
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'. (f)Q \rightarrow \{\alpha, Q'\} \land ?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 \((L \cdot F) R P\)

using \(PQ QR\) by blast

ultimately show \(P \sim \{F\} R\)

unfolding FL-bisimilar-def by meson

qed

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

end

theory FL-Formal
imports
Nominal-Bounded-Set
Nominal-Wellfounded
Residual
FL-Transition-System
begin
14 Infinitary Formulas With Effects

14.1 Infinitely branching trees

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

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

```plaintext
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.

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

lemma wf-Tree-wf: wf Tree-wf
unfolding wf-def
proof (rule allI, rule impI, rule allI)
  fix P :: ('idx,'pred,'act,'eff) Tree ⇒ bool and t
  assume ∀ x. (∀ y. (y, x) ∈ Tree-wf ⇒ P y) ⇒ P x
  then show P t
  proof (induction t)
    case tConj then show ?case
    by (metis Tree.distinct(2) Tree.distinct(5) Tree.inject(1) Tree-wf.cases)
  next
    case tNot then show ?case
    by (metis Tree.distinct(1) Tree.distinct(9) Tree.inject(2) Tree-wf.cases)
  next
    case tPred then show ?case
    by (metis Tree.distinct(11) Tree.distinct(3) Tree.distinct(7) Tree-wf.cases)
  next
    case tAct then show ?case
    by (metis Tree.distinct(10) Tree.distinct(6) Tree.inject(4) Tree-wf.cases)
  qed
```

qed
We define a permutation operation on the type of trees.

**instantiation**

\[
\text{Tree :: } (\text{type, pt, pt, pt}) \text{ pt}
\]

**begin**

\[\text{primrec permute-Tree :: perm \Rightarrow } (-,-,-,-) \text{ Tree \Rightarrow } (-,-,-,-) \text{ Tree where}\]

\[p \cdot (tConj tset) = tConj (\text{map-bset (permute p) tset})\]

— neat trick to get around the fact that \(tset\) is not of permutation type yet

\[| p \cdot (tNot t) = tNot (p \cdot t)\]

\[| p \cdot (tPred f \varphi) = tPred (p \cdot f) (p \cdot \varphi)\]

\[| p \cdot (tAct f \alpha t) = tAct (p \cdot f) (p \cdot \alpha) (p \cdot t)\]

**instance**

**proof**

\[\begin{align*}
\text{fix } t &: (-,-,-,-) \text{ Tree} \\
\text{show } 0 \cdot t = t \\
\text{proof (induction } t) \\
\text{case } tConj & \text{ then show } ?\text{case } \\
\text{by (simp, transfer)} (\text{auto simp: image-def}) \\
\text{qed simp-all}
\end{align*}\]

**next**

\[\begin{align*}
\text{fix } p, q &: \text{perm and } t &: (-,-,-,-) \text{ Tree} \\
\text{show } (p + q) \cdot t = p \cdot q \cdot t \\
\text{proof (induction } t) \\
\text{case } tConj & \text{ then show } ?\text{case } \\
\text{by (simp, transfer)} (\text{auto simp: image-def}) \\
\text{qed simp-all}
\end{align*}\]

**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**

\[\text{permute-Tree-tConj [simp]: p \cdot tConj tset = tConj (p \cdot tset)}\]

by (simp add: map-bset-permute)

**declare**

\[\text{permute-Tree.simps(1) [simp del]}\]

The relation \(\text{Tree-wf}\) is equivariant.

**lemma** \(\text{Tree-wf-eqvt-aux:}\)

**assumes** \((t1, t2) \in \text{Tree-wf}\) **shows** \((p \cdot t1, p \cdot t2) \in \text{Tree-wf}\)

**using** \(\text{assms proof (induction rule: Tree-wf.induct)}\)

**fix** \(t :: ('a,'b,'c,'d) \text{ Tree and } tset :: ('a,'b,'c,'d) \text{ Tree set['a]}\)

**assume** \(t \in \text{set-bset tset}\) **then show** \((p \cdot t, p \cdot tConj tset) \in \text{Tree-wf}\)

by (metis \text{Tree-wf.intros(1) mem-permute-iff permute-Tree-tConj set-bset-eqvt})

**next**

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

**show** \((p \cdot t, p \cdot tNot t) \in \text{Tree-wf}\)
by (metis Tree-wf.intros(2) permute-Tree.simps(2))

next

fix t :: ('a,'b,'c,'d) Tree and f and α

show (p · t, p · tAct f α t) ∈ Tree-wf

by (metis Tree-wf.intros(3) permute-Tree.simps(4))

qed

lemma Tree-wf-eqvt [eqvt, simp]: p · Tree-wf = Tree-wf

proof

show p · Tree-wf ⊆ Tree-wf

by (auto simp add: permute-set-def) (rule Tree-wf-eqvt-aux)

next

show Tree-wf ⊆ p · 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 ∪ 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 ∪ supp α ∪ supp t

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

14.2 Trees modulo α-equivalence

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

lemma supp-rel-eqvt [eqvt]:

p · supp-rel R x = supp-rel (p · R) (p · x)

by (simp add: supp-rel-def)

Usually, the definition of α-equivalence presupposes a notion of free variables. However, the variables that are “free” in an infinitary conjunction are not necessarily those that are free in one of the conjuncts. For instance, consider a conjunction over all names. Applying any permutation will yield the same conjunction, i.e., this conjunction has no free variables.
To obtain the correct notion of free variables for infinitary conjunctions, we initially defined α-equivalence and free variables via mutual recursion. In particular, we defined the free variables of a conjunction as term \( \text{fv-Tree} \) \((\text{tConj tset})\) = \( \text{supp-rel} \) \( \alpha \)-Tree \((\text{tConj tset})\).

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** \( \text{supp-rel-cong} \) [fundef-cong]:

\[
\implies \quad \text{by (simp add: supp-rel-def)}
\]

**Lemma** \( \text{rel-bset-cong} \) [fundef-cong]:

\[
\implies \quad \text{by (simp add: rel-bset-def rel-set-def)}
\]

**Lemma** \( \text{alpha-set-cong} \) [fundef-cong]:

\[
\implies \quad \text{by (simp add: alpha-set)}
\]

**Quotient-type**

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

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

**Lemma** \( \text{permute-rep-abs-Tree}_p\):

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

**Lemma** \( \text{Tree-wf}_p\) :: \((\text{idx}, \text{pred}::\text{pt}, \text{act}::\text{bn}, \text{eff}::\text{fs})\) \(\text{Tree}_p \) rel is \(\text{Tree-wf}\).

**Lemma** \( \text{Tree-wf}_p\)-I [simp]:

**Assumes** \((a, b) \in \text{Tree-wf}\)

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

**Using asserts** **by** (metis (erased, lifting) \(\text{Tree}_p\), \(\text{abs-eq-iff}\) \(\text{Tree-wf}_p\), \(\text{abs-eq} \) \(\text{hull-relp}\).\(\text{intros}\) \(\text{map-prod-simp}\) \(\text{rev-image-eqI}\))

**Lemma** \( \text{Tree-wf}_p\)-trivialI [simp]:

79
assumes \((a, b) \in \text{Tree-wf}\)
shows \((\text{abs-Tree}_p \ a, \text{abs-Tree}_p \ b) \in \text{Tree-wf}_p\)
using assms by (metis \text{Tree-wf}_p \ I \ \text{permute-zero})

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

**lemma** \text{wf-Tree-wf}_p; \text{wf Tree-wf}_p
apply (rule \text{wf-subset}[of \text{inv-image} \ (\text{hull-rel} \ O \ \text{Tree-wf}) \ \text{rep-Tree}_p])
apply (metis \text{Tree-wf-eqvt} \ \text{wf-Tree-wf} \ \text{wf-hull-rel-recomp} \ \text{wf-inv-image})
apply (auto elim: \text{Tree-wf}_p \ E)
apply (rename-tac t1 t2)
apply (rule-tac t=t1 in \text{permute-rep-abs-Tree}_p)
apply (rule-tac t=t2 in \text{permute-rep-abs-Tree}_p)
apply (rename-tac p1 p2)
apply (subgoal-tac (p2 \cdot t1, p2 \cdot t2) \in \text{Tree-wf})
apply (subgoal-tac (p1 \cdot t1, p2 \cdot t1) \in \text{hull-rel})
apply (metis \text{recomp.recompI})
apply (metis \text{hull-rel.simps} \ \text{permute-minus-cancel}(2) \ \text{permute-plus})
apply (metis \text{Tree-wf-eqvt-aux})
done

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

Here it comes . . .

**function** (sequential)
\text{alpha-Tree} :: (\text{idx},\text{pred}::pt,\text{act}::bn,\text{eff}::fs) \ \text{Tree} \Rightarrow (\text{idx},\text{pred},\text{act},\text{eff}) \ \text{Tree} \Rightarrow \text{bool} (\text{infix} =_a 50) \ where
− \(= (\_ =_a \_)
| \text{alpha-tConj}: \ tConj \ tset1 =_a tConj \ tset2 \leftrightarrow \text{rel-bset alpha-Tree} \ tset1 \ tset2
| \text{alpha-tNot}: \ tNot \ t1 =_a tNot \ t2 \leftrightarrow t1 =_a t2
| \text{alpha-tPred}: \ tPred \ f1 \ \varphi \ 1 =_a tPred \ f2 \ \varphi \ 2 \leftrightarrow f1 = f2 \ \land \ \varphi \ 1 = \varphi \ 2
− \text{the action may have binding names}
| \text{alpha-tAct}: \ tAct \ f1 \ \alpha \ 1 \ t1 =_a tAct \ f2 \ \alpha \ 2 \ t2 \leftrightarrow f1 = f2 \ \land \ (\exists \ p. \ (bn \ \alpha \ 1, \ t1) \ \approxset \ \text{alpha-Tree} \ \text{(supp-rel alpha-Tree)} \ p \ (bn \ \alpha \ 2, \ t2) \ \land \ (bn \ \alpha \ 1, \ \alpha \ 1) \ \approxset (\approx) \ \text{supp} \ p \ (bn \ \alpha \ 2, \ \alpha \ 2))
| \text{alpha-other}: - =_a \_ \leftrightarrow \text{False}
− \text{254 subgoals (!)}
by \text{pat-completeness auto}

termination

**proof**
let \(?R = \text{inv-image} \ (\text{max-ext Tree-wf}_p) \ \text{alpha-Tree-termination} 
show \text{wf} \ ?R
by (metis \text{max-ext-wf} \ \text{Tree-wf}_p \ \text{wf-inv-image})

80
We provide more descriptive case names for the automatically generated induction principle, and specialize it to an induction rule for $\alpha$-equivalence.

**Lemma** alpha-Tree-induct' = alpha-Tree.induct\[case-names\] alpha-tConj \alpha\text{-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 $t_1 =_\alpha t_2$
- and $\bigwedge tset_1 \ tset_2. (\bigwedge a \ b. a \in \text{set-bset} \ tset_1 \implies b \in \text{set-bset} \ tset_2 \implies a =_\alpha b \implies P \ a \ b) \implies trel-bset (=_\alpha) \ tset_1 \ tset_2 \implies P \ (tConj \ tset_1) \ (tConj \ tset_2)$
- and $\bigwedge t t_1 \ t_2. t_1 =_\alpha t_2 \implies P \ t_1 \ t_2 \implies P \ (tNot \ t_1) \ (tNot \ t_2)$
- and $\bigwedge \varphi. P \ (tPred f \varphi) \ (tPred f \varphi)$
- and $\bigwedge t_1 t_2 t_1 \ t_1 =_\alpha t_2 \implies P \ (p \cdot t_1) \ t_2 \implies f_1 = f_2 \implies (3 \ p. \ (\text{bn} \ t_1) \ t_1 \approx \text{set} (=_\alpha) \ (\text{supp-rel} (=_\alpha)) \ p \ (\text{bn} \ t_2) \ 
\land (\text{bn} \ t_1) \ t_2 = (\text{supp} \ (\text{bn} \ t_2) \ (\text{bn} \ t_1, \ t_1) \approx \text{set} (=_\alpha) \ (\text{supp-rel} (=_\alpha)) \ (p \cdot t)) \implies P \ (tAct f_1 \ t_1) \ (tAct f_2 \ t_2) \ 
\text{shows} P \ t_1 \ t_2 \ 
\text{using} \ \text{assms} \ by \ \text{(induction} \ t_1 \ t_2 \text{ rule: alpha-Tree.induct)} \ \text{simp-all} \ 
\alpha$-equivalence is equivariant.

**Lemma** alpha-Tree-eqvt-aux:

- assumes $\bigwedge a \ b. (a =_\alpha b) \cdot t =_\alpha t \longleftrightarrow p \cdot (a =_\alpha b) \cdot t =_\alpha p \cdot t$
- shows $p \cdot \text{supp-rel} (=_\alpha) \ t = \text{supp-rel} (=_\alpha) \ (p \cdot t)$
- proof –
  
  - fix $a$
  
  - let $?B = \{ b. \neg ((a =_\alpha b) \cdot t) =_\alpha t \}$
  
  - let $?pB = \{ b. \neg ((p \cdot a =_\alpha b) \cdot p \cdot t) =_\alpha (p \cdot t) \}$
  
  - assume finite $?B$
  
  - moreover have inj-on (unpermute $p$) $?pB$
    
  - by (simp add: inj-on-def unpermute-def)
  
  - moreover have unpermute $p \cdot ?pB \subseteq ?B$
    
  - 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 $p \cdot ?B \subseteq ?pB$

81
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)
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
\[
\begin{aligned}
\{ & \text{fix } x \\
& \text{assume } 1: x \in \text{set-bset } tset1 \\
& \text{then have } p \cdot x \in \text{set-bset (p } \cdot \text{tset1) } \\
& \quad \text{by (metis mem-permute-iff set-bset-eqvt) } \\
& \text{then obtain } y' \text{ where } 2: y' \in \text{set-bset (p } \cdot \text{tset2) and } 3: p \cdot x =_{\alpha} y' \\
& \text{using * by (metis FL-Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq rel-set-def) } \\
& \quad \text{from 2 obtain } y \text{ where } 4: y \in \text{set-bset tset2 and } 5: y' = p \cdot y \\
& \quad \text{by (metis imageE permute-bset.rep-eq permute-set-eq-image) } \\
& \text{from 1 and 3 and 4 and 5 have } x =_{\alpha} y \\
& \text{using alpha-tConj.IH by blast } \\
& \quad \text{with } 4 \text{ have } \exists y \in \text{set-bset tset2. } x =_{\alpha} y \ldots \\
\}
\text{moreover} \\
\{ & \text{fix } y \\
& \text{assume } 1: y \in \text{set-bset tset2} \\
& \text{then have } p \cdot y \in \text{set-bset (p } \cdot \text{tset2) } \\
& \quad \text{by (metis mem-permute-iff set-bset-eqvt) } \\
& \text{then obtain } x' \text{ where } 2: x' \in \text{set-bset (p } \cdot \text{tset1) and } 3: x' =_{\alpha} p \cdot y \\
& \text{using * by (metis FL-Formula.alpha-tConj permute-Tree-tConj rel-bset.rep-eq rel-set-def) } \\
& \quad \text{from 2 obtain } x \text{ where } 4: x \in \text{set-bset tset1 and } 5: p \cdot x = x' \\
& \quad \text{by (metis imageE permute-bset.rep-eq permute-set-eq-image) } \\
& \text{from 1 and 3 and 4 and 5 have } x =_{\alpha} y \\
& \text{using alpha-tConj.IH by blast } \\
& \quad \text{with } 4 \text{ have } \exists x \in \text{set-bset tset1. } x =_{\alpha} y \ldots \\
\}
\text{ultimately show } tConj \text{ tset1 } =_{\alpha} tConj \text{ tset2 } \\
\text{by (simp add: rel-bset-def rel-set-def) }
\text{qed}
\end{aligned}
\]
by (metis Diff-eqvt alpha-set bn-eqvt fresh-star-permute-iff permute-perm-def)

moreover from 1 and alpha-tAct.IH(1) have \( p \cdot q \cdot t1 =_a p \cdot t2 \)
by (simp add: alpha-set)

moreover from 2 have \( p \cdot q \cdot \neg p \cdot bn \ (p \cdot \alpha1) = bn \ (p \cdot \alpha2) \)
by (simp add: alpha-set bn-eqvt)

ultimately have \( (bn \ (p \cdot \alpha1), p \cdot t1) \approx set (=_a) \ (supp-rel (=_a)) \ (p + q - p) \ (bn \ (p \cdot \alpha2), p \cdot t2) \)
by (simp add: alpha-set)

moreover from 2 have \( (bn \ (p \cdot \alpha1), p \cdot \alpha1) \approx set (=_a) \ supp (p + q - p) \ (bn \ (p \cdot \alpha2), p \cdot \alpha2) \)
by (simp add: alpha-set) (metis (mono-tags, lifting) Diff-eqvt bn-eqvt fresh-star-permute-iff permute-minus-cancel(2) permute-perm-def supp-eqvt)

ultimately show \( p \cdot tAct f1 \alpha1 t1 =_a p \cdot tAct f2 \alpha2 t2 \) using 0
by auto

next

assume \( p \cdot tAct f1 \alpha1 t1 =_a p \cdot tAct f2 \alpha2 t2 \)

then obtain \( q \) where 0: \( f1 = f2 \) and 1: \( (bn \ (p \cdot \alpha1), p \cdot t1) \approx set (=_a) \)

(supp-rel (=_a)) \( q \ (bn \ (p \cdot \alpha2), p \cdot t2) \) and 2: \( (bn \ (p \cdot \alpha1), p \cdot \alpha1) \approx set (=_a) \)

supp \( q \ (bn \ (p \cdot \alpha2), p \cdot \alpha2) \)
by auto

\{
  from 1 and \( t1 \) and \( t2 \) have \( supp-rel (=_a) \) \( t1 - bn \alpha1 = supp-rel (=_a) \) \( t2 - bn \alpha2 \)

  by (metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff)

  moreover with \( t1 \) and \( t2 \) have \( supp-rel (=_a) \) \( t1 - bn \alpha1 \) \( \approx* (-p + q + p) \)

  by (auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting)

    Diff bn-eqvt mem-permute-iff permute-minus-cancel(2))

  moreover from 1 have \( -p \cdot q \cdot p \cdot t1 =_a t2 \)

  using alpha-tAct.IH(1) by (simp add: alpha-set) (metis (no-types, lifting)

    permute-eqvt permute-minus-cancel(2))

  moreover from 1 have \( -p \cdot q \cdot p \cdot bn \alpha1 = bn \alpha2 \)

  by (metis alpha-set bn-eqvt permute-minus-cancel(2))

  ultimately have \( (bn \alpha1, t1) \approx set (=_a) \ (supp-rel (=_a)) \ (-p + q + p) \) \( (bn \alpha2, t2) \)

  by (simp add: alpha-set)
\}

moreover

\{
  from 2 have \( supp \alpha1 - bn \alpha1 = supp \alpha2 - bn \alpha2 \)

  by (metis (no-types, lifting) Diff-eqvt alpha-set bn-eqvt permute-eq-iff

    supp-eqvt)

  moreover with \( 2 \) have \( supp \alpha1 - bn \alpha1 \) \( \approx* (-p + q + p) \)

  by (auto simp add: fresh-star-def fresh-perm alphas) (metis (no-types, lifting)

    Diff bn-eqvt mem-permute-iff permute-minus-cancel(1) supp-eqvt)

  moreover from \( 2 \) have \( -p \cdot q \cdot p \cdot \alpha1 = \alpha2 \)

  by (simp add: alpha-set)

  moreover have \( -p \cdot q \cdot p \cdot bn \alpha1 = bn \alpha2 \)

  by (simp add: bn-eqvt calculation(3))
\}
ultimately have \((bn \alpha_1, \alpha_1) \approx set (=) supp (-p + q + p) (bn \alpha_2, \alpha_2)\) by \((simp add: alpha-set)\)

ultimately show \(\text{tAct} f1 \alpha_1 t1 =_\alpha tAct f2 \alpha_2 t2\) using 0 by auto
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 \(t :: ('a,'b,'c,'d) Tree\)
show \(t =_\alpha t\)
proof \((induction t)\)
case \(\text{tConj}\) then show \(?case\) by \((metis alpha-tConj rel-bset.rep-eq rel-setI)\)
next
case \(\text{tNot}\) then show \(?case\) by \((metis alpha-tNot)\)
next
case \(\text{tPred}\) show \(?case\) by \((metis alpha-tPred)\)
next
case \(\text{tAct}\) then show \(?case\) by \((metis (mono-tags) alpha-tAct alpha-refl (1))\)
qed simp-all
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 \(\text{tConj}\) then show \(?case\)
  by \((simp add: rel-bset-def rel-set-def)\) metis
next
case \((\text{tAct} f1 \alpha_1 t1 f2 \alpha_2 t2)\)
  then obtain \(p\) where \(f1=f2 \land (bn \alpha_1, t1) \approx set (=) (supp-rel (=)) p (bn \alpha_2, t2) \land (bn \alpha_1, \alpha_1) \approx set (=) supp p (bn \alpha_2, \alpha_2)\)
  by auto
  then have \(f1=f2 \land (bn \alpha_2, t2) \approx set (=) (supp-rel (=)) (-p) (bn \alpha_1, t1) \land (bn \alpha_2, \alpha_2) \approx set (=) supp (-p) (bn \alpha_1, \alpha_1)\)
  using \(\text{tAct.IH}\) 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 =_a y and y =_a z
then show x =_a 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 (=_a) tset-x tset-z unfolding rel-bset.rep-eq rel-set-def Ball-def Bex-def
proof
show \( \forall x'. x' \in \text{set-bset} \text{tset-x} \rightarrow (\exists z'. z' \in \text{set-bset} \text{tset-z} \land x' =_a z') \)
proof (rule allI, rule impI)
fix x' assume 1: x' \in \text{set-bset} \text{tset-x}
then obtain y' where 2: y' \in \text{set-bset} \text{tset-y} and 3: x' =_a y'
by (metis rel-bset.rep-eq rel-set-def tConj.prems)
from 2 obtain z' where 4: z' \in \text{set-bset} \text{tset-z} and 5: y' =_a z'
by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prems)
from 1 2 3 5 have x' =_a z'
by (rule tConj.IH)
with 4 show \( \exists z'. z' \in \text{set-bset} \text{tset-z} \land x' =_a z' \)
by auto
qed
next
show \( \forall z'. z' \in \text{set-bset} \text{tset-z} \rightarrow (\exists x'. x' \in \text{set-bset} \text{tset-x} \land x' =_a z') \)
proof (rule allI, rule impI)
fix z' assume 1: z' \in \text{set-bset} \text{tset-z}
then obtain y' where 2: y' \in \text{set-bset} \text{tset-y} and 3: y' =_a z'
by (metis alpha-tConj rel-bset.rep-eq rel-set-def tConj.prems)
from 2 obtain x' where 4: x' \in \text{set-bset} \text{tset-x} and 5: x' =_a y'
by (metis rel-bset.rep-eq rel-set-def tConj.prems)
from 4 2 5 3 have x' =_a z'
by (rule tConj.IH)
with 4 show \( \exists x'. x' \in \text{set-bset} \text{tset-x} \land x' =_a z' \)
by auto
qed
qed
with z show tConj tset-x =_a 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 x f2 y) show ?case
proof (cases z)
  fix \( f \alpha t \)
  assume \( z \colon z = t \text{Act} f \alpha t \)
  obtain \( p \) where \( 1 \colon f1 = f2 \land (bn \alpha1, t1) \approxset (\equivp \alpha) \) \( (supp \equivp \alpha) p \) \( (bn \alpha2, t2) \land (bn \alpha1, \alpha1) \approxset (\equiv) supp p \) \( (bn \alpha2, \alpha2) \)
  using \( t \text{Act}.\text{hyps} \) by auto
  obtain \( q \) where \( 2 \colon f2 = f \land (bn \alpha2, t2) \approxset (\equivp \alpha) \) \( (supp \equivp \alpha) q \) \( (bn \alpha, t) \land (bn \alpha2, \alpha2) \approxset (\equiv) supp q \) \( (bn \alpha, \alpha) \)
  using \( t \text{Act}.\text{prems} z \) by auto
  have \( f1 = f \land (bn \alpha1, t1) \approxset (\equivp \alpha) \) \( (supp \equivp \alpha) (q + p) \) \( (bn \alpha, t) \)
  proof
    have \( supp \equivp \alpha t1 - bn \alpha1 = supp \equivp \alpha t - bn \alpha \)
    using \( 1 \) and \( 2 \) by (metis \( \alpha \)-set)
    moreover have \( (supp \equivp \alpha t1 - bn \alpha1) \sharp (q + p) \)
    using \( 1 \) and \( 2 \) by (metis \( \alpha \)-set \( \# \)-star-plus)
    moreover have \( (q + p) \cdot t1 = \alpha t \)
    using \( 1 \) and \( 2 \) and \( t \text{Act}.IH \) by (metis \( no \)-types, lifting) \( \alpha \)-Tree-eqvt \( \alpha \)-set \( \equivp \) permute-minus-cancel(\( 1 \)) \( \equivp \) permute-plus)
    moreover have \( (q + p) \cdot bn \alpha1 = bn \alpha \)
    using \( 1 \) and \( 2 \) by (metis \( \alpha \)-set \( \equivp \) permute-plus)
    moreover have \( f1 = f \)
    using \( 1 \) and \( 2 \) by simp
    ultimately show \( ?\text{thesis} \)
    by (metis \( \alpha \)-set)
  qed

  moreover have \( (bn \alpha1, \alpha1) \approxset (\equiv) supp (q + p) \) \( (bn \alpha, \alpha) \)
  using \( 1 \) and \( 2 \) by (metis \( mono \)-tags \( \equivp \)) \( \alpha \)-trans \( \equivp \) permute-plus)
  ultimately show \( t \text{Act} f1 \alpha1 t1 = \alpha z \)
  using \( z \) by auto
  qed (insert \( t \text{Act}.\text{prems}, auto \)
  qed

lemma \( \alpha \)-Tree-equivp: equivp \( \alpha \)-Tree
by (auto intro: equivpI \( \alpha \)-Tree-reflp \( \equivp \) \( \alpha \)-Tree-symp \( \alpha \)-Tree-transp)
\( \alpha \)-equivp trees have the same support modulo \( \alpha \)-equivalence.

lemma \( \alpha \)-Tree-supp-rel:
  assumes \( t1 = \alpha t2 \)
  shows \( supp \equivp \alpha t1 = supp \equivp \alpha t2 \)
  using \( \text{assms} \) proof (induction rule: \( \alpha \)-Tree-induct)
  case \( t \text{Conj} tset1 tset2 \)
  have \( \wedge x. y. \text{rel-bset} (\equivp \alpha) x y \iff \text{rel-bset} (\equivp \alpha) y x \)
  by (meson \( \alpha \)-Tree-symp \( \text{bset}\).rel-symp \( \text{symp} \) \( \text{E} \))
  \{
    fix \( a b \)
    from \( t \text{Conj}.\text{hyps} \) have \( \ast \colon \text{rel-bset} (\equivp \alpha) ((a = b) \cdot tset1) ((a = b) \cdot tset2) \)
    by (metis \( \alpha \)-trans \( \equivp \) \( \text{tConj} \) \( \alpha \)-Tree-eqvt permute-Tree-tConj)
    have \( \text{rel-bset} (\equivp \alpha) ((a = b) \cdot tset1) tset1 \iff \text{rel-bset} (\equivp \alpha) ((a = b) \cdot tset2) \)
  \}

87
tset2

\begin{proof}
\begin{enumerate}
\item Then show \texttt{?case}
  \begin{enumerate}
  \item by (simp add: supp-rel-def)
  \end{enumerate}
\item case \texttt{tNot} then show \texttt{?case}
  \begin{enumerate}
  \item by (simp add: supp-rel-def)
  \end{enumerate}
\item case \texttt{(tAct f1 α1 t1 f2 α2 t2)}
  \begin{enumerate}
  \item fix \(a\) \(b\)
  \item have \(tAct f1 α1 t1 =_α tAct f2 α2 t2\)
  \begin{enumerate}
  \item using \(tAct\).hyps by simp
  \end{enumerate}
  \item then have \((a = b) \cdot tAct f1 α1 t1 =_α tAct f1 α1 t1 \leftrightarrow (a = b) \cdot tAct f2 α2 t2 =_α tAct f2 α2 t2\)
  \begin{enumerate}
  \item by (metis (no-types, lifting) alpha-Tree-eqvt alpha-Tree-symp alpha-Tree-transp sympE transpE)
  \end{enumerate}
  \end{enumerate}
\item then show \texttt{?case}
  \begin{enumerate}
  \item by (simp add: supp-rel-def)
  \end{enumerate}
\end{enumerate}
\end{proof}

\texttt{tAct} preserves \(α\)-equivalence.

\textbf{lemma alpha-Tree-tAct:}
\begin{enumerate}
\item assumes \(t1 =_α t2\)
\item shows \(tAct f α t1 =_α tAct f α t2\)
\end{enumerate}
\begin{proof}
\begin{enumerate}
\item have \((bn α, t1) \approx \set{=} (supp-rel \(=\alpha\)) 0 (bn α, t2)\)
  \begin{enumerate}
  \item using \(assms\) by (simp add: alpha-Tree-supp-rel alpha-set fresh-star-zero)
  \end{enumerate}
\item moreover have \((bn α, α) \approx \set{=} \set{0} (bn α, α)\)
  \begin{enumerate}
  \item by (metis (full-types) alpha-refl(1))
  \end{enumerate}
\item ultimately show \texttt{?thesis}
  \begin{enumerate}
  \item by auto
  \end{enumerate}
\end{enumerate}
\end{proof}

The following lemmas describe the support modulo \(α\)-equivalence.

\textbf{lemma supp-rel-tNot [simp]: supp-rel \(=\alpha\) \(tNot \ t\) = supp-rel \(=\alpha\) \(t\)}
\begin{proof}
\textbf{unfolding} supp-rel-def \textbf{by} simp
\end{proof}

\textbf{lemma supp-rel-tPred [simp]: supp-rel \(=\alpha\) \(tPred f \ ϕ\) = supp f \(∪\) supp \(ϕ\)}
\begin{proof}
\textbf{unfolding} supp-rel-def \textbf{supp-def} \textbf{by} (simp add: Collect-imp-eq Collect-neg-eq)
\end{proof}

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

\textbf{lemma infinite-mono: infinite \(S\) \(⇒\) \(∀x. x \in S \Rightarrow x \in T\) \(⇒\) infinite \(T\)}
by (metis infinite-super subsetI)

lemma supp-rel-tAct [simp]:
  assumes finite (supp-rel (=α) t)
  shows supp-rel (=α) (tAct f α t) = supp f ∪ (supp α ∪ supp-rel (=α) t − bn α)
proof
  show supp f ∪ (supp α ∪ supp-rel (=α) t − bn α) ⊆ supp-rel (=α) (tAct f α t)
  proof
    fix x
    assume x ∈ supp f ∪ (supp α ∪ supp-rel (=α) t − bn α)
    moreover
      { assume x1: x ∈ supp f
        from x1 have infinite \{ b. (x ⇔ b) ⋅ f ≠ f \}
        unfolding supp-def ..
        then have infinite \{ b. (x ⇔ b) ⋅ f ≠ f \} − supp f
        by (simp add: finite-supp)
        moreover
          { fix b
            assume b ∈ \{ b. (x ⇔ b) ⋅ f ≠ f \} − supp f
            then have b1: (x ⇔ b) ⋅ f ≠ f and b2: b /∈ supp f
            by simp+
            then have sort-of x = sort-of b
              using swap-different-sorts by fastforce
            then have (x ⇔ b) ⋅ supp f ≠ supp f
              using b2 x1 using swap-set-in by blast
            then have b ∈ \{ b. ¬ (x ⇔ b) ⋅ tAct f α t =α tAct f α t \}
              by auto
            }}
      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 α and x2: x /∈ bn α
        from x1 have infinite \{ b. (x ⇔ b) ⋅ α ≠ α \}
        unfolding supp-def ..
        then have infinite \{ b. (x ⇔ b) ⋅ α ≠ α \} − supp α
          by (simp add: finite-supp)
        moreover
          { fix b
            assume b ∈ \{ b. (x ⇔ b) ⋅ α ≠ α \} − supp α
            then have b1: (x ⇔ b) ⋅ α ≠ α and b2: b /∈ supp α − bn α
              by simp+
          }
      }

89
from b1 have sort-of x = sort-of b
  using swap-differentSorts by fastforce
then have \((x \equiv b) \cdot (\Supp \alpha - bn \alpha) \neq \Supp \alpha - bn \alpha\)
  using b2 x1 x2 by (simp add: swap-set-in)
then have \(b \in \{ b, \neg (x \equiv b) \cdot t \Act f \alpha t =_\alpha t \Act f \alpha t \}\)
  by (auto simp add: alpha-set Diff-eqvt bn-eqvt)
} ultimately have infinite \(\{ b. \neg (x \equiv b) \cdot t \Act f \alpha t =_\alpha t \Act f \alpha t \}\)
  by (rule infinite mono)
then have \(x \in \Supp -rel \(=\alpha \) \(t \Act f \alpha t \)\)
unfolding \(\Supp -rel -def \) ..
} moreover
{ assume x1: \(x \in \Supp -rel \(=\alpha \) \(t\)\) and x2: \(x \notin bn \alpha\)
from x1 have infinite \(\{ b. \neg (x \equiv b) \cdot t =_\alpha t\}\)
  unfolding \(\Supp -rel -def \) ..
then have infinite \(\{ b. \neg (x \equiv b) \cdot t =_\alpha t\} - \Supp -rel \(=\alpha \) \(t\)\)
  using assms by simp
moreover
{ fix b
  assume b1: \(b. \neg (x \equiv b) \cdot t =_\alpha t\) - \Supp -rel \(=\alpha \) \(t\)
  then have \(b1: \neg (x \equiv b) \cdot t =_\alpha t\) and b2: \(b \notin \Supp -rel \(=\alpha \) \(t\) - bn \(=\alpha \) \(t\)\)
  by simp+
  from b1 have \((x \equiv b) \cdot t \neq t\)
    by (metis alpha -Tree-reflp reflpE)
  then have sort-of x = sort-of b
    using swap-differentSorts by fastforce
  then have \((x \equiv b) \cdot (\Supp -rel \(=\alpha \) \(t\) - bn \(=\alpha \) \(t\) \neq \Supp -rel \(=\alpha \) \(t\) - bn \(=\alpha \) \(t\)\)
    using b2 x1 x2 by (simp add: swap-set-in)
  then have \(\Supp -rel \(=\alpha \) \((x \equiv b) \cdot t\) - bn \((x \equiv b) \cdot \alpha\) \neq \Supp -rel \(=\alpha \) \(t\)\)
    t - bn \(=\alpha \) \(t\)
    by (simp add: Diff-eqvt bn-eqvt)
  then have \(b \in \{ b. \neg (x \equiv b) \cdot t \Act f \alpha t =_\alpha t \Act f \alpha t \}\)
    by (simp add: alpha-set)
} ultimately have infinite \(\{ b. \neg (x \equiv b) \cdot t \Act f \alpha t =_\alpha t \Act f \alpha t \}\)
  by (rule infinite mono)
then have \(x \in \Supp -rel \(=\alpha \) \(t \Act f \alpha t\)\)
  unfolding \(\Supp -rel -def \) ..
} ultimately show \(x \in \Supp -rel \(=\alpha \) \(t \Act f \alpha t\)\)
  by auto
qed
next
show \(\Supp -rel \(=\alpha \) \(t \Act f \alpha t\) \subseteq \Supp \cup (\Supp \alpha \cup \Supp -rel \(=\alpha \) \(t - bn \alpha\)\))
proof
  fix x
assume \( x \in \text{supp-rel} (=_a) (\text{tAct} f \alpha t) \)
then have \( \star : \text{infinite} \{ b. \neg (x \equiv b) \cdot \text{tAct} f \alpha t =_a \text{tAct} f \alpha t \} \)
  unfolding supp-rel-def ..
moreover
\{
  \text{fix} b
  assume \( \neg (x \equiv b) \cdot \text{tAct} f \alpha t =_a \text{tAct} f \alpha t \)
  then have \( (x \equiv b) \cdot f \neq f \lor (x \equiv b) \cdot \alpha \neq \alpha \lor \neg (x \equiv b) \cdot t =_a t \)
  using alpha-Tree-tAct by force
\}
ultimately have \( \text{infinite} \{ b. (x \equiv b) \cdot f \neq f \lor (x \equiv b) \cdot \alpha \neq \alpha \lor \neg (x \equiv b) \cdot t =_a t \} \)
  using infinite-mono mem-Collect-eq by force
then have \( \text{infinite} \{ b. (x \equiv b) \cdot f \neq f \} \lor \text{infinite} \{ b. (x \equiv b) \cdot \alpha \neq \alpha \} \lor \text{infinite} \{ b. \neg (x \equiv b) \cdot t =_a t \} \)
  by (metis (mono-tags) finite-Collect-disjI)
then have \( x \in \text{supp} f \lor \text{supp} \alpha \cup \text{supp-rel} (=_a) t \)
  by (simp add: supp-def supp-rel-def)
moreover
\{
  \text{assume} \( \star\star : x \in \text{bn} \alpha \land x \notin \text{supp} f \)
  from \( \star\) obtain \( b \) where \( b\theta : \neg (x \equiv b) \cdot \text{tAct} f \alpha t =_a \text{tAct} f \alpha t \) \text{ and } \( b1: b \notin \text{supp} f \) \text{ and } \( b2: b \notin \text{supp} \alpha \) \text{ and } \( b3: b \notin \text{supp-rel} (=_a) t \)
  using assms by (metis (no-types, lifting) UnCI finite-UnI finite-supp infinite-mono mem-Collect-eq)
  let \( ?p = (x \equiv b) \)
  have \( \text{supp-rel} (=_a) ((x \equiv b) \cdot t) = \text{bn} ((x \equiv b) \cdot \alpha) = \text{supp-rel} (=_a) t = \text{bn} \alpha \)
  using \( \star\star\) and \( b3 \) by (metis (no-types, lifting) Diff-eqvt Diff-iff alpha-Tree-eqvt' alpha-Tree-eqvt-aux bn-eqvt swap-set-not-in)
  moreover then have \( (\text{supp-rel} (=_a) ((x \equiv b) \cdot t) = \text{bn} ((x \equiv b) \cdot \alpha)) \star\star \ ?p \)
  using \( \star\star\) and \( b3 \) by (metis Diff-iff fresh-perm fresh-star-def swap-atom-simps(3))
moreover have \( ?p \cdot (x \equiv b) \cdot t =_a t \)
  using alpha-Tree-reflp reflpE by force
moreover have \( ?p \cdot \text{bn} ((x \equiv b) \cdot \alpha) = \text{bn} \alpha \)
  by (simp add: bn-eqvt)
moreover have \( \text{supp} ((x \equiv b) \cdot \alpha) = \text{bn} ((x \equiv b) \cdot \alpha) = \text{supp} \alpha = \text{bn} \alpha \)
  using \( \star\star\) and \( b2 \) by (metis (mono-tags, hide-lams) Diff-eqvt Diff-iff bn-eqvt supp-eqvt swap-set-not-in)
moreover then have \( (\text{supp} ((x \equiv b) \cdot \alpha) = \text{bn} ((x \equiv b) \cdot \alpha)) \star\star ?p \)
  using \( \star\star\) and \( b2 \) by (simp add: fresh-star-def fresh-def supp-perm) (metis Diff-iff swap-atom-simps(3))
moreover have \( ?p \cdot (x \equiv b) = \alpha = \alpha \)
  by simp
ultimately have \( \exists p. (\text{bn} ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot \alpha) \approx \text{set} (=_a) \text{supp-rel} (=_a) p (\text{bn} \alpha, t) \land (\text{bn} ((x \equiv b) \cdot \alpha), (x \equiv b) \cdot \alpha) \approx \text{set} (=_a) \text{supp} p (\text{bn} \alpha, \alpha) \)
  by (auto simp add: alpha-set.simps)
moreover have \((x \equiv b) \cdot f = f\) using \(*\) and \(b1\)
by \(\text{simp add: fresh-def swap-fresh-fresh}\)
ultimately have \((x \equiv b) \cdot t \alpha t = _\alpha t \alpha t\)
by \text{simp}
with \(bu\) have \(\mathit{False}..\)
}
ultimately show \(x \in \text{supp } f \cup (\text{supp } \alpha \cup \text{supp-rel } (=_\alpha) \ t - \text{bn } \alpha)\)
by \text{blast}
qed

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

\begin{align*}
\text{quotient-type} & \quad \text{('idx,'pred,'act,'eff) } \text{Tree}_\alpha = \text{('idx,'pred::pt,'act::bn,'eff::fs) Tree} / \alpha\text{-Tree} \\
\text{by} & \quad \text{fact alpha-Tree-equivp}
\end{align*}

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

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

The permutation operation is lifted from trees.

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

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

\text{instance}
\text{proof}
\fix \quad t_\alpha :: (_\cdot\cdot\cdot) \text{ Tree}_\alpha
\show \quad \theta \cdot t_\alpha = _\alpha t_\alpha
\by \quad \text{transfer (metis alpha-Tree-equivp equivp-reflp permute-zero)}
\next
\fix \quad p \ q :: \text{perm and } t_\alpha :: (_\cdot\cdot\cdot) \text{ Tree}_\alpha
\show \quad (p + q) \cdot t_\alpha = p \cdot q \cdot t_\alpha
\by \quad \text{transfer (metis alpha-Tree-equivp equivp-reflp permute-plus)}
\qed

\text{end}

The abstraction function from trees to trees modulo \(\alpha\)-equivalence is equivariant. The representation function is equivariant modulo \(\alpha\)-equivalence.

\text{lemmas} \quad \text{permute-Tree}_\alpha\text{-abs-eq [eqvt, simp]}

\text{lemma} \quad \text{alpha-Tree-permute-rep-commute [simp]}: p \cdot \text{rep-Tree}_\alpha t_\alpha = _\alpha \text{rep-Tree}_\alpha (p \cdot t_\alpha)
14.3 Constructors for trees modulo $\alpha$-equivalence

The constructors are lifted from trees.

lift-definition $\text{Conj}_\alpha :: (\text{'idx}, \text{'pred}, \text{'act}, \text{'eff}) \rightarrow \text{Tree}_\alpha$ is
$\text{tConj}$

by simp

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

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

lift-definition $\text{Not}_\alpha :: (\text{'idx}, \text{'pred}, \text{'act}, \text{'eff}) \rightarrow \text{Tree}_\alpha$ is
$\text{tNot}$

by simp

lift-definition $\text{Pred}_\alpha :: \text{'eff} \Rightarrow \text{'pred} \Rightarrow (\text{'idx}, \text{'act}, \text{'eff}) \rightarrow \text{Tree}_\alpha$ is
$\text{tPred}$

lift-definition $\text{Act}_\alpha :: \text{'eff} \Rightarrow \text{'act} \Rightarrow (\text{'idx}, \text{'pred}, \text{'act}, \text{'eff}) \rightarrow \text{Tree}_\alpha$ is
$\text{tAct}$

by (fact alpha-Tree-tAct)

The lifted constructors are equivariant.

lemma $\text{Conj}_\alpha$-eqvt [eqvt, simp]: $p \cdot \text{Conj}_\alpha \text{tset}_\alpha = \text{Conj}_\alpha (p \cdot \text{tset}_\alpha)$
proof -
{
  fix $x$
  assume $x \in \text{set-bset} (p \cdot \text{map-bset rep-Tree}_\alpha \text{tset}_\alpha)$
  then obtain $y$ where $y \in \text{set-bset} (\text{map-bset rep-Tree}_\alpha \text{tset}_\alpha)$ and $x = p \cdot y$
    by (metis imageE permute-bset-rep-eq permute-set-eq-image)
  then obtain $t_o$ where 1: $t_o \in \text{set-bset tset}_\alpha$ and 2: $x = p \cdot \text{rep-Tree}_\alpha t_o$
    by (metis imageE map-bset-rep-eq)
  let $?x' = \text{rep-Tree}_\alpha (p \cdot t_o)$
  from 1 have $p \cdot t_o \in \text{set-bset} (p \cdot \text{tset}_\alpha)$
    by (metis mem-permute-iff permute-bset-rep-eq)
  then have $?x' \in \text{set-bset} (\text{map-bset rep-Tree}_\alpha (p \cdot \text{tset}_\alpha))$
    by (simp add bset.set-map)
  moreover from 2 have $x =_\alpha ?x'$
    by (metis alpha-Tree-permute-rep-commute)
  ultimately have $\exists x' \in \text{set-bset} (\text{map-bset rep-Tree}_\alpha (p \cdot \text{tset}_\alpha)). x =_\alpha x'$
moreover

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

ultimately show \ ?thesis \\
\text{by } (\text{simp add: Conj}_\alpha\text{-def'} \text{map-bset-eqvt rel-bset-def rel-set-def Tree}_\alpha \text{abs-eq-iff})

qed

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

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

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

The lifted constructors are injective (except for Act\(_\alpha\)).

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

lemma Not\(_\alpha\)-eq-iff [simp]: Not\(_\alpha\) t1\_\alpha = Not\(_\alpha\) t2\_\alpha \iff t1\_\alpha = t2\_\alpha \\
\text{proof}

94
\begin{itemize}
\item \textbf{lemma \texttt{Pred\textalpha\ -eq-iff} [simp]:} \texttt{Pred\textalpha\ f1 \varphi1 = Pred\textalpha\ f2 \varphi2 \iff f1 = f2 \land \varphi1 = \varphi2}
\begin{proof}
\item \textbf{assume} \texttt{Pred\textalpha\ f1 \varphi1 = Pred\textalpha\ f2 \varphi2}
\item \textbf{then have} \texttt{(tPred f1 \varphi1 :: ('e, 'b, 'f, 'd) Tree) =_\textalpha\ tPred f2 \varphi2} — note the unrelated type
\item \textbf{by} \texttt{(metis Pred\textalpha\ .abs-eq Tree\textalpha\ .abs-eq-iff)}
\item \textbf{then show} \texttt{f1 = f2 \land \varphi1 = \varphi2}
\item \textbf{using} \texttt{alpha-Tree.cases by auto}
\end{proof}
\end{itemize}

\begin{itemize}
\item \textbf{next}
\item \textbf{assume} \texttt{f1 = f2 \land \varphi1 = \varphi2}
\item \textbf{then show} \texttt{Pred\textalpha\ f1 \varphi1 = Pred\textalpha\ f2 \varphi2}
\item \textbf{by} \texttt{simp}
\end{itemize}
\begin{proof}
\item \textbf{lemma \texttt{Act\textalpha\ -eq-iff} [simp]:} \texttt{Act\textalpha\ f1 \textalpha\ tset\textalpha\ =} \texttt{Act\textalpha\ f2 \textalpha\ tset\textalpha\ \iff tAct f1 \textalpha\ tset\textalpha\ \leftrightarrow tAct f2 \textalpha\ tset\textalpha\}
\item \textbf{by} \texttt{(metis Act\textalpha\ .abs-eq Tree\textalpha\ .abs-eq-iff Tree\textalpha\ .abs-rep)}
\end{proof}

The following lemmas describe the support of constructed trees modulo $\textalpha$-equivalence.
\begin{itemize}
\item \textbf{lemma \texttt{supp-alpha-supp-rel} [simp]:} \texttt{supp t\textalpha\ = supp-rel (=\textalpha\) (rep-Tree\textalpha\ t\textalpha\)}
\item \textbf{unfolding supp-def supp-rel-def by} \texttt{(metis (mono-tags, lifting) Collect-cong Tree\textalpha\ .abs-eq-iff Tree\textalpha\ .abs-rep alpha-Tree-permute-rep-commute)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma \texttt{supp-Conj\textalpha\ [simp]:} \texttt{supp (Conj\textalpha\ tset\textalpha\) = supp tset\textalpha\}}
\item \textbf{unfolding supp-def by} \texttt{simp}
\end{itemize}

\begin{itemize}
\item \textbf{lemma \texttt{supp-Not\textalpha\ [simp]:} \texttt{supp (Not\textalpha\ t\textalpha\) = supp t\textalpha\}}
\end{itemize}

95
unfolding supp-def by simp

lemma supp-Pred_α [simp]: supp (Pred_α f ϕ) = supp f ∪ supp ϕ
unfolding supp-def by (simp add: Collect-imp-eq Collect-neg-eq)

lemma supp-Act_α [simp]:
  assumes finite (supp t_α)
  shows supp (Act_α f_α t_α) = supp f_α ∪ (supp α ∪ supp t_α − ba α)
  using assms by (metis Act_α.abs-eq Tree_α-abs-rep Tree_α-rep-abs alpha-Tree-supp-rel supp-alpha-supp-rel supp-rel-tAct)

14.4 Induction over trees modulo α-equivalence

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

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

We cannot obtain the type of infinitary formulas simply as the sub-type of all trees (modulo $\alpha$-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.

\[
\text{inductive hereditarily-fs} :: (\text{'idx,'pred::fs,'act::bn,'eff::fs}) \to \text{Tree}_\alpha \Rightarrow \text{bool}
\]
where
\[
\begin{align*}
\text{Conj}_\alpha &:: \text{finite} (\text{supp tset}_\alpha) \Rightarrow (\forall t_\alpha. t_\alpha \in \text{set-bset tset}_\alpha \Rightarrow \text{hereditarily-fs} t_\alpha) \\
\text{Not}_\alpha &:: \text{hereditarily-fs} t_\alpha \Rightarrow \text{hereditarily-fs} (\text{Not}_\alpha t_\alpha) \\
\text{Pred}_\alpha &:: \text{hereditarily-fs} (\text{Pred}_\alpha f \varphi) \\
\text{Act}_\alpha &:: \text{hereditarily-fs} t_\alpha \Rightarrow \text{hereditarily-fs} (\text{Act}_\alpha f \alpha t_\alpha)
\end{align*}
\]

hereditarily-fs is equivariant.

\[
\text{lemma hereditarily-fs-eqvt [eqvt]}:
\]
assumes hereditarily-fs $t_\alpha$
shows hereditarily-fs ($p \cdot t_\alpha$)
using assms proof (induction rule: hereditarily-fs.induct)
case Conj $\alpha$ then show ?case
by (metis (erased, hide-lams) Conj $\alpha$-eqvt hereditarily-fs Conj $\alpha$ mem-permute-iff permute-finite permute-minus-cancel (1) set-bset-eqvt supp-eqvt)
next
case Not $\alpha$ then show ?case
by (metis Not $\alpha$-eqvt hereditarily-fs Not $\alpha$)
next
case Pred $\alpha$ then show ?case
by (metis Pred $\alpha$-eqvt hereditarily-fs Pred $\alpha$)
next
case Act $\alpha$ then show ?case
by (metis Act $\alpha$-eqvt hereditarily-fs Act $\alpha$)
qed

hereditarily-fs is preserved under $\alpha$-renaming.

\[
\text{lemma hereditarily-fs-alpha-renaming}:
\]
assumes Act $\alpha$ $f \alpha t_\alpha = Act $\alpha$ $f' \alpha' t_\alpha'$
shows hereditarily-fs $t_\alpha \iff$ hereditarily-fs $t_\alpha'$
proof
assume hereditarily-fs $t_\alpha$
then show hereditarily-fs $t_\alpha'$
using assms by (auto simp add: Act $\alpha$-def Tree $\alpha$ abs-eq-iff alphas) (metis Tree $\alpha$ abs-eq-iff Tree $\alpha$ abs-rep hereditarily-fs-eqvt permute-Tree $\alpha$ abs-eq)
next
assume hereditarily-fs $t_\alpha'$
then show hereditarily-fs $t_\alpha$
using assms by (auto simp add: Act α-def Tree α.abs-eq-iff alphas) (metis Tree α.abs-eq-iff Tree α.abs-rep hereditarily-fs-eqvt permute-Tree α.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)

14.6 Infinitary formulas

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

typedef ('idx,'pred::fs,'act::bn,'eff::fs) formula = {t α::('idx,'pred,'act,'eff) Tree α. hereditarily-fs t α}
by (metis hereditarily-fs.Pred α 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 $t_\alpha$
shows $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha)$
by (metis assms eq-onp-same-args permute-formula.abs-eq)

lemma supp-Abs-formula [simp]:
assumes hereditarily-fs $t_\alpha$
shows $\text{supp } (\text{Abs-formula } t_\alpha) = \text{supp } t_\alpha$
proof
  \[
  \begin{align*}
  &\text{fix } p :: \text{perm} \\
  &\text{have } p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha) \\
  &\quad \text{using assms by (metis Abs-formula-eqvt)} \\
  &\quad \text{moreover have hereditarily-fs } (p \cdot t_\alpha) \\
  &\quad \text{using assms by (metis hereditarily-fs-eqvt)} \\
  &\quad \text{ultimately have } p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } t_\alpha \iff p \cdot t_\alpha = t_\alpha \\
  &\quad \text{using assms by (metis Abs-formula-inverse)}
  \end{align*}
  \]
then show \(?thesis\) unfolding supp-def by simp
qed

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

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

lemma supp-map-bset-Rep-formula [simp]: supp (map-bset Rep-formula $xset$) = supp $xset$
proof
  have \(?eqvt\) (map-bset Rep-formula)
  unfolding \(?eqvt-def\) by (simp add: ext)
then show supp (map-bset Rep-formula $xset$) \(\subseteq\) supp $xset$
  by (fact supp-fun-app-eqvt)
next
\[
\begin{align*}
  &\text{fix } a :: \text{atom} \\
  &\text{have inj } (\text{map-bset Rep-formula}) \\
  &\quad \text{by (metis bset.inj-map Rep-formula-inject injI)} \\
  &\quad \text{then have } \forall x, y. x \neq y \implies \text{map-bset Rep-formula } x \neq \text{map-bset Rep-formula } y \\
  &\quad \text{by (metis inj-eq)} \\
  &\quad \text{then have } \{ b. (a \equiv b) \cdot xset \neq xset \} \subseteq \{ b. (a \equiv b) \cdot \text{map-bset Rep-formula } xset \neq \text{map-bset Rep-formula } xset \} \text{ (is } ?S \subseteq ?T) \\
  &\quad \text{by auto} \\
  &\quad \text{then have } \text{infinite } ?S \implies \text{infinite } ?T \\
  &\quad \text{by (metis infinite-super)}
\end{align*}
\]
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) fs
by standard (metis Rep-formula" hereditarily-fs-implies-finite-supp supp-Rep-formula)
```

### 14.7 Constructors for infinitary formulas

We lift the constructors for trees (modulo $\alpha$-equivalence) to infinitary formulas. Since $\text{Conj}_\alpha$ 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 $\text{Conj}$. Instead, theorems about terms of the form $\text{Conj } \text{set}$ will usually carry an assumption that $\text{set}$ is finitely supported.

```
definition Conj :: ('idx,'pred,'act,'eff) formula set['idx] ⇒ ('idx,'pred::fs,'act::bn,'eff::fs)
formula where
Conj \text{set} = \text{Abs-formula} (\text{Conj}_\alpha (\text{map-bset Rep-formula} \text{set}))
```

```
lemma finite-supp-implies-hereditarily-fs-Conj
\text{α} [simp]:
assumes finite (supp \text{set})
shows hereditarily-fs (Conj\text{α} (map-bset Rep-formula \text{set}))
proof (rule hereditarily-fs.Conj\text{α})
  show finite (supp (map-bset Rep-formula \text{set}))
  using assms by (metis supp-map-bset-Rep-formula)
next
  fix t\text{α} assume t\text{α} ∈ set-bset (map-bset Rep-formula \text{set})
  then show hereditarily-fs t\text{α}
  by (auto simp add: bset.set-map)
qed

lemma Conj-rep-eq:
assumes finite (supp \text{set})
shows Rep-formula (Conj \text{set}) = Conj\text{α} (map-bset Rep-formula \text{set})
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\text{α}
by (fact hereditarily-fs.Not\text{α})

lift-definition Pred :: 'eff ⇒ 'pred ⇒ ('idx,'pred::fs,'act::bn,'eff::fs) formula is
  Pred\text{α}
by (fact hereditarily-fs.Pred\text{α})

lift-definition Act :: 'eff ⇒ 'act ⇒ ('idx,'pred,'act,'eff') formula ⇒ ('idx,'pred::fs,'act::bn,'eff::fs)
formula is
  Act\text{α}
by (fact hereditarily-fs.Act\text{α})
```

The lifted constructors are equivariant (in the case of $\text{Conj}$, on finitely sup-
ported arguments).

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

**lemma** Not-evq [eqvt, simp]: \( p \cdot \text{Not} \ x = \text{Not} \ (p \cdot x) \)
by transfer simp

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

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

The following lemmas describe the support of constructed formulas.

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

**lemma** supp-Not [simp]: supp (\( \text{Not} \ x \)) = supp x
by (metis Not.rep-eq supp-Not α supp-Rep-formula)

**lemma** supp-Pred [simp]: supp (\( \text{Pred} \ f \ \varphi \)) = supp f \( \cup \) supp ϕ
by (metis Pred.rep-eq supp-Pred α supp-Rep-formula)

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

The lifted constructors are injective (partially for Act).

**lemma** Conj-eq-iff [simp]:
assumes finite (supp xset1) and finite (supp xset2)
shows Conj xset1 = Conj xset2 ←→ xset1 = xset2
using assms

**lemma** Not-eq-iff [simp]: Not x1 = Not x2 ←→ x1 = x2
by (metis Not.rep-eq Not-α-eq-iff Rep-formula-inverse)

**lemma** Pred-eq-iff [simp]: Pred f1 \( \varphi \)1 = Pred f2 \( \varphi \)2 ←→ f1 = f2 \( \land \) \( \varphi \)1 = \( \varphi \)2
by (metis Pred.rep-eq Pred-α-eq-iff)

**lemma** Act-eq-iff: Act f1 \( \alpha \)1 x1 = Act f2 \( \alpha \)2 x2 ←→ Act_α f1 \( \alpha \)1 (Rep-formula x1) = Act_α f2 \( \alpha \)2 (Rep-formula x2)
by (metis Act.rep-evq Rep-formula-inverse)

Helpful lemmas for dealing with equalities involving Act.
lemma Act-eq-iff-perm: Act \( f_1 \) α1 \( x_1 \) = Act \( f_2 \) α2 \( x_2 \) \iff \\
\( f_1 = f_2 \wedge (\exists p. \supp x_1 - bn \alpha_1 = \supp x_2 - bn \alpha_2 \wedge (\supp x_1 - bn \alpha_1) \approx p \wedge p \cdot x_1 = x_2 \wedge \supp \alpha_1 - bn \alpha_1 = \supp \alpha_2 - bn \alpha_2 \wedge (\supp \alpha_1 - bn \alpha_1) \approx p \wedge p \cdot \alpha_1 = \alpha_2) \)

(is \( \alpha ?l \) \( \rightarrow \alpha ?r \))

proof

assume \( \ast: \alpha ?l \)

then have \( f_1 = f_2 \)

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

moreover from \( \ast \) obtain \( p \) where alpha: \( (bn \alpha_1, \text{rep-Tree}_\alpha (\text{Rep-formula } x_1)) \approxset (\alpha) (\text{supp-rel } (\alpha)) p (bn \alpha_2, \text{rep-Tree}_\alpha (\text{Rep-formula } x_2)) \) and eq: \( (bn \alpha_1, \alpha_1) \approxset (\alpha) \supp p (bn \alpha_2, \alpha_2) \)

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

from alpha have \( \supp x_1 - bn \alpha_1 = \supp x_2 - bn \alpha_2 \)

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

moreover from alpha have \( (\supp x_1 - bn \alpha_1) \approx p \)

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

moreover from alpha have \( p \cdot x_1 = x_2 \)

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 \( \supp \alpha_1 - bn \alpha_1 = \supp \alpha_2 - bn \alpha_2 \)

by (metis alpha-set.simps)

moreover from eq have \( (\supp \alpha_1 - bn \alpha_1) \approx p \)

by (metis alpha-set.simps)

moreover from eq have \( p \cdot \alpha_1 = \alpha_2 \)

by (simp add: alpha-set.simps)

ultimately show \( \alpha ?r \)

by metis

next

assume \( \ast: \alpha ?r \)

then have \( f_1 = f_2 \)

by metis

moreover from \( \ast \) obtain \( p \) where 1: \( \supp x_1 - bn \alpha_1 = \supp x_2 - bn \alpha_2 \) and 2: \( (\supp x_1 - bn \alpha_1) \approx p \) and 3: \( p \cdot x_1 = x_2 \)

and 4: \( \supp \alpha_1 - bn \alpha_1 = \supp \alpha_2 - bn \alpha_2 \) and 5: \( (\supp \alpha_1 - bn \alpha_1) \approx p \) and 6: \( p \cdot \alpha_1 = \alpha_2 \)

by metis

from 1 2 3 6 have \( (bn \alpha_1, \text{rep-Tree}_\alpha (\text{Rep-formula } x_1)) \approxset (\alpha) (\text{supp-rel } (\alpha)) p (bn \alpha_2, \text{rep-Tree}_\alpha (\text{Rep-formula } x_2)) \)


moreover from 4 5 6 have \( (bn \alpha_1, \alpha_1) \approxset (\alpha) \supp p (bn \alpha_2, \alpha_2) \)

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

ultimately show \( \text{Act } f_1 \alpha_1 x_1 = \text{Act } f_2 \alpha_2 x_2 \)

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

qed

lemma Act-eq-iff-rename: Act \( f_1 \) \( \alpha_1 x_1 \) = Act \( f_2 \) \( \alpha_2 x_2 \) \iff \\
\( f_1 = f_2 \wedge (\exists p. \supp x_1 - bn \alpha_1 = \supp x_2 - bn \alpha_2 \wedge (\supp x_1 - bn \alpha_1) \approx p \wedge p \cdot \alpha_1 = \alpha_2) \)
\[ p \land p \cdot x_1 = x_2 \land \text{supp } \alpha_1 - b_1 \alpha_1 = \text{supp } \alpha_2 - b_2 \alpha_2 \land (\text{supp } \alpha_1 - b_1 \alpha_1) \]
\[ \sharp \ast p \land p \cdot \alpha_1 = \alpha_2 \land \text{supp } p \subseteq b_1 \alpha_1 \cup p \cdot b_1 \alpha_1 \]

(is \ ?l \longleftrightarrow \ ?r)

proof
assume \ ?l then have \( f_1 = f_2 \)
by (metis Act-eq-iff-perm)
moreover from \( \langle ?l \rangle \) obtain \( p \) where \( p: \text{supp } x_1 = b_1 \alpha_1 \land \text{supp } x_2 = b_2 \alpha_2 \land (\text{supp } x_1 - b_1 \alpha_1) \ast\ast p \land p \cdot x_1 = x_2 \land \text{supp } \alpha_1 = b_1 \alpha_1 = \text{supp } \alpha_2 = b_2 \alpha_2 \land (\text{supp } \alpha_1 - b_1 \alpha_1) \ast\ast p \land p \cdot \alpha_1 = \alpha_2 \ast\ast p \land p \cdot \alpha_1 = \alpha_2 \)
by (metis Act-eq-iff-perm)
moreover obtain \( q \) where \( q-p: \forall b \in b_1 \alpha_1. q \cdot b = p \cdot b \) and \( \text{supp } q: \text{supp } q \subseteq b_1 \alpha_1 \cup p \cdot b_1 \alpha_1 \)
by (metis set-renaming-perm2)
have \( \text{supp } q \subseteq \text{supp } p \)
proof
fix \( a \) assume \( a \in \text{supp } q \) then show \( a \in \text{supp } p \)
proof (cases \( a \in b_1 \alpha_1 \))
\begin{enumerate}
  \item case \( \text{True} \) then show \( \text{thesis} \)
  \begin{enumerate}
    \item using \( q-p \) by (metis mem-Collect-eq supp-perm)
  \end{enumerate}
\end{enumerate}
next
\begin{enumerate}
  \item case \( \text{False} \) then have \( \text{thesis} \)
  \begin{enumerate}
    \item using \( \text{target} \)
  \end{enumerate}
\end{enumerate}
qed
qed
with \( p\) have \( (\text{supp } x_1 = b_1 \alpha_1) \ast\ast q \) and \( (\text{supp } \alpha_1 = b_1 \alpha_1) \ast\ast q \)
by (meson fresh-def fresh-star-def subset-if)
moreover with \( p \) and \( q-p\) have \( \forall a. a \in \text{supp } a \Rightarrow q \cdot a = p \cdot a \) and \( \forall a. a \in \text{supp } x_1 \Rightarrow q \cdot a = p \cdot a \)
by (metis Diff-if fresh-def fresh-star-def)
then have \( q \cdot \alpha_1 = p \cdot \alpha_1 \) and \( q \cdot x_1 = p \cdot x_1 \)
by (metis supp-perm-perm-eq)
ultimately show \( ?r \)
using \( \text{supp } q \) by (metis bn-eq)
next
assume \( ?r \) then show \( ?l \)
by (meson Act-eq-iff-perm)
qed

The lifted constructors are free (except for \( \text{Act} \)).

lemma Tree-free [simp]:
shows finite \( \text{supp } xset \) \( \Rightarrow \) \( \text{Conj } xset \neq \text{Not } x \)
and finite \( \text{supp } xset \) \( \Rightarrow \) \( \text{Conj } xset \neq \text{Pred } f \ \varphi \)
and finite \( \text{supp } xset \) \( \Rightarrow \) \( \text{Conj } xset \neq \text{Act } f \ \alpha \ x \)
and \( \text{Not } x \neq \text{Pred } f \ \varphi \)
and \( \text{Not } x_1 \neq \text{Act } f \ \alpha \ x_2 \)
and $\text{Pred } f_1 \varphi \not\equiv \text{Act } f_2 \alpha x$

proof
- show finite (supp xset) $\implies$ Conj xset $\not\equiv$ Not x
  by (metis Conj-rep-eq Not.rep-eq Tree$\alpha$-free(1))
next
show finite (supp xset) $\implies$ Conj xset $\not\equiv$ Pred $f$ $\varphi$
  by (metis Conj-rep-eq Pred.rep-eq Tree$\alpha$-free(2))
next
show finite (supp xset) $\implies$ Conj xset $\not\equiv$ Act $f$ $\alpha x$
  by (metis Conj-rep-eq Act.rep-eq Tree$\alpha$-free(3))
next
show Not $x \not\equiv$ Pred $f$ $\varphi$
  by (metis Not.rep-eq Pred.rep-eq Tree$\alpha$-free(4))
next
show Not $x_1 \not\equiv$ Act $f$ $\alpha x_2$
  by (metis Not.rep-eq Act.rep-eq Tree$\alpha$-free(5))
next
show $\text{Pred } f_1 \varphi \not\equiv \text{Act } f_2 \alpha x$
  by (metis Pred.rep-eq Act.rep-eq Tree$\alpha$-free(6))
qed

14.8 $F/L$-formulas

context effect-nominal-ts
begin
The predicate is-$F/L$-formula will characterise exactly those formulas in a particular set $A$.

inductive is-$F/L$-formula :: $'$effect first $\Rightarrow$ ($'$idx,$'$pred,$'$act,$'$effect) formula $\Rightarrow$ bool
where
  Conj: finite (supp xset) $\Rightarrow$ ($\forall$ x. x $\in$ set-bset xset $\Rightarrow$ is-$F/L$-formula $F$ x) $\Rightarrow$
  is-$F/L$-formula $F$ (Conj xset)
  | Not: is-$F/L$-formula $F$ x $\Rightarrow$ is-$F/L$-formula $F$ (Not x)
  | Pred: $f \in f_s F$ $\Rightarrow$ is-$F/L$-formula $F$ (Pred $f$ $\varphi$)
  | Act: $f \in f_s F$ $\Rightarrow$ bn $\alpha$ $\sharp$ (F,$f$) $\Rightarrow$ is-$F/L$-formula (L ($\alpha$,$F$,$f$)) x $\Rightarrow$ is-$F/L$-formula $F$ (Act $f$ $\alpha$ x)

abbreviation in-$A$ :: ($'$idx,$'$pred,$'$act,$'$effect) formula $\Rightarrow$ $'$effect first $\Rightarrow$ bool
  ($\cdot$ $\in$ A$[$[51,0]$] 50) where
  $x \in A[F]$ $\equiv$ is-$F/L$-formula $F$ x

declare is-$F/L$-formula.induct [case-names Conj Not Pred Act, induct type: formula]

lemma is-$F/L$-formula-eqv [eqv]: $x \in A[F]$ $\Rightarrow$ p $\cdot$ x $\in A[p \cdot F]$
proof (erule is-$F/L$-formula.induct)
  fix xset :: ($'$a,$'$pred,$'$act,$'$effect) formula set$['a$] and $F$
  assume 1: finite (supp xset) and 2: $\forall$ x. x $\in$ set-bset xset $\Rightarrow$ p $\cdot$ x $\in A[p \cdot F]$
  from 1 have finite (supp (p $\cdot$ xset))

104
by (metis permute-finite supp-eqvt)
moreover from 2 have $\forall x. x \in \text{set-bset} \ (p \cdot x \in x \in A[p \cdot F]$
  by (metis (mono-tags) imageE permute-set-eq-image set-bset-eqvt)
ultimately show $p \cdot \text{Conj} x \in A[p \cdot F]$
  using 1 by (simp add: Conj)

next
fix $F$ and $x :: (\text{a}, \text{pred}, \text{act}, \text{effect})$ formula
assume $p \cdot x \in A[p \cdot F]$
then show $p \cdot \text{Not} x \in A[p \cdot F]$
  by (simp add: Not)

next
fix $f$ and $F :: \text{effect} \text{ first}$ and $\varphi$
assume $f \in Fs F$
then show $p \cdot \text{Pred} f \varphi \in A[p \cdot F]$
  by (simp add: Pred)

next
fix $f \alpha$ and $x :: (\text{a}, \text{pred}, \text{act}, \text{effect})$ formula
assume $f \in Fs F$ and $bn \alpha \# \ast (F,f)$ and $p \cdot x \in A[p \cdot L (\alpha, F, f)]$
then show $p \cdot \text{Act} f \alpha x \in A[p \cdot F]$
  by (metis (mono-tags, lifting) Act Act-eqvt L-eqvt Pair-eqvt bn-eqvt fresh-star-permute-iff
member-fs-set-permute-iff)

qed

end

14.9 Induction over infinitary formulas
14.10 Strong induction over infinitary formulas

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

15 Validity With Effects

The following is needed to prove termination of $FL\text{-validTree}$.

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

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

lemma alpha-Tree-relE:
  assumes $(x,y) \in \text{alpha-Tree-rel}$ and $x =_\alpha y \Rightarrow P$
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-eqv' 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 $x :: (x1, x4)$ 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 $= α x4$
    by (rule alpha-Tree-rel-E)
  then obtain tset' where $x4 :: x4 = tConj tset'$ and rel-bset $(= α) tset tset'$
    by (cases $x4$) simp-all
  with * obtain $t' :: t' ∈ set-bset tset'$ and $t = α t'$
    by (metis rel-bset.rep-ep rel-set-def)
  with $x1$ have $(x1, p ∗ t') ∈ alpha-Tree-rel$
    by (metis Tree-$α$.abs-eq-iff alpha-Tree-rel permutation-tree-$α$.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 $tNot t = α x4$
    by (rule alpha-Tree-rel-E)
  then obtain $t' :: x4 = tNot t'$ and $t = α t'$
    by (cases $x4$) simp-all
  with $x1$ have $(x1, p ∗ t') ∈ alpha-Tree-rel$

106
by \((\text{metis } \text{Tree}_{\alpha}, \text{abs-eq-iff } \text{alpha-Tree-relI} \text{ permute-Tree}_{\alpha}, \text{abs-eq } x1)\)
moreover have \((p \cdot t', t') \in \text{hull-rel}\)
by (rule \text{hull-rel.intro})
moreover from \(x4'\) have \((t', x4') \in \text{Tree-wf}\)
using \text{Tree-wf.intro}\((2)\) by blast
ultimately show \((x1,x4) \in \text{alpha-Tree-rel O hull-rel O Tree-wf}\)
by auto

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

qed
with \(x\) show \(x \in \text{alpha-Tree-rel O hull-rel O Tree-wf}\)
by simp

qed

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

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

15.1 Validity for infinitely branching trees

context \text{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) ←→ (∃fP ∈ f. fP ⊢ φ)
| FL-valid-Tree P (tPred 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

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 t.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)

108
by simp

next
assume *: FL-valid-Tree \((p \cdot P) (p \cdot tConj tset)\)
\{
  fix \ t
  assume \ t \in set-bset tset
  with \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
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 \((p \cdot P') (p \cdot t')\)
    by blast
  moreover from * have \( p \cdot (f)P \rightarrow \{ p \cdot α', p \cdot P'\}\)
    by (metis transition-eqvt')
  moreover from * have \( p \cdot tAct f α t =_α tAct (p \cdot f) (p \cdot α') (p \cdot t')\)
    by (metis alpha-Tree-eqvt permute-Tree.simps(4))
  ultimately show FL-valid-Tree \((p \cdot P) (p \cdot tAct f α t)\)
    by auto
next
assume FL-valid-Tree \((p \cdot P) (p \cdot tAct f α t)\)
then obtain α' t' P' where *: \( p \cdot tAct f α t =_α tAct (p \cdot f) α' t' \wedge (p \cdot (f)P) \rightarrow \{α',P'\}\ ∧ FL-valid-Tree P' t'\)
  by auto
then have eq: \(tAct f α t =_α tAct f (−p \cdot α') (−p \cdot t')\)
    by (metis alpha-Tree-eqvt permute-Tree.simps(4) permute-minus-cancel(2))
moreover from * have \((f)P \rightarrow (−p \cdot α',−p \cdot P')\)
    by (metis permute-minus-cancel(2) transition-eqvt')
moreover with 4.IH have FL-valid-Tree \((−p \cdot P') (−p \cdot t')\)
  using eq and * by simp
ultimately show FL-valid-Tree \ P (tAct f α t)
  by auto
qed

qed

lemma FL-valid-Tree-eqvt [eqvt]:
assumes \( FL\text{-valid-Tree} \ P \ t \) shows \( FL\text{-valid-Tree} \ (p \ P) \ (p \ t) \)
using \( \text{assms} \) by \( \text{(metis FL-valid-Tree-eqvt') \)
by (metis Tree α-abs Tree-FL-valid-Tree)
then show ?thesis
  by (simp add: FL-valid-Tree α-def Conj α-def map-bset rep eq)
qed

lemma FL-valid-Tree α-Not [simp]: FL-valid-Tree α P (Not α t α) \iff \neg FL-valid-Tree α P t α
  by transfer simp

lemma FL-valid-Tree α-Pred [simp]: FL-valid-Tree α P (Pred α f \varphi) \iff (f)P
t \varphi
  by transfer simp

lemma FL-valid-Tree α-Act [simp]: FL-valid-Tree α P (Act α f α t α) \iff (\exists \alpha' t α' P'). Act α f α t α = Act α f \alpha' t α' \land (f)P \rightarrow (\alpha',P') \land FL-valid-Tree α P' t α'
proof
  assume FL-valid-Tree α P (Act α f α t α)
  moreover have Act α f α t α = abs-Tree α (tAct f α (rep-Tree α t α))
    by (metis Act α-abs eq Tree α-abs rep)
  ultimately show (\exists \alpha' t α' P'). Act α f α t α = Act α f \alpha' t α' \land (f)P \rightarrow (\alpha',P') \land FL-valid-Tree α P' t α'
    by (metis Act α-abs eq Tree α-abs eq iff FL-valid-Tree simp (4) FL-valid-Tree α-abs eq)
next
  assume (\exists \alpha' t α' P'). Act α f α t α = Act α f \alpha' t α' \land (f)P \rightarrow (\alpha',P') \land FL-valid-Tree α P' t α'
  moreover have (\exists \alpha' t α' P'). Act α f α t α = Act α f \alpha' t α' \land (f)P \rightarrow (\alpha',P') \land FL-valid-Tree α P' t α'
    by (metis Act α-abs eq Tree α-abs rep)
  ultimately show FL-valid-Tree α P (Act α f α t α)
    by (metis Tree α-abs eq iff FL-valid-Tree simp (4) FL-valid-Tree α-abs eq FL-valid-Tree α rep eq)
qed

15.3 Validity for infinitary formulas

lift-definition FL-valid :: 'state \Rightarrow ('idr,'pred,'act,'effect) formula \Rightarrow bool (infix \models 70) is
  FL-valid-Tree α

lemma FL-valid-eqvt [eqvt]:
  assumes P \models x shows (p \cdot P) \models (p \cdot x)
  using assms by transfer (metis FL-valid-Tree α-eqvt)

lemma FL-valid-Conj [simp]:
  assumes finite (supp xset)
  shows P \models Conj xset \iff (\forall x \in set xset \cdot 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 \iff \neg P \models x
  by transfer simp
lemma FL-valid-Pred [simp]: \( P \models \phi \iff \langle f \rangle P \models \phi \)

by transfer simp

lemma FL-valid-Act: \( P \models \text{Act} f \alpha x \iff (\exists \alpha' x' P'). \text{Act} f \alpha x = \text{Act} f \alpha' x' \wedge (\forall f) P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \)

proof
  assume \( P \models \text{Act} f \alpha x \)
  moreover have \( \text{Rep-formula} (\text{Abs-formula} (\text{Act}_\alpha f \alpha (\text{Rep-formula} x))) = \text{Act}_\alpha f \alpha (\text{Rep-formula} x) \)
  by (metis Act_rep-eq Rep-formula_inverse)
  ultimately show \( \exists \alpha' x' P'. \text{Act} f \alpha x = \text{Act} f \alpha' x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \)
  by (auto simp add: FL-valid-def Act_def hereditarily-fs-alpha-renaming)

next
  assume \( \exists \alpha' x' P'. \text{Act} f \alpha x = \text{Act} f \alpha' x' \wedge \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \)
  then show \( P \models \text{Act} f \alpha x \)
  by (metis Act_rep-eq FL-valid_rep-eq FL-valid-Tree Acts Act_def)

qed

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

lemma FL-valid-Act-strong:
  assumes finite (supp X)
  shows \( P \models \text{Act} f \alpha x \iff (\exists \alpha' x' P'). \text{Act} f \alpha x = \text{Act} f \alpha' x' \wedge (\forall f) P \rightarrow \langle \alpha', P' \rangle \wedge P' \models x' \wedge \text{bn } \alpha' \#* X \)

proof
  assume \( P \models \text{Act} f \alpha x \)
  then obtain \( \alpha' x' P' \) where eq: \( \text{Act} f \alpha x = \text{Act} f \alpha' x' \) and valid: \( P' \models x' \)
  by (metis FL-valid-Act)

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

  from fresh-p have supp (supp x' – bn \( \alpha' \)) \#* \( p \) and supp (supp \( \alpha' – bn \alpha' \)) \#* \( p \)
  by (meson fresh-Pair fresh-def fresh-star-def)+
  then have 2: (supp x' – bn \( \alpha' \)) \#* \( p \) and 3: (supp \( \alpha' – bn \alpha' \)) \#* \( p \)
  by (simp add: finite-supp finite-atom-set)+

112
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 \(\langle p \cdot \alpha', p \cdot P' \rangle = (\alpha', P')\)
by (metis abs-residual-pair-eqvt supp-perm-eq)
ultimately show \(\exists \alpha' x' P'. Act f \alpha x = Act f \alpha' x' \land \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \land P'\)
using eq and trans and valid and fresh-X by (metis bn-eqvt FL-valid-eqvt)
next
assume \(\exists \alpha' x' P'. Act f \alpha x = Act f \alpha' x' \land \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \land P' \models x' \land bn \alpha' \sharp X\)
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 \(P \models Act f \alpha x \iff (\exists P', \langle f \rangle P \rightarrow \langle \alpha', P' \rangle \land P' \models x)\)
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: \(\langle f \rangle P \rightarrow \langle \alpha', P' \rangle \land P' \models x'\)
and fresh: \(bn \alpha' \sharp* \langle f \rangle P\)
by (metis FL-valid-Act-strong)
from eq obtain \(p\) where \(p-\alpha: \alpha' = p \cdot \alpha\) and \(p-x: x' = p \cdot x\) and supp-p:
supp \(p \subseteq bn \alpha \cup p \cup bn \alpha\)
by (metis Act-eq-iff-perm-renaming)
from assms and fresh have \((bn \alpha \cup p \cup bn \alpha) \sharp* \langle f \rangle P\)
using p-\alpha 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-\alpha 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 \((\exists P', \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \land P' \models x)\)
by meson

113
next
  assume \( \exists P'. (f) P \rightarrow (\alpha.P') \land P' \models x \)
  then show \( P \models Act f \alpha x \)
    by (metis FL-valid-Act)
  qed

end

end

theory FL-Logical-Equivalence
imports
  FL-Validity
begin

16 (Strong) Logical Equivalence

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

locale indexed-effect-nominal-ts = effect-nominal-ts satisfies transition effect-apply
for satisfies :: 'state::fs \Rightarrow 'pred::fs \Rightarrow bool (infix \models 70)
and transition :: 'state \Rightarrow ('act::bn,'state) residual \Rightarrow bool (infix \models 70)
and effect-apply :: 'effect::fs \Rightarrow 'state \Rightarrow '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 \Rightarrow 'state \Rightarrow 'state \Rightarrow bool where
    FL-logically-equivalent F P Q \equiv
      \forall x::('idx,'pred,'act,'effect) formula. x \in A[F] \rightarrow (P \models x \iff Q \models x)

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

end

end

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

17 \( F/L \)-Bisimilarity Implies Logical Equivalence

context indexed-effect-nominal-ts
begin

  lemma FL-bisimilarity-implies-equivalence-Act:
    assumes \( f \in fs \)

and $bn \alpha \triangleright (F, f)$
and $x \in A[L (\alpha, F, f)]$
and $\bigwedge P. Q. P \sim[L (\alpha, F, f)] Q \implies P \models x \iff Q \models x$
and $P \sim[F] Q$
and $P \models \text{Act} f \alpha x$
shows $Q \models \text{Act} f \alpha x$

proof —

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

from $(P \sim[F] Q)$ and $(f \in_{fs} F)$ and fresh and trans obtain $Q'$ where
trans': $(f) Q \sim \langle (\alpha', Q') \rangle$ and bisim': $P' \sim[L (\alpha', F, f)] Q'$
by (metis FL-bisimilar-simulation-step)

from $eq$ obtain $p$ where $p-\alpha: \alpha' = p \cdot \alpha$ and $p-x: x' = p \cdot x$
and fresh-p: $(\text{supp} x - bn \alpha) \triangleright p \land (\text{supp} \alpha - bn \alpha) \triangleright p$
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 $p-\alpha$ have $(p \cdot bn \alpha) \triangleright (F, f)$
by (simp add: bn-eqvt fresh-Star-Pair)
with $bn \alpha \triangleright (F, f)$ and supp-p have $\text{supp} (F, f) \triangleright p$
by (meson UnE fresh-def fresh-star-def subsetCE)
then have $\text{supp} F \triangleright p$ and $\text{supp} f \triangleright p$
by (simp add: fresh-star-Un supp-Pair)+

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

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

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

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

qed

theorem FL-bisimilarity-implies-equivalence: assumes $P \sim[F] Q$ shows FL-logically-equivalent $F \models P Q$

unfolding FL-logically-equivalent-def proof

fix $x : (\text{idx}, \text{pred}, \text{act}, \text{effect})$ formula
show $x \in A[F] \rightarrow P \models x \leftrightarrow Q \models x$

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

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

18 Logical Equivalence Implies F/L-Bisimilarity

context indexed-effect-nominal-ts
begin

  definition is-distinguishing-formula :: 
    (\ indexing\ wirk, \ pred, \ act, \ effect) \ formula \Rightarrow \ \ state \Rightarrow \ \ state \Rightarrow \ bool
  where
    $x$ distinguishes $P$ from $Q \equiv P \models x \land \neg Q \models x$

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

  lemma FL-equivalent-iff-not-distinguished:
    $FL$-logically-equivalent $F P Q \iff \neg \exists x. x \in A[F] \land x$ distinguishes $P$ from $Q$
  by (meson FL-logically-equivalent-def Not is-distinguishing-formula-def FL-valid-Not)
There exists a distinguishing formula for $P$ and $Q$ in $A[F]$ whose support is contained in $\text{supp } (F, P)$.

**lemma** FL-distinguished-bounded-support:

**assumes** $x \in A[F]$ and $x$ distinguishes $P$ from $Q$

**obtains** $y$ where $y \in A[F]$ and $\text{supp } y \subseteq \text{supp } (F, P)$ and $y$ distinguishes $P$ from $Q$

**proof**

- let $?B = \{ p \cdot x | p. \text{supp } (F, P) \# p \}$
- have $\text{supp } (F, P)$ supports $?B$
- unfolding supports-def

**proof** (clarify)

fix $a$ $b$

assume $a: a \notin \text{supp } (F, P)$ and $b: b \notin \text{supp } (F, P)$

have $(a \rightleftharpoons b) \cdot ?B \subseteq ?B$

**proof**

fix $x'$

assume $x' \in (a \rightleftharpoons b) \cdot ?B$

then obtain $p$ where 1: $x' = (a \rightleftharpoons b) \cdot p \cdot x$ and 2: $\text{supp } (F, P) \# p$

by (auto simp add: permute-set-def)

let $?q = (a \rightleftharpoons b) + p$

from 1 have $x' = ?q \cdot x$

by simp

moreover from $a$ and $b$ and 2 have $\text{supp } (F, P) \# ?q$

by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))

ultimately show $x' \in ?B$ by blast

qed

moreover have $?B \subseteq (a \rightleftharpoons b) \cdot ?B$

**proof**

fix $x'$

assume $x' \in ?B$

then obtain $p$ where 1: $x' = p \cdot x$ and 2: $\text{supp } (F, P) \# p$

by auto

let $?q = (a \rightleftharpoons b) + p$

from 1 have $x' = (a \rightleftharpoons b) \cdot ?q \cdot x$

by simp

moreover from $a$ and $b$ and 2 have $\text{supp } (F, P) \# ?q$

by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))

ultimately show $x' \in (a \rightleftharpoons b) \cdot ?B$

using mem-permute-iff by blast

qed

ultimately show $(a \rightleftharpoons b) \cdot ?B = ?B$...

qed

then have $\text{supp-B-subset-supp-P}$: $\text{supp } ?B \subseteq \text{supp } (F, P)$

by (metis (erased, lifting) finite-supp supp-is-subset)

then have $\text{finite-supp-B}$: $\text{finite } (\text{supp } ?B)$

using finite-supp rev-finite-subset by blast

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

by auto

then have $|?B| \leq 0 |\text{UNIV} :: \text{perm set}|$
by (rule surj-imp-ordLeq)
also have \(|UNIV :: \text{perm set}| \prec o |UNIV :: 'idx set|\)
by (metis card-idx-perm)
also have \(|UNIV :: 'idx set| \leq o \text{natLeq} + c |UNIV :: 'idx set|\)
by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-B: \(|?B| \prec o \text{natLeq} + c |UNIV :: 'idx set|\).

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

from finite-supp-B and card-B and supp-B-subset-supp-P have supp \(?y \subseteq supp (F,P)\)
by simp
moreover have \(?y \in A[F]\)
proof
  show finite (supp (Abs-bset ?B :: (-,-,-) formula set['idx]))
  using finite-supp-B card-B by simp
next
  fix \(x'\)
  assume \(x' \in \text{set-bset} (Abs-bset ?B :: (-,-,-) formula set['idx])\)
  then obtain \(p\) where \(p-x: x' = p \cdot x\) and fresh-p: supp \((F,P) \#^\ast p\)
  using card-B by auto
  from fresh-p have \(p \cdot F = F\)
  using fresh-star-Pair fresh-star-supp-conv perm-supp-eq by blast
  with \(x \in A[F]\) show \(x' \in A[F]\)
  using p-x by (metis is-FL-formula-eqvt)
qed
moreover have \(?y\) distinguishes \(P\) from \(Q\)
unfolding is-distinguishing-formula-def proof
  from \(x\) distinguishes \(P\) from \(Q\) show \(P \models \?y\)
  by (auto simp add: card-B finite-supp-B) (metis is-distinguishing-formula-def fresh-star-Un supp-Pair supp-perm-eq FL-valid-eqvt)
next
  from \(x\) distinguishes \(P\) from \(Q\) show \(\neg Q \models \?y\)
  by (auto simp add: card-B finite-supp-B) (metis is-distinguishing-formula-def permute-zero fresh-star-zero)
qed
ultimately show \(?\thesis\)
  using that by blast
qed

lemma \(\text{FL-equivalence-is-L-bisimulation}: \text{is-L-bisimulation FL-logically-equivalent}\)
proof –
\{ 
  fix \(F\) have symp (FL-logically-equivalent \(F\))
  by (rule sympI) (metis FL-logically-equivalent-def)
\}
moreover
\{ 
  fix \(F P Q f \varphi\)
assume FL-logically-equivalent $F P Q$ and $f \in_{f^*} F$ and $(f)P \vdash \varphi$
then have $(f)Q \vdash \varphi$
  by (metis FL-logically-equivalent-def Pred FL-valid-Pred)
}\nmoreover
\{\n  fix $F P Q f \alpha P'$
assume FL-logically-equivalent $F P Q$ and $f \in_{f^*} F$ and $bn \alpha \sharp^* ((f)Q, F, f)$ and $(f)P \rightarrow \langle \alpha, P' \rangle$
then have $\exists Q'. (f)Q \rightarrow \langle \alpha, Q' \rangle \land FL-logically-equivalent \langle L (\alpha, F, f) \rangle P' Q'$
proof -
  \{\n  let $?Q' = \{ Q'. (f)Q \rightarrow \langle \alpha, Q' \rangle \}$
  assume $\forall Q' \in {?Q'}. \neg FL-logically-equivalent \langle L (\alpha, F, f) \rangle P' Q'$
  then have $\forall Q' \in {?Q'}. \exists x : ('idx, 'pred, 'act, 'effect) formula. x \in A[L (\alpha, F, f)] \land x \text{ distinguishes } P' \text{ from } Q'$
  by (metis FL-equivalent-iff-not-distinguished)
  then have $\forall Q' \in {?Q'}. \exists x : ('idx, 'pred, 'act, 'effect) formula. x \in A[L (\alpha, F, f)] \land supp x \subseteq supp \langle L (\alpha, F, f), P' \rangle \land x \text{ distinguishes } P' \text{ from } Q'$
  by (metis FL-distinguished-bounded-support)
  then obtain $g :: 'state \Rightarrow ('idx, 'pred, 'act, 'effect) formula \text{ where}$
  $\forall Q' \in {?Q'}. g Q' \in A[L (\alpha, F, f)] \land supp (g Q') \subseteq supp \langle L (\alpha, F, f), P' \rangle \land (g Q') \text{ distinguishes } P' \text{ from } Q'$
  by metis
  have $\text{supp } (g \cdot {?Q'}) \subseteq supp \langle L (\alpha, F, f), P' \rangle$
    by (rule set-bounded-supp, fact finite-suppr, cut-tac \#, blast)
  then have $\text{finite-suppr-image: finite } (\text{supp } (g \cdot {?Q'}))$
    using finite-suppr rev-finite-subset by blast
  have $|{g \cdot {?Q'}}| \leq o |UNIV :: 'state set|$
    by (metis card-of-UNIV card-of-image ordLeg-transitive)
  also have $|UNIV :: 'state set| < o |UNIV :: 'idx set|$
    by (metis card-idx-state)
  also have $|UNIV :: 'idx set| \leq o \text{natLeq } + c |UNIV :: 'idx set|$
    by (metis CardOfZero-UNIV OrdLeq-Csum2)
  finally have $\text{card-image: } |g \cdot {?Q'}| < o \text{natLeq } + c |UNIV :: 'idx set|$
  let $?y = \text{Conj} (\text{Abs-bset } (g \cdot {?Q'})) :: ('idx, 'pred, 'act, 'effect) formula$
  have $\text{Act } f \alpha \ ?y \in A[F]$
  proof
    from $f \in_{f^*} F$ show $f \in_{f^*} F$.
  next
    from $\text{bn } \alpha \sharp^* ((f)Q, F, f)$ show $\text{bn } \alpha \sharp^* (F, f)$
      using fresh-star-Pair by blast
  next
    show $\text{Conj} (\text{Abs-bset } (g \cdot {?Q'})) \in A[L (\alpha, F, f)]$
    proof
      show finite $(\text{supp } (\text{Abs-bset } (g \cdot {?Q'})) :: ('-, -, -, -) \text{ formula set["idx"]})$
        using finite-suppr-image card-image by simp
    next
    fix $x'
assume \( x' \in \text{set-set} \) \((\text{Abs}\)-\text{set}\( \langle g \, ?Q' \rangle \) :: (\_,\_,\_,\_) \text{ formula set}[\text{idx}])\
then obtain \( Q' \) where \( x' = g \, Q' \) and \((f)\, Q \rightarrow \langle \alpha, Q' \rangle \)
using \text{card-image} \text{ by aut} \text{ o}
with * show \( x' \in A[L(\alpha, F, f)] \)
using \text{mem-Collect-eq} \text{ by blast} 
qed 

moreover have \( P \models \text{Act} f \, \alpha \, ?y \)
unfolding \text{FL-valid-Act} \text{ proof} \text{ (standard+)}
show \( \langle f \rangle\, P \rightarrow \langle \alpha, P' \rangle \) \text{ by fact} 
next 
\{
x'\text{ \text{fix}} Q'\text{ \text{assume} \( \langle f \rangle\, Q \rightarrow \langle \alpha, Q' \rangle \)}
with * have \( P' \models g \, Q' \)
by \text{(metis is-distinguishing-formula-def mem-Collect-eq)}
\}
then show \( P' \models ?y \)
by \text{(simp add: finite-suppp-image card-image) }
qed 

moreover have \( \neg \models Q \models \text{Act} f \, \alpha \, ?y \)
proof 
assume \( Q \models \text{Act} f \, \alpha \, ?y \)
then obtain \( Q' \) where 1: \( (f)\, Q \rightarrow \langle \alpha, Q' \rangle \) and 2: \( Q' \models ?y \)
using \text{bn} \, \alpha \, \{*\} \text{ \( (f)\, Q, F, f )\} \text{ \text{by} \text{ (metis fresh-star-Pair FL-valid-Act-fresh) \text{ from 2 have} \( \wedge Q'''. \, (f)\, Q \rightarrow \langle \alpha, Q'''\rangle \longrightarrow Q' \models g \, Q'' \)
by \text{(simp add: finite-suppp-image card-image)}
with 1 \text{ and } * \text{ show False \text{ by blast}} 
\text{ultimately have False}
by \text{(metis \langle FL-logically-equivalent F P Q \rangle FL-logically-equivalent-def) \text{)} \}
then show \( ?\text{thesis} \) \text{ by aut} 
qed 
\}
ultimately show \( ?\text{thesis} \) \text{ by aut} 
unfolding \text{is-L-bisimulation-def} \text{ by metis} 
qed }

\textbf{theorem} \text{FL-equivalence-implies-bisimilarity:} \textbf{assumes} \text{FL-logically-equivalent F P Q shows P \sim [F] Q }
\text{using} \text{assms} \text{ by} \text{ \text{ (metis FL-bisimilar-def FL-equivalence-is-L-bisimulation) \text{) \text{)}}} 
\textbf{end} 
\textbf{end} 
\textbf{theory L-Transform} 
\textbf{imports} 
\textbf{end} 
\footnotesize

120
Validity
Bisimilarity-Implies-Equivalence
FL-Equivalence-Implies-Bisimilarity

begin

19 L-Transform

19.1 States

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

datatype ('state,'effect) L-state =
    AC 'effect × '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)

end
by (cases x) (simp add: finite-suppp)+
qed
end

19.2 Actions and binding names

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

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

  fun permute-L-action :: perm ⇒ ('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-suppp)+
  qed

end
instantiation $L$-action :: \((bn, fs) \ bn\)

begin

fun \(bn\)-L-action :: \(('a,'b) \ L\)-action \Rightarrow \ atom \ set\) where
\(bn\)-L-action \((\text{Act} \ \alpha)\) = \(bn\ \alpha\)
| \(bn\)-L-action \((\text{Eff} \ -)\) = \{

instance

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

end

19.3 Satisfaction

context effect-nominal-ts

begin

fun \(L\)-satisfies :: \((\text{'state,'effect}) \ L\)-state \Rightarrow \ 'pred \Rightarrow \ \text{bool}\) (infix \(\vdash_{L}\) 70) where
\(AC\ (\cdot\cdot\cdot,P) \vdash_{L} \varphi \leftrightarrow P \vdash \varphi\)
| \(EF\ -\ \vdash_{L} \varphi \leftrightarrow False\)

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 \(L\)-satisfies-eqvt snd-eqvt)
  then show \(?thesis\)
    using \(AC\) by (metis \(L\)-satisfies.simps(1) permute-L-state.simps(1) prod.collapse)
next
  case \(EF\)
  with assms have False
    by simp
  then show \(?thesis\) ..
qed

end

19.4 Transitions

context effect-nominal-ts
begin

fun L-transition :: ('state, 'effect) L-state ⇒ (('act, 'effect) L-action, ('state, 'effect) L-state) residual ⇒ bool (infix ⇒_L 70) where

AC (f,F,P) ⇒_L αP' ←→ (∃ α P. P ⇒ ⟨α,P⟩ ∧ αP' = ⟨Act α, EF (L (α,F,f), P')⟩) ∧ bn α ♯ (F,f) — note the freshness condition

| EF (F,P) ⇒_L αP' ←→ (∃ f ∈ fs F ∧ αP' = ⟨Eff f, AC (f, F, ⟨f⟩P)⟩)

lemma L-transition-eqvt: assumes P_L ⇒_L αL P_L' shows (p ⋅ P_L) ⇒_L (p ⋅ αL 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 ⇒ ⟨α,P⟩ and αP': αL P_L' = ⟨Act α, EF (L (α,F,f), P')⟩
  by auto
  from trans have p ⋅ P ⇒ ⟨p ⋅ α, p ⋅ P⟩
  by (simp add: transition-eqvt)
  moreover from αP' have p ⋅ αL P_L' = ⟨Act (p ⋅ α), EF (L (p ⋅ α, p ⋅ F, p ⋅ f), p ⋅ P')⟩
  by (simp add: L-eqvt)
  moreover from fresh have bn (p ⋅ α) ♯ (p ⋅ F, p ⋅ f)
    by (metis bn-eqvt fresh-star-Pair fresh-star-permute-iff)
  ultimately have p ⋅ P_L ⇒_L p ⋅ αL P_L'
    using * by auto
  }
  with AC show thesis
  by (metis transp_collapse)
}

next

case EF

{ fix F P
  assume *: P_L = EF (F,P)
  with assms obtain f where f ∈ fs F and αL P_L' = ⟨Eff f, AC (f, F, ⟨f⟩P)⟩
  by auto
  then have (p ⋅ f) ∈ fs (p ⋅ F) and p ⋅ αL P_L' = ⟨Eff (p ⋅ f), AC (p ⋅ f, p ⋅ F, (p ⋅ f)(p ⋅ P))⟩
  by simp+
  then have p ⋅ P_L ⇒_L p ⋅ αL P_L'
    using * L-transition.simps(2) Pair-eqvt permute-L-state.simps(2) by force
  }
  with EF show thesis
  by (metis transp_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) →L (αL.PL′)
  shows ∃α P′. P → ⟨α,P′⟩ ∧ ⟨αL,PL′⟩ = ⟨Act α, EF (L (α,F,f), P′)⟩ ∧ bn α
  using assms proof –
  from AC (f,F,P) →L (αL.PL′) obtain α P′ where transition: P → ⟨α,P′⟩
  and alpha: ⟨αL,PL′⟩ = ⟨Act α, EF (L (α,F,f), P′)⟩ and fresh: bn α * (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 ((?Act, EF (L (α,F,f), P′)), ⟨α,P′⟩, F, f))
  by (metis finite-Diff finite-UnI finite-supp supp-Pair supp-abs-residual-pair)
  moreover from fresh have bn α * ((?Act, EF (L (α,F,f), P′)), ⟨α,P′⟩, F, f)
  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, EF (L (α,F,f), P′)), ⟨α,P′⟩, F, f) * p
  by (metis at-set-avoiding2)
  then have supp (?Act, EF (L (α,F,f), P′)) * p and supp ⟨α,P′⟩ * p and supp (F,f) * p
  by (metis fresh-star-Un supp-Pair+)
  then have p · (?Act, EF (L (α,F,f), P′)) = ⟨?Act, EF (L (α,F,f), P′)⟩ and p · ⟨α,P′⟩ = ⟨α,P′⟩ and p · (F,f) = (F,f)
  by (metis supp-perm-eqv)+
  then have ⟨Act (p · α), EF (L (p · α, F, f), p · P′)⟩ = ⟨?Act, EF (L (α,F,f), P′)⟩ and
  ⟨p · α, p · P′⟩ = ⟨α,P′⟩
  using permute-L-action.simps(1) permute-L-state.simps(2) abs-residual-pair-eqvt
  L-eqvt' Pair-eqvt by auto
  then show ∃α P′. P → ⟨α,P′⟩ ∧ ⟨αL,PL′⟩ = ⟨Act α, EF (L (α,F,f), P′)⟩ ∧
  bn α * X
  using transition and alpha and fresh-X by (metis bn-eqvt)
qed

lemma L-transition-AC-fresh:
  assumes bn α * (F,f,P)
  shows AC (f,F,P) →L ⟨Act α, PL′⟩ ĕ (∃P′. PL′ = EF (L (α,F,f), P′) ∧ P → ⟨α,P′⟩)
proof
  assume AC (f,F,P) →L ⟨Act α, PL′⟩
  moreover have finite (supp (F,f,P))
  by (fact finite-supp)
  ultimately obtain α P′ where trans: P → ⟨α,P′⟩ and eq: ⟨Act α ::
  (act,effect) L-action, PL′⟩ = ⟨Act α′, EF (L (α,F,f), P′)⟩ and fresh: bn α'
  * (F,f,P)

125
using \textit{L-transition-AC-strong} by \texttt{blast}

from eq obtain p where p: p \cdot \langle \text{act}'\rangle = \langle \text{act},'effect\rangle \, L\text{-action}, \, P_L' = \langle \text{act}', EF \, (L \, \langle \text{act}',F,f\rangle, \, P') \rangle \text{ and supp-p: supp p} \subseteq \text{bn (Act \, \langle \text{act},'effect\rangle \, L\text{-action})} \\
\cup \, p \cdot \text{bn (Act \, \langle \text{act},'effect\rangle \, L\text{-action})} \\
\text{using residual-eq-iff-perm-renaming by \texttt{metis}}

from p have p-\alpha: p \cdot \alpha = \alpha' \text{ and } p-P_L': p \cdot P_L' = EF \, (L \, \langle \text{act}',F,f\rangle, \, P') \\
\text{by \texttt{simp-all}}

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

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

from p-P_L' have P_L' = \neg p \cdot EF \, (L \, \langle \text{act}',F,f\rangle, \, P') \\
\text{by (metis permute-minus-cancel(2))}

then have \quad\quad\quad\quad P_L' = EF \, (L \, \langle \text{act},F,f\rangle, \neg p \cdot P') \\
\text{using p-\alpha p-F p-f by simp (metis (full-types) permute-minus-cancel(2))}

moreover from trans have P \rightarrow \langle \alpha, \neg p \cdot P' \rangle \\
\text{using p-P and p-\alpha by (metis permute-minus-cancel(2) transition-eqv)}

ultimately show \exists P'. \, P_L' = EF \, (L \, \langle \text{act},F,f\rangle, \, P') \wedge P \rightarrow \langle \alpha,P' \rangle \\
\text{by \texttt{blast}}

next

assume \exists P'. \, P_L' = EF \, (L \, \langle \text{act},F,f\rangle, \, P') \wedge P \rightarrow \langle \alpha,P' \rangle \\
moreover from assms have bn \alpha \quad*\quad (F,f) \\
\text{by (simp add: fresh-star-Pair)}

ultimately show \langle \alpha \rangle \rightarrow_{L} (Act \, \langle \alpha, P_L' \rangle) \\
\text{using \texttt{L-transition.simps(1) by blast}}

qed

end

19.5 Translation of $F/L$-formulas into formulas without effects

Since we defined formulas via a manual quotient construction, we also need to define the $L$-transform via lifting from the underlying type of infinitely branching trees. As before, we cannot use \texttt{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 $\alpha$-equivalence) rather than formulas. This allows us to prove equivariance for \textit{all} argument trees, without an assumption that they are (hereditarily) finitely supported. Further below–after this auxiliary function has been lifted to $F/L$-formulas as arguments–we derive a version that returns formulas.
**primrec** \( L\text{-}\text{transform-Tree} :: (\text{idx, 'pred}' :: fs, 'act' :: bn, 'eff' :: fs) \Rightarrow (\text{idx, 'pred}', ('act', 'eff')) \text{ L-}\text{action} \) Formula.Tree, \( \text{where} \)

\[
L\text{-}\text{transform-Tree}(\text{tConj tset}) = \text{Formula Conj}_\alpha (\text{map-bset } L\text{-}\text{transform-Tree} \text{ tset}) \\
L\text{-}\text{transform-Tree}(\text{tNot t}) = \text{Formula Not}_\alpha (L\text{-}\text{transform-Tree} \text{ t}) \\
L\text{-}\text{transform-Tree}(\text{tPred f } \varphi) = \text{Formula Act}_\alpha (\text{Eff f}) (\text{Formula Pred}_\alpha \varphi) \\
L\text{-}\text{transform-Tree}(\text{tAct f } \alpha \ t) = \text{Formula Act}_\alpha (\text{Eff f}) (\text{Formula Act}_\alpha (\text{Act } \alpha) (L\text{-}\text{transform-Tree} \text{ t}))
\]

**lemma** \( L\text{-}\text{transform-Tree-eqvt eqvt mem-Collect-eq \subset \text{subsetI supp-rel-def}} \)

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

**proof** (induct t)
- case (tConj tset)
  - then show \(?\text{case}\)
    - by simp (metis (no-types, hide-lams) bset.map-cong0 map-bset-eqvt permute-fun-def permute-minus-cancel(1))
  - qed simp-all

\( L\text{-}\text{transform-Tree} \text{ respects } \alpha\text{-equivalence.} \)

**lemma** \( \alpha\text{-Tree-L\text{-}transform-Tree}: \)
- assumes \( \alpha\text{-Tree } \text{t1 t2} \)
- shows \( L\text{-}\text{transform-Tree} \text{ t1} = L\text{-}\text{transform-Tree} \text{ t2} \)
- using \( \text{assms} \) proof (induction \( \text{t1 t2} \) rule: \( \alpha\text{-Tree-induct}' \))
  - case (alpha-tConj tset1 tset2)
    - then have \( \text{rel-bset } (=) (\text{map-bset } L\text{-}\text{transform-Tree} \text{ tset1}) (\text{map-bset } L\text{-}\text{transform-Tree} \text{ tset2}) \)
      - by (simp add: bset.rel-map(1) bset.rel-map(2) bset.rel-mono-strong)
    - then show \(?\text{case}\)
      - by (simp add: bset.rel-eq)
  - next
    - case (alpha-tAct f1 α1 t1 f2 α2 t2)
    - from \( \alpha\text{-Tree} (\text{FL-Formula.Tree.tAct f1 α1 t1}) (\text{FL-Formula.Tree.tAct f2 α2 t2}) \)
      - obtain p where \(*: (\text{bn } \alpha1, \text{ t1}) \approxset \alpha\text{-Tree} (\text{supp-rel } \alpha\text{-Tree} ) \text{ p} (\text{bn } \alpha2, \text{ t2}) \)
      - and \(**: (\text{bn } \alpha1, \text{ t1}) \approxset (=) \text{ supp p (bn } \alpha2, \text{ t2}) \text{ and } f1 = f2 \)
    - by auto
    - from \( * \) have \( \text{fresh'} : (\text{supp-rel } \alpha\text{-Tree } \text{ t1} - \text{ bn } \alpha1) \approxstar p \text{ and } \alpha\text{: } \alpha\text{-Tree} (\text{p } \cdot \text{t1}) \text{ t2 and } \text{eq: p } \cdot \text{ bn } \alpha1 = \text{ bn } \alpha2 \)
      - by (auto simp add: alpha-set)
    - from \( \alpha\text{-Tree.IH}(2) \) have \( \text{supp-rel } \text{Formula.alpha-Tree} (\text{Formula.rep-Tree}_\alpha (L\text{-}\text{transform-Tree } \text{t1})) \subseteq \text{supp-rel } \alpha\text{-Tree } \text{t1} \)
      - 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 \( \text{fresh} \) have \( \text{fresh'} : (\text{supp-rel } \text{Formula.alpha-Tree} (\text{Formula.rep-Tree}_\alpha (L\text{-}\text{transform-Tree } \text{t1}))) \subseteq \text{bn } \alpha1 \approxstar p \)
      - by (meson DiffD1 DiffD2 DiffI fresh-star-def subsetCE)
    - moreover from \( \alpha\text{: } \alpha\text{-Tree} (\text{p } \cdot \text{Formula.rep-Tree}_\alpha (L\text{-}\text{transform-Tree } \text{t1}))) (\text{Formula.rep-Tree}_\alpha (L\text{-}\text{transform-Tree } \text{t2})) \)
      - using \( \alpha\text{-Tree.IH}(1) \) by (metis Formula.alpha-Tree-permute-rep-commute)
moreover from fresh' alpha' eq have supp_rel Formula.alpha-Tree (Formula.rep-Tree_\alpha (L-transform-Tree t1)) - bn \alpha 1 = supp_rel Formula.alpha-Tree (Formula.rep-Tree_\alpha (L-transform-Tree t2)) - bn \alpha 2
  by (metis (mono-tags) Diff-eqvt Formula.alpha-Tree-eqvt' Formula.alpha-Tree-eqvt-aux Formula.alpha-Tree-supp-rel atom-set-perm-eq)

ultimately have (bn \alpha 1, Formula.rep-Tree_\alpha (L-transform-Tree t1)) \approx set Formula.alpha-Tree (supp_rel Formula.alpha-Tree) p (bn \alpha 2, Formula.rep-Tree_\alpha (L-transform-Tree t2))
  using eq by (simp add: alpha-set)
moreover from ** have (bn \alpha 1, Act \alpha 1) \approx set (=) supp p (bn \alpha 2, Act \alpha 2)
  by (metis (mono-tags, lifting) L-Transform.supp-Act alpha-set permute-L-action.simps(1))
ultimately have Formula.Act_\alpha (Act \alpha 1) (L-transform-Tree t1) = Formula.Act_\alpha (Act \alpha 2) (L-transform-Tree t2)
  by (auto simp add: Formula.Act_\alpha-eq-iff)
with \(f1 = f2\); show ?case
  by simp
qed simp-all

L-transform for trees modulo \alpha-equivalence.

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

lemma L-transform-Tree_\alpha-eq [eqvt]: p \cdot L-transform-Tree_\alpha t_\alpha = L-transform-Tree_\alpha (p \cdot t_\alpha)
  by (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-conv0 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 \varphi) = Formula.Act_\alpha (Eff f) (Formula.Pred_\alpha \varphi)
  by transfer simp

lemma L-transform-Tree_\alpha-Act_\alpha [simp]: L-transform-Tree_\alpha (Act_\alpha f \alpha t_\alpha) = Formula.Act_\alpha (Eff f) (Formula.Act_\alpha (Act \alpha) (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))

128
proof
have eqvt map-bset and eqvt L-transform-Tree_α
  by (simp add: eqvtI)+
then have supp (map-bset L-transform-Tree_α) = {}
  using supp-fun-eqvt supp-fun-app-eqvt by blast
then have supp (map-bset L-transform-Tree_α tset_α) ⊆ supp tset_α
  using supp-fun-app by blast
with assms show finite (supp (map-bset L-transform-Tree_α tset_α))
  by (metis finite-subset)
qed

lemma L-transform-Tree_α-preserves-hereditarily-fs:
  assumes hereditarily-fs t_α
  shows Formula.hereditarily-fs (L-transform-Tree_α t_α)
using assms proof (induct rule: hereditarily-fs.induct)
case (Conj tset_α)
  then show ?case
    by (auto intro!: Formula.hereditarily-fs.Conj_α) (metis imageE map-bset.rep-eq)
next
case (Not t_α)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Not_α)
next
case (Pred f ϕ)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Act_α Formula.hereditarily-fs.Pred_α)
next
case (Act t_α f α)
  then show ?case
    by (simp add: Formula.hereditarily-fs.Act_α)
qed

L-transform for F/L-formulas.

lift-definition L-transform-formula :: ('idx::fs,'pred::bn,'act::bn,'eff::fs) formula ⇒
('idx, 'pred, ('act,'eff) L-action) Formula.Tree_α is
L-transform-Tree_α
.

lemma L-transform-formula-eqvt [eqvt]: p · L-transform-formula x = L-transform-formula (p · x)
  by transfer (simp)

lemma L-transform-formula-Conj [simp]:
  assumes finite (supp xset)
  shows L-transform-formula (Conj xset) = Formula.Conj_α (map-bset L-transform-formula xset)
  using assms by (simp add: Conj-def L-transform-formula-def bset.map-comp map-fun-def)
lemma $L$-transform-formula-Not [simp]: $L$-transform-formula $(\text{Not } x) = \text{Formula.Not}_{\alpha}$ ($L$-transform-formula $x$)
  by transfer simp

lemma $L$-transform-formula-Pred [simp]: $L$-transform-formula $(\text{Pred } \phi f \alpha x) = \text{Formula.Act}_{\alpha} (\text{Eff } f) (\text{Formula.Pred}_{\alpha} \phi)$
  by transfer simp

lemma $L$-transform-formula-Act [simp]: $L$-transform-formula $(\text{FL-Formula.Act } f \alpha x) = \text{Formula.Act}_{\alpha} (\text{Eff } f) (\text{Formula.Act}_{\alpha} (\text{Act } f) (L$-transform-formula $x))$
  by transfer simp

lemma $L$-transform-formula-hereditarily-fs [simp]: $\text{Formula.hereditarily-fs} (L$-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 $:: ('idx,'pred::fs,'act::bn,'eff::fs) \text{ formula } \Rightarrow ('idx,'pred, ('act,'eff) \text{ L-action}) \text{ Formula.formula}$  where
$L$-transform $x = \text{Formula.Abs-formula} (L$-transform-formula $x$)

lemma $L$-transform-eqvt [eqvt]: $p \cdot L$-transform $x = L$-transform $((p \cdot x)$
  unfolding $L$-transform-def by simp

lemma finite-suppp-map-bset-L-transform [simp]:
  assumes finite $\text{(supp } xset)$
  shows finite $\text{(supp (map-bset L-transform } xset))$
proof –
  have eqvt map-bset and eqvt $L$-transform
    by (simp add: eqvtI)+
  then have supp $\text{(map-bset L-transform)} = {}$
    using supp-fun-eqvt supp-fun-app-eqvt by blast
  then have supp $\text{(map-bset L-transform } xset) \subseteq \text{supp } xset$
    using supp-fun-app by blast
  with assms show finite $\text{(supp (map-bset L-transform } xset))$
    by (metis finite-subset)
qed

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

lemma $L$-transform-Not [simp]: $L$-transform $\text{(Not } x) = \text{Formula.Not} (L$-transform $x)$
  unfolding $L$-transform-def by (simp add: Formula.Not-def)
lemma L-transform-Pred [simp]: $ L\text{-transform} (\text{Pred } f \varphi) = \text{Formula.Act} (\text{Eff } f)$ 
(Formula.Pred $\varphi$)

unfolding $L\text{-transform-def}$ by (simp add: Formula.Act-def Formula.Pred-def Formula.hereditarily-fs.Pred,$\alpha$)

lemma L-transform-Act [simp]: $ L\text{-transform} (\text{FL-Formula.Act } f \alpha x) = \text{Formula.Act} (\text{Eff } f) (\text{Formula.Act } (\text{Act } \alpha) (L\text{-transform } x))$

unfolding $L\text{-transform-def}$ by (simp add: Formula.Act-def Formula.hereditarily-fs.Act,$\alpha$)

context effect-nominal-ts

begin

interpretation $L\text{-transform}$: nominal-ts ($\vdash L$) ($\rightarrow L$)
by unfold-locales (fact L-satisfies-eqvt, fact L-transition-eqvt)

The $L\text{-transform}$ preserves satisfaction of formulas in the following sense:

theorem FL-valid-iff-valid-L-transform:
assumes $(x::(\'idx, \'pred, \'act, \'effect) \text{ formula}) \in A[F]$
shows $FL\text{-valid } P x \iff L\text{-transform}.\text{valid} (\text{EF } (F, P)) (L\text{-transform } x)$
using assms proof (induct x arbitrary: $P$)

next case (Conj xset $F$

then show $?case$
by auto (metis imageE map-bset.rep-eq, simp add: map-bset.rep-eq)

next case (Not $F$ $x$

then show $?case$ by simp

next case (Pred $f$ $F$ $\varphi$

let $?\varphi = \text{Formula.Pred } \varphi :: (\'idx, \'pred, (\'act,\'effect) \text{ L-action}) \text{ Formula.formula}$
show $?case$

proof

assume $FL\text{-valid } P$ (Pred $f$ $\varphi$

then have $L\text{-transform}.\text{valid} (\text{AC } (f, F, (f)P)) \ ?\varphi$
by (simp add: L-transform.valid-Act)

moreover from $f \in_{fs} F$ have $EF$ (F, P) $\rightarrow L$ (Eff f, AC (f, F, (f)P))
by (metis L-transition.simps(2))

ultimately show $L\text{-transform}.\text{valid} (\text{EF } (F, P)) (L\text{-transform } (\text{Pred } f \varphi))$
using L-transform.valid-Act by fastforce

next

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

then obtain $P'$ where trans: $EF$ (F, P) $\rightarrow L$ (Eff f, P') and valid:
$L\text{-transform}.\text{valid } P'$ $\ ?\varphi$
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, (f)P)
by (simp add: residual-empty-bn-eq-iff)

with valid show $FL\text{-valid } P$ (Pred $f$ $\varphi$
by simp

qed
next
  case (Act f F α x)
  show ?case
  proof
    assume FL-valid P (FL-Formula.Act f α x)
    then obtain α' α' P' where eq: FL-Formula.Act f α x = FL-Formula.Act f α' α' x' and trans: (f)P Ñ (α' P') and valid: FL-valid P' x' and fresh: bn α' zs (F, f)
    by (metis FL-valid-Act-strong finite-supp)
    from eq obtain p where p-x: p ∗ x = x' and p-α: p ∗ α = α' and supp-p: supp p ⊆ bn α ∪ bn α'
    by (metis bn-eq FL-Formula.Act-eq-iff-perm-renaming)
    from (bn α zs (F, f)) and fresh have supp (F, f) zs p
    using supp-p by (auto simp add: fresh-star-Pair fresh-star-def supp-Pair
fresh-def)
    then have p ∗ F = F and p ∗ f = f
    using supp-perm-eq by fastforce

from valid have FL-valid (−p ∗ P') x
  using p-x by (metis FL-valid-eq permute-minus-cancel(2))
then have L-transform valid (EF (L (α, F, f), −p ∗ P')) (L-transform x)
  using Act.hyps(4) by metis
then have L-transform valid (p ∗ EF (L (α, F, f), −p ∗ P')) (p ∗ L-transform x)
  by (fact L-transform.valid-eq)
then have L-transform valid (EF (L (α', F, f), P')) (L-transform x')
  using p-x and p-α and (p ∗ F = F) and (p ∗ f = f) by simp
then have L-transform valid (AC (F, f, ⟨f⟩P)) (Formula.Act (Act α') (L-transform x'))
  using trans fresh L-transform.valid-Act by fastforce
with ⟨f ∈ f., 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 ∗ bn α) zs (F, f, P) and 2: supp (F, FL-Formula.Act f α x) zs 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 z* (F, f); show bn α z* (F, FL-Formula.Act f α x)
  by (simp add: fresh-star-Pair fresh-star-def fresh-def supp-Pair)
qed

from 2 have supp F z* p and Act-fresh: supp (FL-Formula.Act f α x) z* p
  by (simp add: fresh-star-Pair fresh-star-def supp-Pair)+
from (supp F z* p) have p · F = F
  by (metis supp-perm-eq)
from Act-fresh have p · f = f
  using fresh-star-Un supp-perm-eq by fastforce
from Act-fresh have eq: FL-Formula.Act f α x = FL-Formula.Act f (p · α)
  (p · x)

with * obtain P' where trans: EF (F, P) ⊆L (Eff f,P') and valid:
  L-transform.valid P' (Formula.Act (Act (p · α)) (L-transform (p · x)))
  using L-transform-Act by (metis L-transform.valid-Act-fresh bn-L-action.simps(2)
  empty-iff fresh-star-def)
from trans have P'. P' = AC (f, F, ⟨f⟩P)
  by (simp add: residual-empty-bn-eqvt)
have supp-f-P: supp (⟨f⟩P) ⊆ supp F ∪ supp P
  using effect-apply-eqvt supp-fun-app supp-fun-app-eqvt by fastforce
with 1 have bn (Act (p · α)) z* AC (f, F, ⟨f⟩P)
  by (auto simp add: bn-eqvt fresh-star-def fresh-def supp-Pair)
with valid obtain P'' where trans': AC (f, F, ⟨f⟩P) ⊆L (Act (p · α),P'')
  and valid': L-transform.valid P'' (L-transform (p · x))
  using P' by (metis L-transform.valid-Act-fresh)

from supp-f-P and 1 have bn (p · α) z* (F, f, ⟨f⟩P)
  by (auto simp add: bn-eqvt fresh-star-def fresh-def supp-Pair)
with trans' obtain P' where P'': P'' = EF (L (p · α), F, P') and trans'':
  ⟨f⟩P → ⟨p · α⟩P'
  by (metis L-transition-AC-fresh)
from valid' have L-transform.valid (−p · P'') (L-transform x)
  by (metis (mono-tags) L-transform.valid-eqvt L-transform-eqvt permute-minus-cancel(2))
with P'' (p · F = F) · p · f = f; have L-transform.valid (EF (L (α, F, f),
  − p · P')) (L-transform x)
  by simp (metis permute-minus-self permute-minus-cancel(1))
then have FL-valid P' (p · 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
19.6 Bisimilarity in the $L$-transform

context effect-nominal-ts

begin

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

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

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

inductive $L$-bisimilar :: ('state,'effect) $L$-state $\Rightarrow$ ('state,'effect) $L$-state $\Rightarrow$ bool
where
\[ P \sim_L [F] Q \Rightarrow L$-bisimilar ($FL$ (F,P)) ($EF$ (F,Q)) \]
\[ | P \sim_L [F] Q \Rightarrow \exists F \in f, F \Rightarrow L$-bisimilar ($AC$ (f, F, (f) P)) ($AC$ (f, F, (f) Q)) \]

lemma $L$-bisimilar-is-$L$-transform-bisimulation: $L$-transform.is-bisimulation $L$-bisimilar

unfolding $L$-transform.is-bisimulation-def
proof

show symp $L$-bisimilar
by (metis FL-bisimilar-symp L-bisimilar)

next

have $\forall P_L Q_L. L$-bisimilar $P_L Q_L \Rightarrow (\forall \varphi. P_L \vdash_L \varphi \rightarrow Q_L \vdash_L \varphi)$ (is $?S$)
using FL-bisimilar-is-L-bisimulation $L$-bisimilar

by auto

moreover have $\forall P_L Q_L. L$-bisimilar $P_L Q_L \Rightarrow (\forall \alpha_L P_L', bn \alpha_L \not\in* Q_L)
\rightarrow P_L \rightarrow_L (\alpha_L, P_L') \rightarrow (\exists Q_L'. Q_L \rightarrow_L (\alpha_L, Q_L') \land L$-bisimilar $P_L' Q_L')$ (is $?T$)

proof (clarify)
fix $P_L Q_L \alpha_L P_L'$
assume $L$-bisimilar $P_L Q_L$ and fresh$_L$: $bn \alpha_L \not\in* Q_L$ and trans$_L$:

$P_L \rightarrow_L (\alpha_L, P_L')$

obtain $Q_L'$ where $Q_L \rightarrow_L (\alpha_L, Q_L')$ and $L$-bisimilar $P_L' Q_L'$
using $L$-bisimilar-proof (rule $L$-bisimilar.cases)
fix $P F Q$
assume $P_L: P_L = EF (F, P)$ and $Q_L: Q_L = EF (F, Q)$ and bisim:

$P \sim_{[F]} Q$
from $P_L$ and trans$_L$ obtain $f$ where effect: $f \in f$, $F$ and $\alpha_L P_L'$:
$\langle \alpha_L, P_L' \rangle = \{ Eff f, AC (f, F, (f) P) \}$
using $L$-transition.simps(2) by blast
from $Q_L$ and effect have $Q_L \rightarrow_L (Eff f, AC (f, F, (f) Q))$
using $L$-transition.simps(2) by blast
moreover from bisim and effect have $L$-bisimilar ($AC$ (f, F, (f) P))

$\langle AC (f, F, (f) Q) \rangle$
using $L$-bisimilar.intros(2) by blast
moreover from $\alpha_L P_L'$ have $\alpha_L = Eff f$ and $P_L' = AC (f, F, (f) P)$
by (metis bn-L-action.simps(2) residual-empty-bn-eq-iff) +
ultimately show thesis
using $\langle Q_L' \rangle. Q_L \rightarrow_L (\alpha_L, Q_L') \Rightarrow L$-bisimilar $P_L' Q_L' \Rightarrow thesis$
by *blast*

next

fix \( P F Q f \)

assume \( P_L: P_L = AC (f, F, (f)P) \) and \( Q_L: Q_L = AC (f, F, (f)Q) \)

and *bisim: \( P \sim [F] \) \( Q \) and *effect: \( f \in f_s, F \)

have finite \((supp ((f)Q, F, f))\)

by (fact finite-supp)

with \( P_L \) and \( trans_L \) obtain \( \alpha P' \) where \( trans-P: (f)P \to (\alpha, P') \) and \( \alpha_L P_L': (\alpha_L, P_L') = \langle \text{Act } \alpha, EF (L (\alpha, F, f), P') \rangle \) and *fresh: \( \text{bn } \alpha \sharp^* (\langle f \rangle Q, F, f) \)

by (metis \( L \)-transition-\( AC \)-strong)

from *bisim* and *effect* and *fresh* and \( trans-P \) obtain \( Q' \) where \( trans-Q: \langle f \rangle Q \to \langle \alpha, Q' \rangle \) and *bisim': \( P' \sim [L (\alpha, F, f)] \) \( Q' \)

by (metis FL-bisimilar-simulation-step)

from *fresh* have \( \text{bn } \alpha \sharp^* (F, f) \)

by (meson fresh-PairD(2) fresh-star-def)

with \( Q_L \) and \( trans-Q \) have \( trans-Q_L: Q_L \to L (\langle Act \alpha, EF (L (\alpha, F, f), Q) \rangle) \)

by (metis \( L \)-transition-simps(1))

from \( \alpha_L P_L' \) obtain \( p \) where \( p: (\alpha_L, P_L') = p \cdot (\text{Act } \alpha, EF (L (\alpha, F, f), P')) \)

and \( supp-p: supp p \subseteq \text{bn } \alpha \cup \text{bn } \alpha_L \)

by (metis \( (\alpha, F, f) \) residual-eq-iff-perm-renaming)

from \( supp-p \) and \( \text{fresh} \) and \( fresh_L \) and \( Q_L \) have \( supp p \sharp^* (\langle f \rangle Q, F, f) \)

unfolding fresh-star-def by (metis \( (\alpha, F, f) \) Un-iff)

fresh-Pair fresh-def \( subsetCE \) \( supp-AC \)

then have \( p-fQ: p \cdot (\langle f \rangle Q) = (\langle f \rangle Q) \) and \( p-Ff: p \cdot (F, f) = (F, f) \)

by (simp add: fresh-star-def perm-supp-eq)

from \( p \) and \( p-Ff \) have \( \alpha_L = \text{Act } (p \cdot \alpha) \) and \( P_L' = EF (L (p \cdot \alpha, F, f), p \cdot P') \)

by auto

moreover from \( Q_L \) and \( p-fQ \) and \( p-Ff \) have \( p \cdot Q_L = Q_L \)

by simp

with \( trans-Q_L \) have \( Q_L \to L (\langle p \cdot \alpha, EF (L (p \cdot \alpha, F, F), p \cdot Q) \rangle) \)

by (metis \( L \)-transform.transition-eqvt)

then have \( Q_L \to L (\langle p \cdot \alpha, EF (L (p \cdot \alpha, F, f), p \cdot Q) \rangle) \)

using \( p-Ff \) by simp

moreover from \( p-Ff \) have \( p \cdot F = F \) and \( p \cdot f = f \)

by simp+ with \( bisim' \) have \( (p \cdot P') \sim [L (p \cdot \alpha, F, f)] (p \cdot Q') \)

by (metis FL-bisimilar-eqvt L-eqvt')

then have \( L \)-bisimilar \( (EF (L (p \cdot \alpha, F, f), p \cdot P')) (EF (L (p \cdot \alpha, F, f), p \cdot Q')) \)

by (metis \( L \)-bisimilar.intros(1))

ultimately show *thesis*

using \( \forall Q_L'. Q_L \to L (\alpha_L, Q_L') \Rightarrow L \)-bisimilar \( P_L' \) \( Q_L' \Rightarrow \) *thesis*

by *blast*
simp: \( L \)-bisim where \( L \)-bisimilar \( P \, L' \) and \( L \)-bisimulation: \( AC \) \( (f, (f)P) \parallel L \) \( AC \) \( (f, (f)Q) \) by \( \text{metis} \, L\text{-act} \, \text{act} \, \text{def} \) then have \( P \, Q \parallel L \) \( AC \) \( (f, (f)P) \parallel L \) \( AC \) \( (f, (f)Q) \) by \( \text{metis} \, L\text{-act} \, \text{act} \, \text{def} \) then show \( \forall \alpha \, (\alpha, (f)Q) \) by \( \text{metis} \, L\text{-act} \, \text{act} \, \text{def} \)

    qed

moreover have \( \forall P \, Q \, invL-FL-bisimilar F \, P \, Q \parallel L \) \( AC \) \( (f, (f)P) \parallel L \) \( AC \) \( (f, (f)Q) \) by \( \text{metis} \, L\text{-act} \, \text{act} \, \text{def} \) then show \( \forall \alpha \, (\alpha, (f)Q) \) by \( \text{metis} \, L\text{-act} \, \text{act} \, \text{def} \)

    qed

ultimately show \( ?S \land ?T \) by \( \text{metis} \)

qed
proof (clarify)

fix P Q f α P′

assume bisim: invL-FL-bisimilar F P Q and effect: f ∈ f∗ F and fresh: bn α ∗∗ ((f)Q, F, f) and trans: (f)P → ⟨α,P′⟩

from bisim have EF (F,P) ∼L EF (F,Q)
  by (metis invL-FL-bisimilar-def)

moreover have bn (Eff f) ∗∗ EF (F,Q)
  by (simp add: fresh-star-def)

moreover from effect have EF (F,P) −→L ⟨Eff f, AC (f, F, ⟨f⟩P)⟩
  by (metis L-transition.simps(2))

ultimately obtain QL′ where transL: EF (F,Q) −→L ⟨Eff f, QL′⟩ and L-bisim: AC (f, F, ⟨f⟩P) ∼L QL′
  by (metis L-transform.bisimilar-simulation-step)

from transL obtain f′ where (Eff f :: ('act,'effect) L-action, QL′) = ⟨Eff f′, AC (f′, F, ⟨f⟩Q)⟩
  by (metis L-action.inject(2) bn-L-action.simps(2) residual-empty-bn-eq-iff)

from L-bisim and QL′ have AC (f, F, ⟨f⟩P) ∼L AC (f, F, ⟨f⟩Q)
  by metis

moreover from fresh have bn (Act α) ∗∗ AC (f, F, ⟨f⟩Q)
  by (simp add: fresh-def fresh-star-def supp-Pair)

moreover from fresh have bn α ∗∗ (F, f)
  by (simp add: fresh-star-Pair)

with trans have AC (f, F, ⟨f⟩P) −→L ⟨Act α, EF (L ⟨α,F,f⟩, P′)⟩
  by (metis L-transition.simps(1))

ultimately obtain QL′′ where transL′: AC (f, F, ⟨f⟩Q) −→L ⟨Act α, QL′′⟩ and L-bisim′: EF (L ⟨α,F,f⟩, P′) ∼L QL′′
  by (metis L-transform.bisimilar-simulation-step)

have finite (supp ((f)Q, F, f))
  by (fact finite-su)

with transL′ obtain α′ Q′ where trans′: (f)Q → ⟨α′,Q′⟩ and alpha: ⟨Act α :: ('act,'effect) L-action, QL′⟩ = ⟨Act α′, EF (L ⟨α,F,f⟩, Q′)⟩ and fresh′: bn α′ ∗∗ ((f)Q, F,f)
  by (metis L-transition-AC-strong)

from alpha obtain p where p: ⟨Act α :: ('act,'effect) L-action, QL′⟩ = p ∗ (Act α′, EF (L ⟨α,F,f⟩, Q′)) and supp-p: supp p ⊆ bn α ∪ 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 ∗∗ ((f)Q, F,f)

unfolding fresh-star-def by (metis no-types, hide-lams Un-iff subsetCE)
then have p-fQ: p · ⟨f⟩Q = ⟨f⟩Q and p-F: p · F = F and p-f: p · f = f
  by (simp add: fresh-star-def perm-supp-eq)+

from p and p-F and p-f have p-α′: p · α′ = α and QL′′: QL′′ = EF (L (p · α′, F, f), p · Q′)
  by auto

137
from \(\text{trans}'\) and \(p\cdot Q\) and \(p\cdot \alpha'\) have \(\langle f \rangle Q \rightarrow \langle \alpha, p \cdot Q' \rangle\) by (metis transition-eqv')
moreover from \(L\text{-bisim} '\) and \(Q_L''\) and \(p\cdot \alpha'\) have \(\text{invL-FL-bisimilar} (L \langle \alpha,F,J \rangle) P' \langle p \cdot Q' \rangle\)
ultimately show \(\exists Q'. \langle f \rangle Q \rightarrow \langle \alpha, Q' \rangle \land \text{invL-FL-bisimilar} (L \langle \alpha,F,J \rangle) P' \langle p \cdot Q' \rangle\)
by metis
qed

ultimately show \(?R \land ?S \land ?T\)
by metis
qed

theorem \(P \sim \cdot [F] Q \longleftrightarrow EF (F,P) \sim_L EF(F,Q)\)
proof
assume \(P \sim [F] Q\)
then have \(L\text{-bisimilar} (EF (F,P)) (EF (F,Q))\)
by (metis L-bisimilar.intrros(1))
then show \(EF (F,P) \sim_L EF(F,Q)\)
by (metis L-bisimilar-is-L-transform-bisimulation L-transform.bisimilar-def)
next
assume \(EF (F, P) \sim_L EF (F, Q)\)
then have \(\text{invL-FL-bisimilar} F P Q\)
by (metis invL-FL-bisimilar-def)
then show \(P \sim [F] Q\)
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 \(\text{indexed-nominal-ts TYPE} (?'idx) \?\text{satisfies} \?\text{transition};\?\text{nominal-ts.bisimilar} \?\text{satisfies} \?\text{transition} \?P \?Q\) \(\implies\) \(\text{indexed-nominal-ts.logically-equivalent TYPE} (?'idx) \?\text{satisfies} \?\text{transition} \?P \?Q\) does not actually make use of the cardinality assumptions provided by indexed nominal transition systems.

locale \(L\text{-transform-indexed-effect-nominal-ts} =\) \(\text{indexed-effect-nominal-ts} L\) satisfies transition effect-apply
for \(L:: ('act::bn) \times ('effect::fs) \text{fs-set} \times 'effect \Rightarrow 'effect \text{fs-set}\)
and satisfies :: 'state::fs \Rightarrow 'pred::fs \Rightarrow bool (infix \iff 70)\)
and transition :: 'state \Rightarrow ('act,'state) \text{residual} \Rightarrow bool (infix \rightarrow 70)\)
and effect-apply :: 'effect ⇒ 'state ⇒ 'state ((\cdot)· [0, 101] 100) + 
asumes card-idx-L-transform-state: [UNIV::('state, 'effect) L-state set] <o |UNIV::'idx set|
begin

interpretation L-transform: indexed-nominal-ts (\tau 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 ∼· L 100)

theorem EF (F,P) ∼· L EF(F,Q) → P ∼·[F] Q
proof
  assume EF (F, P) ∼· 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

20 Nominal Transition Systems and Bisimulations with Unobservable Transitions

20.1 Nominal transition systems with unobservable transitions

locale weak-nominal-ts = nominal-ts satisfies transition
  for satisfies :: 'state::fs ⇒ '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)
\textbf{inductive} \textit{tau-transition} :: 'state ⇒ 'state ⇒ bool (infix ⇒ 70) where
\textit{tau-refl} [simp]: \( P ⇒ P \)
\mid \textit{tau-step}: [ \( P \rightarrow \langle \tau, P' \rangle \); \( P' \Rightarrow P'' \) ] ⇒ \( P \Rightarrow P'' \)

\textbf{definition} \textit{observable-transition} :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/ ⇒{-}\ / - [70, 70, 71]) where
\( P \Rightarrow \{\alpha\} P' \equiv \exists Q Q'. P ⇒ Q \land Q \rightarrow \langle \alpha, Q' \rangle \land Q' ⇒ P' \)

\textbf{definition} \textit{weak-transition} :: 'state ⇒ 'act ⇒ 'state ⇒ bool (-/⇒⟨ - ⟩ / - [70, 70, 71]) where
\( P ⇒ ⟨ \alpha ⟩ P' \equiv \text{if } \alpha = \tau \text{ then } P ⇒ P' \text{ else } P ⇒ \{\alpha\} P' \)

The transition relations defined above are equivariant.

\textbf{lemma} \textit{tau-transition-eqvt} :
\textbf{assumes}\( P ⇒ P' \)\textbf{shows}\( p \cdot P ⇒ p \cdot P' \)
\textbf{using}\ assms\ \textbf{proof}\ (\textit{induction})
\textbf{case} (\textit{tau-refl} \( P \) )\textbf{show} ?case
\textbf{by}(\textit{fact} \textit{tau-transition}.\textit{tau-refl})
\textbf{next}
\textbf{case}(\textit{tau-step} \( P P' P'' \))
\textbf{from} (\( P \rightarrow \langle \tau, P' \rangle \) )\textbf{have}\( p \cdot P ⇒ \langle \tau, p \cdot P' \rangle \)
\textbf{using} \textit{tau-eqvt} \textit{transition-eqvt}\ by \textit{fastforce}
\textbf{with} (\( p \cdot P' ⇒ p \cdot P'' \))\textbf{show} ?case
\textbf{using} \textit{tau-transition}.\textit{tau-step}\ by \textit{blast}
\textbf{qed}

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

\textbf{lemma} \textit{weak-transition-eqvt} :
\textbf{assumes}\( P ⇒ \{\alpha\} P' \)\textbf{shows}\( p \cdot P ⇒ \{p \cdot \alpha\} p \cdot P' \)
\textbf{using}\ assms\ \textbf{unfolding}\ \textit{weak-transition-def} \textbf{by}(\textit{metis} (\textit{full-types}) \textit{observable-transition-eqvt} \textit{permute-minus-cancel}(2) \textit{tau-eqvt} \textit{tau-transition-eqvt})

Additional lemmas about \( ⇒ \), \textit{observable-transition} and \textit{weak-transition}.

\textbf{lemma} \textit{tau-transition-trans}:
\textbf{assumes}\( P ⇒ Q \)\textbf{and}\( Q ⇒ R \)
\textbf{shows}\( P ⇒ R \)
\textbf{using}\ assms\ \textbf{by}(\textit{induction, auto simp add: tau-step})

\textbf{lemma} \textit{observable-transitionI}:
\textbf{assumes}\( P ⇒ Q \)\textbf{and}\( Q ⇒ \{\alpha, Q'\} \)\textbf{and}\( Q' ⇒ P' \)
\textbf{shows}\( P ⇒ \{\alpha\} P' \)
\textbf{using}\ assms\ \textbf{observable-transition-def} \textbf{by}\ \textit{blast}

\textbf{lemma} \textit{observable-transition-stepI} [simp]:
\textbf{assumes}\( P ⇒ \langle \alpha, P' \rangle \)
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]:
\( P \Rightarrow \langle \tau \rangle P' \leftrightarrow P \Rightarrow P' \)
by (simp add: weak-transition-def)

lemma weak-transition-not-tau-iff [simp]:
 assumes \( \alpha \neq \tau \)
 shows \( P \Rightarrow \langle \alpha \rangle P' \leftrightarrow P \Rightarrow \{ \alpha \} P' \)
using assms by (simp add: weak-transition-def)

lemma weak-transition-stepI [simp]:
 assumes \( P \Rightarrow \{ \alpha \} P' \)
 shows \( P \Rightarrow \langle \alpha \rangle P' \)
using assms by (cases \( \alpha = \tau \), simp-all add: observable-tau-transition)

lemma weak-transition-weakI:
 assumes \( P \Rightarrow Q \) and \( Q \Rightarrow \langle \alpha \rangle Q' \) and \( Q' \Rightarrow P' \)
 shows \( P \Rightarrow \langle \alpha \rangle P' \)
 proof (cases \( \alpha = \tau \))
   case True with assms show \(?thesis \) by (metis tau-transition-trans weak-transition-tau-iff)
   next
   case False with assms show \(?thesis \) using observable-transition-def tau-transition-trans weak-transition-not-tau-iff
   by blast
 qed

end

20.2 Weak bisimulations

class weak-nominal-ts
begin

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

end
symp \ R \wedge

— weak static implication

\((\forall \ P \ Q \varphi. \ R \ P \ Q \wedge \ P \vdash \varphi \rightarrow (\exists Q'. \ Q \Rightarrow Q' \wedge R \ P \ Q' \wedge Q' \vdash \varphi)) \wedge

— weak simulation

\((\forall \ P \ Q. \ R \ P \ Q 
\rightarrow (\forall \alpha \ P'. \ bn \ \alpha \sharp* \ Q 
\rightarrow P 
\rightarrow (\alpha, P') 
\rightarrow (\exists Q'. \ Q \Rightarrow (\alpha) \ Q' \wedge R \ P' \ Q'))))

definition weakly-bisimilar ::  'state ⇒ 'state ⇒ bool (infix ≈ 100) where

\(P \approx Q \equiv \exists R. \ is-weak-bisimulation \ R \wedge R \ P \ Q\)

\((\approx)\) is an equivariant equivalence relation.

lemma is-weak-bisimulation-eqvt :

assumes is-weak-bisimulation \(R\) shows is-weak-bisimulation \((p \cdot R)\)

using assms unfolding is-weak-bisimulation-def

proof (clarify)

assumep imagine \(t :: \ symp \ R\)

assume 1: \(\ symp \ R\)

assume 2: \(\forall \ P \ Q \varphi. \ R \ P \ Q \wedge \ P \vdash \varphi \rightarrow (\exists Q'. \ Q \Rightarrow Q' \wedge R \ P \ Q' \wedge Q' \vdash \varphi)\)

assume 3: \(\forall \ P \ Q. \ R \ P \ Q 
\rightarrow (\forall \alpha \ P'. \ bn \ \alpha \sharp* \ Q 
\rightarrow P 
\rightarrow (\alpha, P') 
\rightarrow (\exists Q'. \ Q \Rightarrow (\alpha) \ Q' \wedge R \ P' \ Q'))\)

have \(\ symp \ (p \cdot R)\) (is \(\approx S\))

using 1 by (simp add: symp-eqvt)

moreover have \(\forall \ P \ Q \varphi. (p \cdot R) \ P \ Q \wedge \ P \vdash \varphi \rightarrow (\exists Q'. \ Q \Rightarrow Q' \wedge (p \cdot R) \ P \ Q' \wedge Q' \vdash \varphi)\)

(is \(\approx T\))

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 \((p \cdot R) \ ((p \cdot Q')\)

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 \(\forall \ P \ Q. (p \cdot R) \ P \ Q 
\rightarrow (\forall \alpha \ P'. \ bn \ \alpha \sharp* \ Q 
\rightarrow P 
\rightarrow (\alpha, P') 
\rightarrow (\exists Q'. \ Q \Rightarrow (\alpha) \ Q' \wedge (p \cdot R) \ P' \ Q'))\) (is \(\approx U\))

proof (clarify)

fix \(P \ Q \alpha \ P'\)

assume \(*::(p \cdot R) \ P \ Q\) and \(**:: bn \ \alpha \sharp* \ Q\) and \(***:: P \rightarrow (\alpha, P')\)

from * have \(R \ ((-p \cdot P) \ ((-p \cdot Q)\)

by (simp add: eqvt-lambda permute-bool-def unpermute-def)
moreover have \( bn (\neg p \cdot \alpha) \not\approx (\neg p \cdot Q) \)
using ** by (metis bn-eqvt fresh-star-permute-iff)
moreover have \( \neg p \cdot P \rightarrow (\neg p \cdot \alpha, \neg p \cdot P') \)
using *** by (metis transition-eqvt)
ultimately obtain \( Q' \) where \( T: \neg p \cdot Q \Rightarrow (\neg p \cdot \alpha) \)
and \( R: R (\neg p \cdot P') Q' \)
using 3 by metis
from \( T \) have \( Q \Rightarrow (\alpha) \)
by (metis permute-minus-cancel1 weak-transition-eqvt)
moreover from \( R \) have \( (p \cdot R) (p \cdot Q') \)
by (metis eqvt-apply eqvt-lambda permute-boolI unpermute_def)
ultimately show \( \exists Q'. Q \Rightarrow (\alpha) \)
and \( (p \cdot R) P' Q' \)
by metis
qed
ultimately show \( ?S \land ?T \land ?U \) by simp
qed

lemma weakly-bisimilar-eqvt :
assumes \( P \approx Q \) shows \( (p \cdot P) \approx (p \cdot Q) \)
proof –
from assms obtain \( R \) where *: \( \text{is-weak-bisimulation } R \land R P Q \)
unfolding weakly-bisimilar-def ..
then have \( \text{is-weak-bisimulation } (p \cdot R) \)
by (simp add: weakly-bisimulation-eqvt)
moreover from * have \( (p \cdot R) (p \cdot P) (p \cdot Q) \)
by (metis eqvt-apply permute-boolI)
ultimately show \( (p \cdot P) \approx (p \cdot Q) \)
unfolding weakly-bisimilar-def by auto
qed

lemma weakly-bisimilar-reflp : reflp weakly-bisimilar
proof (rule reflpI)
fix \( x \)
have \( \text{is-weak-bisimulation } (=) \)
unfolding weakly-bisimulation-def by (simp add: symp-def)
then show \( x \approx x \)
unfolding weakly-bisimilar-def by auto
qed

lemma weakly-bisimilar-symp : symp weakly-bisimilar
proof (rule sympI)
fix \( P Q \)
assume \( P \approx Q \)
then obtain \( R \) where *: \( \text{is-weak-bisimulation } R \land R P Q \)
unfolding weakly-bisimilar-def ..
then have \( R Q P \)
unfolding weakly-bisimulation-def by (simp add: symp-def)
with * show \( Q \approx P \)
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 (∀ P Q. P ≈· Q ∧ P ⊢ ϕ → (∃ Q’. Q ⇒ Q’ ∧ P ≈· Q’ ∧ 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-weak-bisimulation-def weakly-bisimilar-def)
blast+
qed

lemma weakly-bisimilar-tau-simulation-step:
  assumes P ≈· Q and P ⇒⟩ P’ obtains Q’ where Q ⇒⟩ Q’ and P’ ≈· Q’
using ⟨P ⇒⟩ P’ ; P ≈· 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 ⇒⟩ ⟨τ,P’’⟩ and ⟨P ≈· Q⟩ obtain Q’’ where Q ⇒⟩ Q’’ and P’’ ≈· 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-tau-simulation-step:
  assumes P ≈· Q and bn α ♯∗ Q and P ⇒⟨α⟩ P’
  obtains Q’ where Q ⇒⟨α⟩ Q’ and P’ ≈· Q’
proof (cases α = τ)
  case True with ⟨P ⇒⟩ Q and ⟨P ⇒⟩⟨α⟩ P’ and that show ?thesis
    using weak-transition-tau-iff weakly-bisimilar-tau-simulation-step by force
next
  case False with ⟨P ⇒⟩⟨α⟩ P’ and have P ⇒⟨α⟩ P’
  by simp
  then obtain P1 P2 where tauP: P ⇒ P1 and trans: P1 → ⟨α,P2⟩ and
tauP2: P2 ⇒ P’
  using observable-transition-def by blast
  from ⟨P ⇒⟩ Q and tauP obtain Q1 where tauQ: Q ⇒ Q1 and P1Q1: P1 ≈· Q1
  by (metis weakly-bisimilar-tau-simulation-step)
  — rename ⟨α,P2⟩ to avoid Q1, without touching Q
  obtain p where 1: (p • bn α) ♯∗ Q1 and 2: supp ⟨⟨α,P2⟩, Q⟩ ♯∗ p
  proof (rule set-avoiding2[of bn α Q1 ⟨⟨α,P2⟩, Q⟩, THEN exE])
    show finite (bn α) by (fact bn-finite)
  next
    show finite (supp Q1) by (fact finite-supp)
next
  show finite (supp ((α, P2), Q)) by (simp add: finite-supp supp-Pair)
next
  show bn α ⋆∗ ((α, P2), Q) using (bn α ⋆∗ Q) by (simp add: fresh-star-Pair)
qed metis

from 2 have 3: supp (⟨α, P2⟩ ⋆∗ p) and 4: supp Q ⋆∗ p
  by (simp add: fresh-star-Un supp-Pair)

then obtain Q2 where trans': Q1 ⇒ (p · α) Q2 and P2Q2: (p · P2) ≈ 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 · P2 ⇒ p · 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. ?bisim P Q ∧ P ⊢ ⪯ ?⟩ Q', Q ⇒ Q' ∧ ?bisim
  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

  145
\[\begin{align*}
\text{using} \ & \text{weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by} \\
\text{force} \\
\text{from} \ & \text{RQ and } \langle R \Rightarrow R' \rangle \text{ obtain } Q' \text{ where } Q \Rightarrow Q' \text{ and } R' \approx Q' \\
\text{by (metis weakly-bisimilar-\tau-simulation-step)} \\
\text{from} \ & \langle R' \approx Q' \rangle \text{ and } * \text{ obtain } Q'' \text{ where } Q' \Rightarrow Q'' \text{ and } R' \approx Q'' \text{ and} \\
\text{**: } Q'' \vdash \varphi \\
\text{using} \ & \text{weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def by} \\
\text{force} \\
\text{from} \ & \langle Q \Rightarrow Q' \rangle \text{ and } \langle Q' \Rightarrow Q'' \rangle \text{ have } Q \Rightarrow Q'' \\
\text{by (fact \tau-transition-trans)} \\
\text{moreover from} \ & \langle P \approx R' \rangle \text{ and } \langle R' \approx Q'' \rangle \text{ have } \text{?bisim } P Q'' \\
\text{by blast} \\
\text{ultimately show } \exists Q'. \ Q \Rightarrow Q' \wedge \text{?bisim } P Q' \wedge Q' \Rightarrow \varphi \\
\text{using ** by metis} \\
\text{qed} \\
\text{moreover have } \forall P. \ Q. \ \text{?bisim } P Q \rightarrow (\forall \alpha \ P'. \ bn \ \alpha \ \sharp \ Q \rightarrow P \rightarrow \langle \alpha, P' \rangle) \\
\rightarrow (\exists Q'. \ Q \Rightarrow \langle \alpha \rangle Q' \wedge \text{?bisim } P' Q') \\
\text{proof (clarify)} \\
\text{fix } P, Q, R, \alpha, P' \\
\text{assume PR: } P \approx R \text{ and } RQ; R \approx Q \text{ and fresh: } bn \ \alpha \ \sharp \ Q \text{ and trans: } P \\
\rightarrow \langle \alpha, P' \rangle \\
\quad \text{rename } (\alpha, P') \text{ to avoid } R, \text{ without touching } Q \\
\text{obtain } p \text{ where } 1: (p \cdot bn \ \alpha) \ \sharp \ R \text{ and } 2: \text{ supp } (\langle \alpha, P' \rangle, Q) \ \sharp \ p \\
\text{proof (rule at-set-avoiding2[of bn } \alpha \text{ R (} \langle \alpha, P' \rangle, Q), \text{ THEN exE])} \\
\text{show finite } (bn \ \alpha) \text{ by (fact bn-finite)} \\
\text{next} \\
\text{show finite } (supp R) \text{ by (fact finite-supp)} \\
\text{next} \\
\text{show finite } (supp (\langle \alpha, P' \rangle, Q)) \text{ by (simp add: finite-supp supp-Pair)} \\
\text{next} \\
\text{show } bn \ \alpha \ \sharp \ (\langle \alpha, P' \rangle, Q) \text{ by (simp add: fresh fresh-star-Pair)} \\
\text{qed metis} \\
\text{from 2 have 3: supp } (\langle \alpha, P' \rangle) \ \sharp \ p \text{ and 4: supp } Q \ \sharp \ p \\
\text{by (simp add: fresh-star-Un supp-Pair)+} \\
\text{from 3 have } (p \cdot \alpha, p \cdot P') = (\langle \alpha, P' \rangle) \\
\text{using supp-perm-eq by fastforce} \\
\text{with trans obtain } pR' \text{ where 5: } R \Rightarrow (p \cdot \alpha) \ \text{pR'} \text{ and 6: } (p \cdot P') \approx pR' \\
\quad \text{using PR 1 by (metis bn-eqvt weakly-bisimilar-is-weak-bisimulation is-weak-bisimulation-def)} \\
\text{from fresh and 4 have } bn \ (p \cdot \alpha) \ \sharp \ Q \\
\text{by (metis bn-eqvt fresh-star-permute-iff supp-perm-eq)} \\
\text{then obtain } pQ' \text{ where 7: } Q \Rightarrow (p \cdot \alpha) \ pQ' \text{ and 8: } pR' \approx pQ' \\
\text{using RQ 5 by (metis weakly-bisimilar-weak-simulation-step)} \\
\text{from 7 have } Q \Rightarrow (\alpha) (\neg p \cdot pQ') \\
\text{using 4 by (metis permute-minus-cancel(2) supp-perm-eq weak-transition-eqvt)} \\
\text{moreover from 6 and 8 have } \text{?bisim } P' (\neg p \cdot pQ') \\
\text{by (metis (no-types, hide-lams) weakly-bisimilar-eqvt permute-minus-cancel(2) relcomp, simp)} \\
\text{ultimately show } \exists Q'. \ Q \Rightarrow (\alpha) Q' \wedge \text{?bisim } P' Q' 
\end{align*}\]
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

theory Weak-Formula

imports
  Weak-Transition-System
  Disjunction

begin

21 Weak Formulas

21.1 Lemmas about α-equivalence involving τ

context weak-nominal-ts

begin

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

proof

  assume ?l

  then obtain p where p-α: p · τ = α and p-x: p · x1 = x2 and fresh: (supp x1 = bn τ) #* p

  by (metis Act-eq-iff-perm)

  from p-α have α = τ

  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
21.2 Weak action modality

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

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

for \( \text{satisfies} :: '\text{state} :: \text{fs} \Rightarrow '\text{pred} :: \text{fs} \Rightarrow \text{bool} \ (\text{infix} \Rightarrow \text{70}) \)

and \( \text{transition} :: '\text{state} \Rightarrow ('\text{act} :: \text{bn}, '\text{state}) \Rightarrow \text{residual} \Rightarrow \text{bool} \ (\text{infix} \Rightarrow \text{70}) \)

assumes \( \text{card-idx-perm} :: |\text{UNIV} :: \text{perm set}| < o |\text{UNIV} :: '\text{idx set}| \)

and \( \text{card-idx-state} :: |\text{UNIV} :: '\text{state set}| < o |\text{UNIV} :: '\text{idx set}| \)

and \( \text{card-idx-nat} :: |\text{UNIV} :: \text{nat set}| < o |\text{UNIV} :: '\text{idx set}| \)

**begin**

The assumption \(|\text{UNIV}| < o |\text{UNIV}|\) is redundant: it is already implied by \(|\text{UNIV}| < o |\text{UNIV}|\). A formal proof of this fact is left for future work.

**lemma** card-idx-nat \[\text{simp}\]:

\(|\text{UNIV} :: \text{nat set}| < o \text{natLeq} + c |\text{UNIV} :: '\text{idx set}| \)

**proof**

note card-idx-nat

also have \(|\text{UNIV} :: '\text{idx set}| \leq o \text{natLeq} + c |\text{UNIV} :: '\text{idx set}| \)

by (metis Cnotzero-UNIV ordLeq-csum2)

finally show ?thesis.

**qed**

primrec tau-steps :: ('idx, 'pred :: fs, 'act :: bn) formula \( \Rightarrow \text{nat} \Rightarrow ('idx, 'pred, 'act) \)

formula

where

\( \text{tau-steps} x 0 = x \)

| \( \text{tau-steps} x (\text{Suc} n) = \text{Act} \tau (\text{tau-steps} x n) \)

**lemma** tau-steps-eqvt \[\text{simp}\]:

\( p \cdot \text{tau-steps} x n = \text{tau-steps} (p \cdot x) (p \cdot n) \)

by (induct n) (simp-all add: permute-nat-def tau-eqvt)

**lemma** tau-steps-eqvt' \[\text{simp}\]:

\( p \cdot \text{tau-steps} x = \text{tau-steps} (p \cdot x) \)

by (simp add: permute-fun-def)

**lemma** tau-steps-eqvt-raw \[\text{simp}\]:

\( p \cdot \text{tau-steps} = \text{tau-steps} \)

by (simp add: permute-fun-def)

**lemma** tau-steps-add \[\text{simp}\]:

\( \text{tau-steps} (\text{tau-steps} x m) n = \text{tau-steps} x (m + n) \)

by (induct n) auto

**lemma** tau-steps-not-self:

\( x = \text{tau-steps} x n \longleftrightarrow n = 0 \)
proof
  assume \( x = \text{tau-steps} \times n \) then show \( n = 0 \)
proof (induct \( n \) arbitrary: \( x \))
  case \( 0 \) show ?case ..
next
  case \( \text{Suc} \ n \)
  then have \( x = \text{Act} \ \tau \ (\text{tau-steps} \times n) \)
    by simp
  then show \( \text{Suc} \ n = 0 \)
proof (induct \( x \))
    case \( \text{Act} \ \alpha \ \times \)
    then have \( x = \text{tau-steps} \ (\text{Act} \ \tau \ \times) \ n \)
      by (metis \text{Act-tau-eq-iff})
    with \text{Act.hyps} show ?thesis
      by (metis \text{add-Suc tau-steps.simps(2) tau-steps-add})
  qed simp-all
  qed
next
  assume \( n = 0 \) then show \( x = \text{tau-steps} \times n \)
    by simp
  qed

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

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

lemma \text{weak-tau-modality-eqvt} [simp]:
  \( p \cdot \text{weak-tau-modality} \times = \text{weak-tau-modality} \ (p \cdot x) \)
unfolding \text{weak-tau-modality-def} by (simp add: \text{map-bset-eqvt})

lemma \text{weak-tau-modality-eq-iff} [simp]:
  \text{weak-tau-modality} \times = \text{weak-tau-modality} \ y \iff 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 \in range (tau-steps y) and y \in 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 \alpha1 (weak-tau-modality x1) = Act \alpha2 (weak-tau-modality x2) \iff Act \alpha1 x1 = Act \alpha2 x2
  by (simp add: Act-eq-iff-perm)

definition weak-action-modality :: \'(act \Rightarrow \'(idx,\'pred::fs,\'act::bn) formula \Rightarrow \'(idx,\'pred,\'act) formula ((\cdot))-)
where
  \langle\langle \alpha \rangle\rangle x \equiv if \alpha = \tau then weak-tau-modality x else weak-tau-modality (Act \alpha (weak-tau-modality x))

lemma weak-action-modality-eqvt [simp]:
  p \cdot (\langle\langle \alpha \rangle\rangle x) = (\langle\langle p \cdot \alpha \rangle\rangle)(p \cdot x)
  using tau-eqvt weak-action-modality-def by fastforce

lemma weak-action-modality-tau:
  (\langle\langle \tau \rangle\rangle x) = 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) = weak-tau-modality (Act \alpha (weak-tau-modality x))
  using assms unfolding weak-action-modality-def by simp

Equality is modulo \alpha-equivalence.
Note that the converse of the following lemma does not hold. For instance, for \( \alpha \neq \tau \) we have \( \langle\langle \tau \rangle\rangle \text{Act } \alpha (\text{weak-tau-modality } x) = \langle\langle \alpha \rangle\rangle x \) by definition, but clearly not \( \text{Act } \tau (\text{Act } \alpha (\text{weak-tau-modality } x)) = \text{Act } \alpha x \).

**Lemma** weak-action-modality-eq:

**Assumes** \( \text{Act } \alpha_1 x_1 = \text{Act } \alpha_2 x_2 \)

**Shows** \( \langle\langle \alpha_1 \rangle\rangle x_1 = \langle\langle \alpha_2 \rangle\rangle x_2 \)

**Proof** (cases \( \alpha_1 = \tau \))

**Case** True

with assumptions have \( \alpha_2 = \alpha_1 \land x_2 = x_1 \)

by (metis Act-tau-eq-iff)

then show \?thesis by simp

next

**Case** False

from assumptions obtain \( p \) where 1: \( \text{supp } x_1 - bn \alpha_1 = \text{supp } x_2 - bn \alpha_2 \) and

2: \( \text{supp } x_1 - bn \alpha_1 ) \# p \)

and 3: \( p \cdot x_1 = x_2 \) and 4: \( \text{supp } x_1 - bn \alpha_1 = \text{supp } x_2 - bn \alpha_2 \) and 5: \( \text{supp } x_1 - bn \alpha_1 ) \# p \)

and 6: \( p \cdot \alpha_1 = \alpha_2 \)

by (metis Act-eq-iff-perm)

from 1 have \( \text{supp } (\text{weak-tau-modality } x_1) - bn \alpha_1 = \text{supp } (\text{weak-tau-modality } x_2) - bn \alpha_2 \)

by (metis supp-weak-tau-modality)

moreover from 2 have \( \text{supp } (\text{weak-tau-modality } x_1) - bn \alpha_1 ) \# p \)

by (metis supp-weak-tau-modality)

moreover from 3 have \( p \cdot \text{weak-tau-modality } x_1 = \text{weak-tau-modality } x_2 \)

by (metis weak-tau-modality-eqvt)

ultimately have \( \text{Act } \alpha_1 (\text{weak-tau-modality } x_1) = \text{Act } \alpha_2 (\text{weak-tau-modality } x_2) \)

using 4 and 5 and 6 and Act-eq-iff-perm by blast

moreover from \( \alpha_1 \neq \tau \) and assumptions have \( \alpha_2 \neq \tau \)

by (metis Act-tau-eq-iff)

ultimately show \?thesis using \( \alpha_1 \neq \tau \) by (simp add: weak-action-modality-not-tau)

qed

### 21.3 Weak formulas

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

where

| wf-Conj: finite (supp xset) ⇒ (∀x. x ∈ set-bset xset ⇒ weak-formula x) ⇒ weak-formula (Conj xset)
| wf-Not: weak-formula x ⇒ weak-formula (Not x)
| wf-Act: weak-formula x ⇒ weak-formula (⟨⟨⟨x⟩⟩)(Conj (binset (Pred ϕ) (bsingleton x)))
| wf-Pred: weak-formula x ⇒ weak-formula ⟨⟨⟨⟩⟩)(Conj (binset (Pred ϕ) (bsingleton x)))

**Lemma** finite-supp-wf-Pred [simp]: finite (supp (binset (Pred ϕ) (bsingleton x)))

**Proof**
have \( \text{supp} \ (\text{bsingleton} \ x) \subseteq \text{supp} \ x \)
by (simp add: eqvtI supp-fun-app-eqvt)
moreover have \( \text{eqvt} \ \text{binsert} \)
by (simp add: eqvtI)
ultimately have \( \text{supp} \ (\text{binsert} \ (\text{Pred} \ \phi) \ (\text{bsingleton} \ x)) \subseteq \text{supp} \ \phi \cup \text{supp} \ x \)
using supp-fun-app supp-fun-app-eqvt by fastforce
then show \( ?\text{thesis} \)
by (metis finite-UnI finite-suppp rev-finite-subset)
qed

weak-formula is equivariant.

lemma weak-formula-eqvt [simp]: \( \text{weak-formula} \ x = \Rightarrow \text{weak-formula} \ (p \cdot x) \)
proof (induct rule: weak-formula.induct)
case (wf-Conj xset) then show \( ?\text{case} \)
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 \( ?\text{case} \)
by (simp add: weak-formula.wf-Not)
next
case (wf-Act x \alpha) then show \( ?\text{case} \)
by (simp add: weak-formula.wf-Act)
next
case (wf-Pred x \phi) then show \( ?\text{case} \)
by (simp add: tau-eqvt weak-formula.wf-Pred)
qed

end

end
theory Weak-Validity
imports
Weak-Formula
Validity
begin

22 Weak Validity

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

context indexed-weak-nominal-ts
begin

lemma valid-weak-tau-modality-iff-tau-steps:
\( P \vdash \text{weak-tau-modality} \ x \leftrightarrow (\exists n. \ P \vdash \text{tau-steps} \ x \ n) \)
unfolding weak-tau-modality-def by (auto simp add: map-bset.rep-eq)

lemma tau-steps-iff-tau-transition:
  (∃n. P |= tau-steps x n) ←→ (∃P', P ⇒ P' ∧ P' |= x)
proof
  assume ∃n. P |= tau-steps x n
  then obtain n where P |= tau-steps x n
    by meson
  then show ∃P'. P ⇒ P' ∧ P' |= 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 ⇒ ⟨τ, P'⟩ and P' |= tau-steps x n
        by (auto simp add: valid-Act-fresh[OF bn-tau-fresh])
      with Suc.hyps show ?case
        using tau-step by blast
    qed
next
  assume ∃P'. P ⇒ P' ∧ P' |= x
  then obtain P' where P ⇒ P' and P' |= x
    by meson
  then show ∃n. P |= tau-steps x n
    proof (induct)
      case (tau-refl P)
      then have P |= tau-steps x 0
        by simp
      then show ?case
        by meson
    next
      case (tau-step P P' P'')
      then obtain n where P' |= tau-steps x n
        by meson
      with ⟨P ⇒ ⟨τ⟩, P'⟩ have P |= tau-steps x (Suc n)
        by (auto simp add: valid-Act-fresh[OF bn-tau-fresh])
      then show ?case
        by meson
    qed
qed

lemma valid-weak-tau-modality:
  P |= 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 α (weak-tau-modality x) = Act
    α' x' and trans': Q ⇒ ⟨α',Q'⟩ and valid': Q' |= x'
    by (metis valid-Act)
  from eq obtain p where p-α: α' = p · α and p-x: x' = p · weak-tau-modality
    x
    by (metis Act-eq-iff-perm)
  with eq have Act α x = Act α' (p · x)
    using Act-weak-tau-modality-eq-iff by simp
  moreover from valid' and p-x have Q' |= weak-tau-modality (p · x)
    by simp
  then obtain P' where trans'": Q' ⇒ P' and valid'": P' |= p · x
    by (metis valid-weak-tau-modality)
  from trans and trans' and trans'' have P ⇒⟨α⟩ 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 α x = Act α' x' and trans: P ⇒⟨α'⟩
\( P' \) and valid: \( P' \models x' \)

by blast

with \( \alpha \neq \tau \) have \( \alpha': \alpha' \neq \tau \)

using eq by (metis Act-tau-eq-iff)

with trans obtain \( Q \ Q' \) where trans': \( P \Rightarrow Q \) and trans'': \( Q \Rightarrow (\alpha', Q') \)

and trans''': \( Q' \Rightarrow P' \)

by (meson observable-transition-def weak-transition-def)

from trans'' and valid have \( Q' \models \) weak-tau-modality \( x' \)

by (metis valid-weak-tau-modality)

with trans'' have \( Q \models \) Act \( \alpha' \) (weak-tau-modality \( x' \))

by (metis valid-Act)

with trans' and \( \alpha' \) have \( P \models (\langle \alpha' \rangle) x' \)

by (metis valid-weak-tau-modality weak-action-modality-not-tau)

moreover from eq have \( (\langle \alpha \rangle) x = (\langle \alpha' \rangle) x' \)

by (metis weak-action-modality-eq)

ultimately show \(?\)!

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 \((\text{supp } X)\)

shows \( P \models (\langle \alpha \rangle) x \mapsto (\exists x' P'). \) Act \( \alpha \) \( x = \) Act \( \alpha' \) \( x' \) \& \( P \Rightarrow \langle \alpha' \rangle P' \land P' \models x' \land \text{bn } \alpha' \sharp X \)

proof

assume \( P \models (\langle \alpha \rangle) x \)

then obtain \( \alpha' \) \( x' P' \) where eq: \( \text{Act } \alpha x = \text{Act } \alpha' x' \) and trans: \( P \Rightarrow \langle \alpha' \rangle P' \)

and valid: \( P' \models x' \)

by (metis valid-weak-tau-modality)

show \( \exists \alpha' \) \( x' P' \). Act \( \alpha \) \( x = \) Act \( \alpha' \) \( x' \) \& \( P \Rightarrow \langle \alpha' \rangle P' \land P' \models x' \land \text{bn } \alpha' \sharp X \)

proof (cases \( \alpha' = \tau \))

\( \text{case } \) True

then show \(?\)thesis

using eq and trans and valid and bn-tau-fresh by blast

next

\( \text{case } \) False

with trans obtain \( Q \ Q' \) where trans': \( P \Rightarrow Q \) and trans'': \( Q \Rightarrow (\alpha', Q') \)

and trans''': \( Q' \Rightarrow P' \)

by (metis weak-transition-def observable-transition-def)

have finite \((\text{bn } \alpha')\)

by (fact bn-finite)

moreover note \((\text{finite } (\text{supp } X))\)

moreover have finite \((\text{supp } (\text{Act } \alpha' \) \( \cdot \) \( x' \), \( \langle \alpha', Q' \rangle \))\)

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

moreover have \( \text{bn } \alpha' \sharp \) \((\text{Act } \alpha' \) \( \cdot \) \( x' \), \( \langle \alpha', Q' \rangle \))

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

ultimately obtain \( p \) where fresh-X: \( (p \cdot \text{bn } \alpha') \sharp X \) and supp \((\text{Act } \alpha')
\[ x', \langle \alpha', Q' \rangle \sharp^* p \]

by (metis at-set-avoiding2)

then have \( \text{supp} (\text{Act} \alpha' x') \sharp^* p \) and \( \text{supp} \langle \alpha', Q' \rangle \sharp^* p \)

by (metis fresh-star-Un supp-Pair+)

then have \( 1: \text{Act} (p \cdot \alpha') (p \cdot x') = \text{Act} \alpha' x' \) and \( 2: \langle p \cdot \alpha', p \cdot Q' \rangle = \langle \alpha', Q' \rangle \)

by (metis Act-eqvt supp-perm-eq, metis abs-residual-pair-eqvt supp-perm-eq)

from \( \text{trans}' \) and \( \text{trans}'' \) and \( \text{trans}''' \) have \( P \Rightarrow \langle p \cdot \alpha' \rangle (p \cdot P') \)

using \( 2 \) by (metis observable-transitionI tau-transition-eqvt weak-transition-stepI)

then show \( \top \)thesis

using eq and \( 1 \) and valid and fresh-X by (metis bn-eqvt valid-eqvt)

qed

next

assume \( \exists \alpha' x'. \text{Act} \alpha x = \text{Act} \alpha' x' \land P \Rightarrow \langle \alpha' \rangle P' \land P' \models x' \land \text{bn} \ \alpha' \sharp^* \)

X

then show \( P \models \langle \langle \alpha \rangle \rangle x \)

by (metis valid-weak-action-modality)

qed

lemma valid-weak-action-modality-fresh:

assumes \( \text{bn} \ \alpha \sharp^* P \)

shows \( P \models ((\langle \alpha \rangle)x) \longleftrightarrow (\exists P'. P \Rightarrow \langle \alpha \rangle P' \land P' \models x) \)

proof

assume \( P \models \langle \langle \alpha \rangle \rangle x \)

moreover have \( \text{finite} \ (\text{supp} P) \)

by (fact finite-supp)

ultimately obtain \( \alpha' x' P' \) where

\( \text{eq}: \text{Act} \alpha x = \text{Act} \alpha' x' \land P \Rightarrow \langle \alpha' \rangle P' \land P' \models x' \land \text{bn} \ \alpha' \sharp^* \)

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

from \( \text{eq} \) obtain \( p \) where \( p\cdot\alpha: \alpha' = p \cdot \alpha \) and \( p\cdot x: x' = p \cdot x \) and \( \text{supp}-p: \text{supp} p \subseteq \text{bn} \ \alpha \cup p \cdot \text{bn} \ \alpha \)

by (metis Act-eq-iff-perm-renaming)

from \( \text{assms} \) and \( \text{fresh} \) have \( (\text{bn} \ \alpha \cup p \cdot \text{bn} \ \alpha) \sharp^* P \)

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

then have \( \text{supp} p \sharp^* P \)

using \( \text{supp}-p \) by (metis fresh-star-def subset-eq)

then have \( p\cdot P: -p \cdot P = P \)

by (metis perm-supp-eq supp-minus-perm)

from \( \text{trans} \) have \( P \Rightarrow \langle \alpha \rangle (\neg p \cdot P') \)

using \( p\cdot P \) \( p\cdot\alpha \) by (metis permute-minus-cancel(1) weak-transition-eqvt)

moreover from \( \text{valid} \) have \( \neg p \cdot P' \models x \)

using \( p\cdot x \) by (metis permute-minus-cancel(1) valid-eqvt)

ultimately show \( \exists P'. P \Rightarrow \langle \alpha \rangle P' \land P' \models x \)

by meson
next
  assume ∃P'. P ⇒⟨α⟩ P' ∧ P' |= x then show P |= ⟨⟨α⟩⟩x
  by (metis valid-weak-action-modality)
qed

end

end

theory Weak-Logical-Equivalence
imports Weak-Formula Weak-Validity
begin

23 Weak Logical Equivalence

context indexed-weak-nominal-ts
begin

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

definition weakly-logically-equivalent :: 'state ⇒ '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 weakly-logically-equivalent-eqvt:
  assumes p · P ≡· p · Q
  unfolding weakly-logically-equivalent-def proof (clarify)
  fix x :: ('idx,'pred,'act) formula
  assume weak-formula x
  then have weak-formula (¬p · x)
  by simp
  then show p · P |= x ←→ p · Q |= 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


24 Weak Bisimilarity Implies Weak Logical Equivalence

context indexed-weak-nominal-ts

begin

lemma weak-bisimilarity-implies-weak-equivalence-Act:
  assumes \( \forall P, Q. P \approx Q \implies P \models x \leftrightarrow Q \models x \)
  and \( P \approx Q \)
  — not needed: and weak-formula \( x \)
  and \( P \models \langle\langle \alpha \rangle \rangle x \)
  shows \( Q \models \langle\langle \alpha \rangle \rangle x \)
proof
  -
  have finite \((\text{supp } Q)\) by \( \text{fact finite-supp} \)
  with \( \langle P \models \langle\langle \alpha \rangle \rangle x \rangle \) obtain \( \alpha' \) \( x' \) \( P' \) where \( \text{eq} \): \( \text{Act } \alpha x = \text{Act } \alpha' x' \) and \( \text{trans} \):
  \( P \Rightarrow \langle\langle \alpha' \rangle \rangle P' \) and \( \text{valid} \): \( P' \models x' \) and \( \text{fresh} \): \( \text{bn } \alpha' \sharp \ast Q \)
  by \( \text{metis valid-weak-action-modality-strong} \)
from \( \langle P \approx Q \rangle \) and \( \text{fresh} \) and \( \text{trans} \) obtain \( Q' \) where \( \text{trans'}: Q \Rightarrow \langle\langle \alpha' \rangle \rangle Q' \)
  and \( \text{bisim'}: P' \approx Q' \)
  by \( \text{metis weakly-bisimilar-weak-simulation-step} \)
from \( \text{eq} \) obtain \( p \) where \( px: x' = p \cdot x \)
  by \( \text{metis Act-eq-iff-perm} \)
with \( \text{valid} \) have \( \neg p \cdot P' \models x \)
  by \( \text{metis permute-minus-cancel(1) valid-eqvt} \)
moreover from \( \text{bisim'} \) have \( \neg p \cdot P \approx \neg p \cdot Q' \)
  by \( \text{metis weakly-bisimilar-eqvt} \)
ultimately have \( \neg p \cdot Q' \models x \)
  using \( \langle P Q. P \approx Q \implies P \models x \leftrightarrow Q \models x \rangle \) by metis
with \( px \) have \( Q' \models x' \)
  by \( \text{metis permute-minus-cancel(1) valid-eqvt} \)
with \( \text{eq} \) and \( \text{trans'} \) 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 \( \forall P, Q. P \approx Q \implies P \models x \leftrightarrow Q \models x \)
  and \( P \approx Q \)
  — not needed: and weak-formula \( x \)
  and \( P \models \langle\langle \tau \rangle \rangle (\text{Conj } (\text{binsert } (\text{Pred } \varphi) \ (\text{bsingleton } x))) \)
  shows \( Q \models \langle\langle \tau \rangle \rangle (\text{Conj } (\text{binsert } (\text{Pred } \varphi) \ (\text{bsingleton } x))) \)
proof
  -
  let \( ?c = \text{Conj } (\text{binsert } (\text{Pred } \varphi) \ (\text{bsingleton } x)) \)
from \( \langle P \models \langle\langle \tau \rangle \rangle ?c \rangle \) obtain \( P' \) where \( \text{trans} \): \( P \Rightarrow P' \) and \( \text{valid} \): \( P' \models ?c \)
using \textit{valid-weak-action-modality} by auto

have \( \text{bn} \, \tau \, \ast \, Q \)
by (simp add: fresh-star-def)
with \( \langle P \approx Q \rangle \) and trans obtain \( Q' \) where trans': \( Q \Rightarrow Q' \) and bisim': \( P' \approx Q' \)
by (metis weakly-bisimilar-weak-simulation-step weak-transition-tau-iff)

from valid have \( *: P' \vdash \varphi \) and \( **: P' \models x \)
by (simp add: binset.rep-eq)

from bisim' and \( * \) obtain \( Q'' \) where trans'': \( Q' \Rightarrow Q'' \) and bisim'': \( P' \approx Q'' \)
by (metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation)

moreover from trans' and trans''' have \( Q \Rightarrow \langle \tau \rangle \, Q'' \)
by (metis tau-transition-trans weak-transition-tau-iff)

ultimately show \( Q \models \langle \langle \tau \rangle \rangle \, ?c \)

qed

\textbf{theorem} \textit{weak-bisimilarity-implies-weak-equivalence}: assumes \( P \approx Q \) shows \( P \equiv Q \)
\textbf{proof} –
{}\hspace{1em}
fix \( x :: \langle \text{idx}, \text{pred}, \text{act} \rangle \) formula
assume weak-formula \( x \)
then have \( \bigwedge P \, Q. \, P \approx Q \Rightarrow P \models x \leftrightarrow Q \models x \)
proof (induct rule: weak-formula.induct)
  case (uf-Conj xset) then show \( ?c \)
  by simp
next
  case (uf-Not x) then show \( ?c \)
  by simp
next
  case (uf-Act x \( \alpha \)) then show \( ?c \)
  by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Act sympE)
next
  case (uf-Pred x \( \varphi \)) then show \( ?c \)
  by (metis weakly-bisimilar-symp weak-bisimilarity-implies-weak-equivalence-Pred sympE)
qed

159
```isar
with assms show ?thesis
  unfolding weakly-logically-equivalent-def by simp
qed

end
end

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

25 Weak Logical Equivalence Implies Weak Bisimilarity

context indexed-weak-nominal-ts
begin

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

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

  lemma weakly-equivalent-iff-not-distinguished: (P ≡· Q) ←→ ¬ (∃ x. weak-formula x ∧ x distinguishes P from Q)
  by (meson is-distinguishing-formula-def weakly-logically-equivalent-def valid-Not wf-Not)

  There exists a distinguishing weak formula for P and Q whose support is contained in 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 ⊆ supp P and y distinguishes P from Q
  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
```

160
have \((a \equiv b) \cdot ?B \subseteq ?B\)

proof
  fix \(x'\)
  assume \(x' \in (a \equiv b) \cdot ?B\)
  then obtain \(p\) where 1: \(x' = (a \equiv b) \cdot p \cdot x\) and 2: \(\text{supp } P \#^* p\)
    by (auto simp add: permute-set-def)
  let \(?q = (a \equiv b) + p\)
  from 1 have \(x' = ?q \cdot x\)
    by simp
  moreover from \(a\) and \(b\) and 2 have \(\text{supp } P \#^* ?q\)
    by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))
  ultimately show \(x' \in ?B\) by blast
qed

moreover have \(?B \subseteq (a \equiv b) \cdot ?B\)

proof
  fix \(x'\)
  assume \(x' \in ?B\)
  then obtain \(p\) where 1: \(x' = p \cdot x\) and 2: \(\text{supp } P \#^* p\)
    by auto
  let \(?q = (a \equiv b) + p\)
  from 1 have \(x' = (a \equiv b) \cdot ?q \cdot x\)
    by simp
  moreover from \(a\) and \(b\) and 2 have \(\text{supp } P \#^* ?q\)
    by (metis fresh-perm fresh-star-def fresh-star-plus swap-atom-simps(3))
  ultimately show \(x' \in ?B\) by blast
qed

ultimately show \(a \equiv b\) \(\cdot ?B = ?B\) ..
qed

then have \(\text{supp-B-subset-supp-P}: \text{supp } ?B \subseteq \text{supp } P\)
  by (metis (erased, lifting) finite-supp supp-is-subset)
then have \(\text{finite-supp-B}: \text{finite } (\text{supp } ?B)\)
  using finite-supp rev-finite-subset by blast
have \(?B \subseteq (\lambda p \cdot x ) \cdot \text{UNIV}\)
  by auto
then have \(|?B| \leq o \cdot |\text{UNIV} :: \text{perm set}|\)
  by (rule surj-imp-ordLeq)
also have \(|\text{UNIV} :: \text{perm set}| < o \cdot |\text{UNIV} :: 'idx set|\)
  by (metis card-idx-perm)
also have \(|\text{UNIV} :: 'idx set| \leq o \cdot \text{natLeq} + c \cdot |\text{UNIV} :: 'idx set|\)
  by (metis Cnotzero-UNIV ordLeq-csum2)
finally have \(\text{card-B}: |?B| < o \cdot \text{natLeq} + c \cdot |\text{UNIV} :: 'idx set|\).

let \(?y = \text{Conj } (\text{Abs-bset } ?B) :: ('idx, 'pred, 'act) \text{ formula}\)
have weak-formula \(?y\)
proof
  show finite (\(\text{supp } (\text{Abs-bset } ?B :: - \text{set}['idx])\))
    using finite-supp-B card-B by simp
next
  fix \(x'\) assume \(x' \in \text{set-bset } (\text{Abs-bset } ?B :: - \text{set}['idx])\)
with \( \text{card-B} \) obtain \( p \) where \( x' = p \cdot x \)

using Abs-bset-inverse mem-Collect-eq by auto

then show weak-formula \( x' \)
  using \langle weak-formula \( x \) \rangle by (metis weak-formula-eqvt)

qed

moreover from finite-supp-B and card-B and sup-B-subset-sup-P have
  supp \( ?y \) \( \subseteq \) supp \( P \)
  by simp

moreover have \( ?y \) distinguishes \( P \) from \( Q \)
  unfolding is-distinguishing-formula-def
  proof
    from assms show \( P \models ?y \)
      by (auto simp add: card-B finite-supp-B)
  next
    from assms show \( \neg Q \models ?y \)
      by (auto simp add: card-B finite-supp-B)

  qed

ultimately show \( \text{thesis ..} \)
  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 
  { 
    fix \( P Q \) \( \varphi \) assume \( P \equiv Q \) and \( P \models \varphi \)
    then have \( \exists \; Q'. \; Q \Rightarrow Q' \wedge P \equiv Q' \land Q' \models \varphi \)
      proof 
        { 
          let \( ?Q' = \{ Q'. \; Q \Rightarrow Q' \wedge Q' \models \varphi \} \)
          assume \( \forall Q' \in ?Q'. \; \neg P \equiv Q' \)
          then have \( \forall Q' \in ?Q'. \exists x :: ( 'idx', 'pred', 'act' ) \text{ formula. weak-formula } x \land x \text{ distinguishes } P \text{ from } Q' \)
            by (metis weakly-equivalent-iff-not-distinguished)
          then have \( \forall Q' \in ?Q'. \exists x :: ( 'idx', 'pred', 'act' ) \text{ formula. weak-formula } x \land \text{ supp } x \subseteq \text{ supp } P \wedge x \text{ distinguishes } P \text{ from } Q' \)
            by (metis distinguished-bounded-support)
          then obtain \( f :: '\text{state} \Rightarrow ( 'idx', 'pred', 'act' ) \text{ formula where} \)
            \( \star: \forall Q' \in ?Q'. \text{ weak-formula } ( f \; Q' ) \land \text{ supp } ( f \; Q' ) \subseteq \text{ supp } P \wedge ( f \; Q' ) \)
            distinguishes \( P \text{ from } Q' \)
            by metis
          have supp \( ( f \; ' ?Q' ) \subseteq \text{ supp } P \)
            by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)
          then have finite-supp-image: finite \( ( \text{ supp } ( f \; ' ?Q' ) ) \)
            using finite-supp rev-finite-subset by blast
          have \( | f \; ' ?Q' | \leq | \text{UNIV} :: '\text{state} set | \)
            using card-of-UNIV card-of-image ordLeq-transitive by blast
        }
also have $|\text{UNIV} :: \text{state set}| < o |\text{UNIV} :: \text{idx set}|$
by (metis card-idx-state)
also have $|\text{UNIV} :: \text{idx set}| \leq o \text{natLeq} + c |\text{UNIV} :: \text{idx set}|$
by (metis Cnotzero-UNIV ordLeq-csum2)
finally have card-image: $|f \cdot ?Q'| < o \text{natLeq} + c$
by (metis Cnotzero-UNIV ordLeq-csum2)

let $?y = \text{Conj} (\text{Abs-bset} (f \cdot ?Q')) :: (\text{idx}, \text{pred}, \text{act})$ formula
have weak-formula $?y$
proof (standard+)
  show finite $(\text{supp} (\text{Abs-bset} (f \cdot ?Q') :: - \text{set}['\text{idx}']))$
    using finite-supp-image card-image by simp
next
fix $x$ assume $x \in \text{set-bset} (\text{Abs-bset} (f \cdot ?Q') :: - \text{set}['\text{idx}']))$
with card-image obtain $Q'$ where $Q' \in ?Q'$ and $x = f Q'$
using Abs-bset-inverse imageE set-bset set-bset-to-set-bset by auto
then show weak-formula $x$
  using * by metis
qed

let $?z = \langle\langle \tau \rangle\rangle (\text{Conj} (\text{binsert} (\text{Pred} \varphi) (\text{bsingleton} ?y)))$
have weak-formula $?z$
  by standard (fact weak-formula $?y)$
moreover have $P \models ?z$
proof
  have $P \Rightarrow \langle\tau \rangle P$
    by simp
  moreover
  { fix $Q'$
    assume $Q \Rightarrow Q' \land Q' \models \varphi$
    with * have $P \models f Q'$
      by (metis is-distinguishing-formula-def mem-Collect-eq)
  }
with $P \models \varphi$ have $P \models \text{Conj} (\text{binsert} (\text{Pred} \varphi) (\text{bsingleton} ?y))$
  by (simp add: binsert.rep-eq finite-supp-image card-image)
ultimately show $?thesis$
  using valid-weak-action-modality by blast
qed
moreover have $\neg Q \models ?z$
proof
assume $Q \models ?z$
then obtain $Q'$ where $1: Q \Rightarrow Q'$ and $Q' \models \text{Conj} (\text{binsert} (\text{Pred} \varphi) (\text{bsingleton} ?y))$
  using valid-weak-action-modality by auto
then have $2: Q' \models \varphi$ and $3: Q' \models ?y$
  by (simp add: binsert.rep-eq finite-supp-image card-image)+
from $3$ have $\land Q'' : Q \Rightarrow Q'' \land Q'' \models \varphi \Rightarrow Q' \models f Q''$
  by (simp add: finite-suppp-image card-image)
with $1$ and $2$ and * show False
using is-distinguishing-formula-def by blast
qed
ultimately have False
  by (metis ⟨P ≡· Q⟩ weakly-logically-equivalent-def)
}
then show thesis
by blast
d qed
}
morerover — 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: $Q \Rightarrow \langle \alpha \rangle Q'$

by auto

with $*$ and $a$ and $b$ have $a \notin \text{supp} (f Q')$ and $b \notin \text{supp} (f Q')$

by auto

with 1 have $x = (a \Rightarrow b) \cdot f Q'$

by (metis fresh-perm fresh-star-def supp-perm-eq swap-atom)

with 2 show $x \in (a \Rightarrow b) \cdot (f ' ?Q')$

using mem-permute-iff by blast

qed

ultimately show $(a \Rightarrow b) \cdot (f ' ?Q') = f ' ?Q'$ ..

qed

then have supp-image-subset-supp-$P'$: $\text{supp} (f ' ?Q') \subseteq \text{supp } P'$

by (metis erased, lifting finite-supp supp-is-subset)

then have finite-supp-image: finite $(\text{supp} (f ' ?Q'))$

using finite-supp rev-finite-subset by blast

have $|f ' ?Q'| \leq o |\text{UNIV} :: '\text{state set}|$

by (metis card-of-UNIV card-of-image ordLeq-transitive)

also have $|\text{UNIV} :: '\text{idx set}| < o |\text{UNIV} :: '\text{idx set}|$

by (metis Cnotzero-UNIV ordLeq-csum2)

finally have card-image: $|f ' ?Q'| < o |\text{UNIV} :: '\text{idx set}|$.

let $?y = \text{Conj (Abs-bset (f ' ?Q')) :: ('idx, 'pred, 'act) formula}$

have weak-formula $(\langle \langle \alpha \rangle \rangle y)$

proof (standard+)

show finite $(\text{supp} (\text{Abs-bset (f ' ?Q')} :: - \text{set ['idx]}))$

using finite-supp-image card-image by simp

next

fix $x$ assume $x \in \text{set-bset (Abs-bset (f ' ?Q') :: - set ['idx])}$

with card-image obtain $Q'$ where $Q' \in ?Q'$ and $x = f Q'$

using Abs-bset-inverse imageE set-bset set-bset-to-set-bset by auto

then show weak-formula $x$

using $*$ by metis

qed

moreover have $P \models \langle \langle \alpha \rangle \rangle y$

unfolding valid-weak-action-modality proof (standard+)

from $(P \Rightarrow \langle \alpha, P' \rangle)$ show $P \Rightarrow \langle \alpha \rangle P'$

by simp

next

{
  fix $Q'$
  assume $Q \Rightarrow \langle \alpha \rangle Q'$
  with $*$ have $P' \models f Q'$
    by (metis is-distinguishing-formula-def mem-Collect-eq)
}

then show $P' \models ?y$

by (simp add: finite-supp-image card-image)

qed
moreover have \( \neg Q \models (\langle \alpha \rangle)?y \)
proof
assume \( Q \models (\langle \alpha \rangle)?y \)
then obtain \( Q' \) where
1: \( Q \Rightarrow (\alpha) \) and
2: \( Q' \models ?y \)
using \( \text{bn} \alpha \sharp \ast \text{Q} \) by \( \text{metis} \ \text{valid-weak-action-modality-fresh} \)
from 2 have \( \land Q'' \Rightarrow (\alpha) \) \( Q'' \Rightarrow Q' \Rightarrow f Q'' \)
by \( \text{simp add: finite-supp-image card-image} \)
with 1 and * show False
using is-distinguishing-formula-def by blast
qed
ultimately have False
by \( \text{metis} (P \equiv Q) \ \text{weakly-logically-equivalent-def} \)
} then show \(?thesis by auto
qed
}
ultimately show \(?thesis
unfolding is-weak-bisimulation-def by metis
qed

theorem weak-equivalence-implies-weak-bisimilarity: assumes \( P \equiv Q \) shows \( P \approx Q \)
using assms by (metis weakly-bisimilar-def weak-equivalence-is-weak-bisimulation)
end

end

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

26 Weak Expressive Completeness

context indexed-weak-nominal-ts
begin

26.1 Distinguishing weak formulas

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

Note that this definition uses Hilbert’s choice operator \( \varepsilon \), which is not necessarily equivariant. This is immediately remedied by a hull construction.
definition distinguishing-weak-formula :: 'state ⇒ 'state ⇒ ('idx, 'pred, 'act) formula where
  distinguishing-weak-formula P Q ≡ Conj (Abs-bset {−p · (e.x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes (p · P) from (p · Q))|p. True})

— just an auxiliary lemma that will be useful further below
lemma distinguishing-weak-formula-card-aux:
  |{−p · (e.x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes (p · P) from (p · Q))|p. True}| <o natLeq +c |UNIV :: 'idx set|
proof –
  let ?some = λp. (e.x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes (p · P) from (p · Q))
  let ?B = {−p · ?some p|p. True}

  have ?B ⊆ (λp. −p · ?some p) ' 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 show ?thesis .
qed

— just an auxiliary lemma that will be useful further below
lemma distinguishing-weak-formula-supp-aux:
  assumes ¬(P ≡· Q)
  shows supp (Abs-bset {−p · (e.x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes (p · P) from (p · Q))|p. True}) :: - set[′idx]] ⊆ supp P
proof –
  let ?some = λp. (e.x. weak-formula x ∧ supp x ⊆ supp (p · P) ∧ x distinguishes (p · P) from (p · Q))
  let ?B = {−p · ?some p|p. True}

  { fix p
    from assms have ¬(p · P ≡· p · Q)
    by (metis weakly-logically-equivalent-eqvt permute-minus-cancel(2))
    then have supp (?some p) ⊆ supp (p · P)
      using distinguished-bounded-support by (metis mono-tags, lifting weakly-equivalent-iff-not-distinguished someI-ex)
  }
  note supp-some = this

  { fix x
    assume x ∈ ?B
    then obtain p where x = −p · ?some p

  
  167
by blast
with supp-some have supp (p · x) ⊆ supp (p · P)
by simp
then have supp x ⊆ supp P
by (metis (full-types) permute-boolE subset-eqvt supp-eqvt)
}
note * = this
have supp-B: supp ?B ⊆ supp P
by (rule set-bounded-supp, fact finite-supp, cut-tac *, blast)

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

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

from assms have supp (Abs-bset ?B :: - set[′idx]) ⊆ supp P
by (rule distinguishing-weak-formula-supp-aux)
then have finite (supp (Abs-bset ?B :: - set[′idx]))
using finite-supp rev-finite-subset by blast
with distinguishing-weak-formula-card-aux have *: p · Conj (Abs-bset ?B) = Conj (Abs-bset (p · ?B))
by simp

let ?some' = λp'. (ε.x. weak-formula x ∧ supp x ⊆ supp (p' · p · P) ∧ x distinguishes (p' · p · P) from (p' · p · Q))
let ?B' = {−p' · ?some' p'|p'. True}

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

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

then show $?B' \subseteq p \cdot ?B$
  by blast
qed

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

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

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

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

  { fix $p$
    from assms have $\neg (p \cdot P) \equiv (p \cdot Q)$
      by (metis permute-minus-cancel(2) weakly-logically-equivalent-eqvt)
then have (\(\exists \text{ some } p\)) distinguishes \((p \cdot P)\) from \((p \cdot Q)\) by (metis (mono-tags, lifting) distinguished-bounded-support weakly-equivalent-iff-not-distinguished someI-ex)

}\n
defines some-distinguishes = this

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

\{
fix \(p\)
have \(\lnot p \cdot \exists \text{ some } p\)
by (metis (mono-tags) is-distinguishing-formula-def permute-minus-cancel(2)
some-distinguishes valid-eqvt)
\}
then have \(P \models \text{distinguishing-weak-formula } P Q\)
using valid-distinguishing-formula by blast

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

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

\lemma distinguishing-weak-formula-is-weak:
\assumes \(\lnot (P \equiv \text{Q})\)
\shows \(\text{weak-formula (distinguishing-weak-formula } P \text{ Q})\)
\proof
let \(\exists \text{ some } = \lambda p. (\epsilon x. \text{weak-formula } x \land \text{supp } x \subseteq \text{supp } (p \cdot P) \land x \text{ distinguishes } (p \cdot P))\) from \((p \cdot Q))\)
let \(\exists B \equiv \{ - p \cdot \exists \text{ some } p | p. \text{True}\}\)
from assms have \(\text{supp (Abs-bset } ?B :: \text{- set}['\text{id}']]) \subseteq \text{supp } P\)
by (rule distinguishing-weak-formula-supp-aux)
then have \(\text{finite (supp (Abs-bset } ?B :: \text{- set}['\text{id}']])\)
using \(\text{finite-supp rev-finite-subset by blast}\)
moreover have \(\text{set-bset (Abs-bset } ?B :: \text{- set}['\text{id}']]) = ?B\)
using distinguishing-weak-formula-card-aux Abs-bset-inverse' by simp

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

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

26.2 Characteristic weak formulas

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

definition characteristic-weak-formula :: 'state ⇒ ('idx, 'pred, 'act) formula where
  characteristic-weak-formula P ≡ Conj (Abs-bset {distinguishing-weak-formula P Q | Q. ¬ (P ≡ Q)})

— just an auxiliary lemma that will be useful further below

lemma characteristic-weak-formula-card-aux:
  |{distinguishing-weak-formula P Q | Q. ¬ (P ≡ Q)}| < o natLeq + c |UNIV :: 'idx set|
proof —
  let ?B = {distinguishing-weak-formula P Q | Q. ¬ (P ≡ Q)}

  have ?B ⊆ (distinguishing-weak-formula P) ⋅ UNIV
    by auto

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

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

  also have |UNIV :: 'idx set| ≤ o natLeq + c |UNIV :: 'idx set|
    by (metis Cnotzero-UNIV ordLeq-csum2)

  finally show ?thesis .
qed

— just an auxiliary lemma that will be useful further below
lemma characteristic-weak-formula-supp-aux:
  shows supp (Abs-bset \{distinction-weak-formula \ P \ Q. \ \neg (\ P \equiv \ Q)\} :: -
  set[‘idt]) \subseteq supp \ P
proof –
  let ?B = \{distinction-weak-formula \ P \ Q. \ \neg (\ P \equiv \ Q)\}

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

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

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

lemma characteristic-weak-formula-eqvt [simp]:
  \ P \cdot characteristic-weak-formula \ P = characteristic-weak-formula \ (\ P \cdot \ P)
proof –
  let ?B = \{distinction-weak-formula \ P \ Q. \ \neg (\ P \equiv \ Q)\}

  have supp (Abs-bset ?B :: - set[‘idt]) \subseteq supp \ P
  by (fact characteristic-weak-formula-supp-aux)
  then have finite (supp (Abs-bset ?B :: - set[‘idt]))
  using finite-supp rev-finite-subset by blast
  with characteristic-weak-formula-card-aux have *: \ P \cdot Conj (Abs-bset ?B) = 
  Conj (Abs-bset (\ P \cdot \ ?B))
  by simp

  let ?B' = \{distinction-weak-formula \ (\ P \cdot \ P) \ Q. \ \neg ((\ P \cdot \ P) \equiv \ Q)\}

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

{ }
then show \( p \cdot ?B \subseteq ?B' \)
  by blast

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

} then show \( ?B' \subseteq p \cdot ?B \)
  by blast
qed

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

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

lemma characteristic-weak-formula-is-weak:
  weak-formula (characteristic-weak-formula P)
proof
  let \( ?B = \{ \text{distinguishing-weak-formula } P \text{ Q} \mid \neg (P \equiv Q) \} \)
  have supp (Abs-bset ?B :: - set['idx]) \( \subseteq \) supp P
    by (fact characteristic-weak-formula-supp-aux)
  then have finite (supp (Abs-bset ?B :: - set['idx]))
    using finite-supp rev-finite-subset by blast
  moreover have set-bset (Abs-bset ?B :: - set['idx]) = ?B
    using characteristic-weak-formula-card-aux Abs-bset-inverse' by simp
  moreover
\{ 
  \text{fix } x \\
  \text{assume } x \in \mathcal{B} \\
  \text{then have } \text{weak-formula } x \\
  \hspace{1em} \text{using } \text{distinguishing-weak-formula-is-weak by blast} \\
\}\n
\text{ultimately show } \text{?thesis} \\
\hspace{1em} \text{unfolding } \text{characteristic-weak-formula-def using } \text{uf-Conj by presburger} \\
\text{qed}

\text{lemma } \text{characteristic-weak-formula-is-characteristic}: \\
\hspace{1em} Q \models \text{characteristic-weak-formula } P \iff P \equiv Q \\
\text{proof --} \\
\hspace{2em} \text{let } \mathcal{B} = \{ \text{distinguishing-weak-formula } P Q \mid Q. \neg (P \equiv \cdot Q) \}\n
\{ 
  \text{fix } P' \\
  \text{have } \text{supp} (\text{Abs-bset } \mathcal{B} :: - \text{set} ['idx']) \subseteq \text{supp } P \\
  \hspace{1em} \text{by } (\text{fact characteristic-weak-formula-suppp-aux}) \\
  \text{then have } \text{finite } (\text{supp} (\text{Abs-bset } \mathcal{B} :: - \text{set} ['idx])) \\
  \hspace{1em} \text{using } \text{finite-suppp rev-finite-subset by blast} \\
  \hspace{2em} \text{with } \text{characteristic-weak-formula-card-aux have } P' \models \text{characteristic-weak-formula } P \iff (\forall x \in \mathcal{B}. P' \models x) \\
  \hspace{4em} \text{unfolding } \text{characteristic-weak-formula-def by simp} \\
\}\n
\text{note } \text{valid-characteristic-formula = this} \\

\text{show } \text{?thesis} \\
\text{proof} \\
\hspace{1em} \text{assume } *: Q \models \text{characteristic-weak-formula } P \\
\hspace{2em} \text{show } P \equiv Q \\
\hspace{4em} \text{proof } (\text{rule ccontr}) \\
\hspace{6em} \text{assume } \neg (P \equiv Q) \\
\hspace{7em} \text{with } * \text{ show False} \\
\hspace{10em} \text{using } \text{distinguishing-weak-formula-distinguishes is-distinguishing-formula-def valid-characteristic-formula by auto} \\
\hspace{12em} \text{qed} \\
\text{next} \\
\hspace{1em} \text{assume } P \equiv Q \\
\hspace{2em} \text{moreover have } P \models \text{characteristic-weak-formula } P \\
\hspace{4em} \text{using } \text{distinguishing-weak-formula-distinguishes is-distinguishing-formula-def valid-characteristic-formula by auto} \\
\hspace{6em} \text{ultimately show } Q \models \text{characteristic-weak-formula } P \\
\hspace{8em} \text{using } \text{weakly-logically-equivalent-def characteristic-weak-formula-is-weak by blast} \\
\hspace{10em} \text{qed} \\
\text{qed}
lemma characteristic-weak-formula-is-characteristic:
\[ Q \models \text{characteristic-weak-formula} \ P \iff P \approx Q \]
using characteristic-weak-formula-is-characteristic' by (meson weak-bisimilarity-implies-weak-equivalence weak-equivalence-implies-weak-bisimilarity)

26.3 Weak expressive completeness

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

theorem weak-expressive-completeness:
assumes finite \((\supp S)\)
and \(\forall P Q. P \in S \Rightarrow P \approx Q \Rightarrow Q \in S\)
shows \(P \models \text{Disj} (\text{Abs-bset} (\text{characteristic-weak-formula} \cdot S)) \iff P \in S\)
and weak-formula (Disj (Abs-bset (characteristic-weak-formula \cdot S)))

proof
let \(?B = \text{characteristic-weak-formula} \cdot S\)

have \(?B \subseteq \text{characteristic-weak-formula} \cdot \text{UNIV}\)
by auto
then have \(|?B| \leq o |\text{UNIV} :: \text{state set}|\)
by (rule surj-imp-ordLeq)
also have \(|\text{UNIV} :: \text{state set}| < o |\text{UNIV} :: \text{idx set}|\)
by (metis card-idx-state)
also have \(|\text{UNIV} :: \text{idx set}| \leq o \text{natlLeq} + c |\text{UNIV} :: \text{idx set}|\)
by (metis Cnotzero-UNIV ordLeq-csum2)
finally have \(\text{card-}\): \(|?B| < o \text{natlLeq} + c |\text{UNIV} :: \text{idx set}|\)

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

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

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

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

have eqvt Formula.Not

175
by (simp add: eqvtI)
with supp-B and (eqvt image) have supp-Not-B: supp (Formula.Not ' ?B) ⊆ supp S
  using supp-fun-eqvt supp-fun-app supp-fun-app-eqvt by blast

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

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

moreover have ∃x. x ∈ Formula.Not ' ?B =⇒ weak-formula x
  using characteristic-weak-formula-is-weak wf-Not by auto

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

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

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

end

end

theory S-Transform

imports
  Bisimilarity-Implies-Equivalence
  Equivalence-Implies-Bisimilarity
  Weak-Bisimilarity-Implies-Equivalence
  Weak-Equivalence-Implies-Bisimilarity
  Weak-Expressive-Completeness

begin
27  $S$-Transform: State Predicates as Actions

27.1  Actions and binding names

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

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

fun permute-$S$-action :: perm ⇒ ('a,'b) $S$-action ⇒ ('a,'b) $S$-action where
   p · (Act α) = Act (p · α)
   | p · (Pred ϕ) = Pred (p · ϕ)

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

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

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

lemma supp-Pred [simp]: supp (Pred ϕ) = supp ϕ
unfolding supp-def by simp

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

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

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

fun bn-$S$-action :: ('a,'b) $S$-action ⇒ atom set where
\[ \text{bn-S-action (Act } \alpha) = \text{bn } \alpha \]
\[ \mid \text{bn-S-action (Pred -) } = \{\} \]

\textbf{instance proof}

\begin{verbatim}
  fix p and α :: ('a,'b) S-action
  show p · bn α = bn (p · α)
    by (cases α) (simp add: bn-eqvt, simp)
\end{verbatim}

\textbf{next}

\begin{verbatim}
  fix α :: ('a,'b) S-action
  show finite (bn α)
    by (cases α) (simp add: bn-finite, simp)
\end{verbatim}

\textbf{qed}

\textbf{end}

\section{27.2 Satisfaction}

\textbf{context nominal-ts}

\textbf{begin}

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

\textbf{definition} $S$-satisfies :: 'state ⇒ unit ⇒ bool (infix $\vdash_S 70$) where

\begin{verbatim}
P $\vdash_S \varphi$ $\iff$ False
\end{verbatim}

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

\textbf{end}

\section{27.3 Transitions}

\textbf{context nominal-ts}

\textbf{begin}

\textbf{inductive} $S$-transition :: 'state ⇒ (('act,'pred) S-action, 'state) residual ⇒ bool (infix $\rightarrow_S 70$) where

\begin{verbatim}
  Act: P → (α,P) → S (Act α,P)
  | Pred: P ⊢ ϕ → P → S (Pred ϕ,P)
\end{verbatim}

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

If there is an $S$-transition, there is an ordinary transition with the same
residual—it is not necessary to consider alpha-variants.

**lemma S-transition-cases** [case-names Act Pred, consumes 1]; assumes \( P \rightarrow S \)

\[ \langle \alpha_S, P' \rangle \]

and \( \alpha_S = \text{Act} \alpha \implies P \rightarrow \langle \alpha, P' \rangle \implies R \)

and \( \alpha_S = \text{Pred} \varphi \implies P' = P \implies P \vdash \varphi \implies R \)

shows \( R \)

using \( \text{assms} \) proof (cases rule: S-transition.cases)

\( \text{case } (\text{Act } \alpha') P'' \) \( \text{let } \langle \alpha_S, P' \rangle = (\text{Act } \alpha') \langle \alpha', P'' \rangle \) \( \text{obtain } \alpha \text{ where } \alpha_S = \text{Act } \alpha \) \( \text{by } (\text{meson bn-S-action.elims residual-empty-bn-eq-iff}) \)

with \( \langle \alpha_S, P' \rangle = (\text{Act } \alpha', P'') \) \( \text{obtain } \varphi \text{ where } \text{supp} \langle ?\text{Act } \alpha, P' \rangle - bn \langle ?\text{Act } \alpha \rangle \) \( \alpha = \text{supp} \langle ?\text{Act } \alpha', P'' \rangle - bn \langle ?\text{Act } \alpha' \rangle \)

and \( \langle ?\text{Act } \alpha, P' \rangle - bn \langle ?\text{Act } \alpha \rangle \) \( \varphi \) and \( \varphi \cdot \langle ?\text{Act } \alpha, P' \rangle = (\text{Act } \alpha', P'') \) and \( p \cdot bn \langle ?\text{Act } \alpha \rangle = bn \langle ?\text{Act } \alpha' \rangle \)

by \( (\text{auto simp add: residual-eq-iff-perm}) \)

then have \( \varphi \cdot \langle ?\text{Act } \alpha, P' \rangle - bn \langle ?\text{Act } \alpha' \rangle = \text{supp} \langle \alpha, P' \rangle - bn \langle \alpha' \rangle \) \( \text{and } (\text{supp } \langle ?\text{Act } \alpha, P' \rangle - bn \langle ?\text{Act } \alpha \rangle) \) \( \varphi \) and \( \varphi \cdot bn \langle ?\text{Act } \alpha \rangle = bn \langle ?\text{Act } \alpha' \rangle \)

by \( (\text{metis residual-eq-iff-perm}) \)

with \( \langle \alpha_S, P' \rangle = (\text{Act } \alpha') \langle \alpha', P'' \rangle \) show \( R \)

using \( (\text{\( \land \)} \alpha. \alpha_S = \text{Act } \alpha \implies P \rightarrow \langle \alpha, P' \rangle \implies R) \) by \( \text{metis} \)

next

\( \text{case } (\text{Pred } \varphi) \) \( \text{from } \langle \alpha_S, P' \rangle = (\text{Pred } \varphi, P') \) have \( \alpha_S = \text{Pred } \varphi \) and \( P' = P \)

by \( (\text{metis bn-S-action.simps(2) residual-empty-bn-eq-iff}) \)

with \( P \vdash \varphi \) show \( R \)

using \( (\text{\( \land \)} \varphi. \alpha_S = \text{Pred } \varphi \implies P' = P \implies P \vdash \varphi \implies R) \) by \( \text{metis} \)

qed

**lemma S-transition-Act-iff**: \( P \rightarrow S \langle \text{Act } \alpha, P' \rangle \iff P \rightarrow \langle \alpha, P' \rangle \)

using S-transition.Aact S-transition-cases by fastforce

**lemma S-transition-Pred-iff**: \( P \rightarrow S \langle \text{Pred } \varphi, P' \rangle \iff P' = P \land P \vdash \varphi \)

using S-transition.Pred S-transition-cases by fastforce

end

27.4 Strong Bisimilarity in the S-transform

context nominal-ts

begin

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

no-notation S-satisfies (infix \( \tau_S \ 70 \)) — denotes \((\tau_S)\) instead
notation \( S\text{-transform}.\text{bisimilar} \) (infix \( \sim_S 100 \))

Bisimilarity is equivalent to bisimilarity in the \( S\text{-transform}.\)

\textbf{lemma bisimilar-is-S-transform-bisimulation:} \( S\text{-transform}.\text{is-bisimulation} \text{ bisimilar} \)

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

\textbf{proof}\n
show symp bisimilar
  by (fact bisimilar-symp)

next
  have \( \forall P \, Q. \, P \sim_S Q \rightarrow (\forall \varphi. \, P \vdash_S \varphi \rightarrow Q \vdash_S \varphi) \) (is \( ?S \))
  by (simp add: \( S\text{-transform}.\text{S-satisfies-def} \))

moreover have \( \forall P \, Q. \, P \sim_S Q \rightarrow (\forall \alpha_S P'. \, \text{bn} \, \alpha_S \, \sharp* \, Q \rightarrow P \rightarrow_S \langle \alpha_S, P' \rangle \)
\rightarrow (\exists Q'. \, Q \rightarrow_S \langle \alpha_S, Q' \rangle \wedge P' \sim_S Q') \) (is \( ?T \))

\textbf{proof} (clarify)
  fix \( P \, Q \, \alpha_S \, P' \)
  assume \( \text{bisim:} \, P \sim_S Q \) \text{ and } \text{fresh}_S: \text{bn} \, \alpha_S \, \sharp* \, Q \text{ and } \text{trans}_S: \, P \rightarrow_S \langle \alpha_S, P' \rangle
  obtain \( Q' \) where \( \text{trans}_S \) \( \rightarrow_S \langle \alpha_S, Q' \rangle \) \text{ and } \( P' \sim_S Q' \)
  using \( \text{trans}_S \) \( \text{proof} \) (cases rule: \( S\text{-transition-cases} \))
  case \( (\text{Act} \, \alpha) \)
  from \( \alpha_S = \text{Act} \, \alpha \) \text{ and } \text{fresh}_S \text{ have } bn \, \alpha \, \sharp* \, Q
  by simp
  with \( \text{bisim} \) \text{ and } \( \langle P \rightarrow \langle \alpha, P' \rangle \rangle \) \text{ obtain } \( Q' \) where \( \text{trans}_Q: \, Q \rightarrow_S \langle \alpha, Q' \rangle \)
  and \( \text{bisim'}: \, P' \sim_S Q' \)
  by (metis bisimilar-simulation-step)
  from \( \alpha_S = \text{Act} \, \alpha \) \text{ and } \text{trans}_Q \text{ have } Q \rightarrow_S \langle \alpha_S, Q' \rangle
  by (simp add: \( S\text{-transition}.\text{Act} \))
  with \( \text{bisim'} \) \text{ show } \text{thesis}
  using \( \langle \wedge Q', \, Q \rightarrow_S \langle \alpha_S, Q' \rangle \rangle \rightarrow P' \sim_S Q' \rightarrow \text{thesis} \) \text{ by blast}

next
  case \( (\text{Pred} \, \varphi) \)
  from \( \text{bisim} \) \text{ and } \( P \vdash \varphi \) \text{ have } Q \vdash \varphi
  by (metis is-bisimulation-def bisimilar-is-bisimulation)
  with \( \alpha_S = \text{Pred} \, \varphi \) \text{ have } Q \rightarrow_S \langle \alpha_S, Q \rangle
  by (simp add: \( S\text{-transition}.\text{Pred} \))
  with \( \text{bisim} \) \text{ and } \( P' = P \) \text{ show } \text{thesis}
  using \( \langle \wedge Q', \, Q \rightarrow_S \langle \alpha_S, Q' \rangle \rangle \rightarrow P' \sim_S Q' \rightarrow \text{thesis} \) \text{ by blast}
  qed

  then show \( \exists Q'. \, Q \rightarrow_S \langle \alpha_S, Q' \rangle \wedge P' \sim_S Q' \)
  by auto
  qed

ultimately show \( ?S \wedge ?T \)
  by metis
  qed

\textbf{lemma S-transform-bisimilar-is-bisimulation:} \( \text{is-bisimulation} \, S\text{-transform}.\text{bisimilar} \)

unfolding \( \text{is-bisimulation-def} \)

\textbf{proof}\n
show symp \( S\text{-transform}.\text{bisimilar} \)

180
by (fact $S$-transform.bisimilar-symp)

next
have $\forall P \; Q. \; P \sim_{S} Q \Rightarrow (\forall \varphi. \; P \vdash \varphi \Rightarrow Q \vdash \varphi)$ (is $?S$)
proof (clarify)
  fix $P \; Q \; \varphi$
  assume bisim: $P \sim_{S} Q$ and valid: $P \vdash \varphi$
  from valid have $P \rightarrow_{S} (\text{Pred } \varphi, P)$
    by (fact $S$-transition.Pred)
  moreover have $\text{bn}(\text{Pred } \varphi) \sharp * Q$
    by (simp add: fresh-star-def)
ultimately obtain $Q' \; \text{where trans'}: \; Q \rightarrow_{S} (\text{Pred } \varphi, Q')$
  using bisim by (metis $S$-transform.bisimilar-simulation-step)
  from trans' show $Q \vdash \varphi$
  using $S$-transition-Pred-iff by blast
qed

moreover have $\forall P \; Q. \; P \sim_{S} Q \Rightarrow (\forall \alpha \; P'. \; \text{bn } \alpha \sharp * Q \rightarrow P \rightarrow \langle \alpha, P' \rangle \rightarrow 
(\exists Q'. \; Q \rightarrow \langle \alpha, Q' \rangle \land P' \sim_{S} Q'))$ (is $?T$)
proof (clarify)
  fix $P \; Q \; \alpha \; P'$
  assume bisim: $P \sim_{S} Q$ and fresh: $\text{bn } \alpha \sharp * Q$ and trans: $P \rightarrow \langle \alpha, P' \rangle$
  from trans have $P \rightarrow_{S} (\text{Act } \alpha, P')$
    by (fact $S$-transition.Act)
  with bisim and fresh obtain $Q' \; \text{where trans'}: \; Q \rightarrow_{S} (\text{Act } \alpha, Q')$ and
bisim': $P' \sim_{S} Q'$
    by (metis $S$-transform.bisimilar-simulation-step bn-S-action.simps(1))
  from trans' have $Q \rightarrow \langle \alpha, Q' \rangle$
    by (metis $S$-transition-Act-iff)
  with bisim' show $\exists Q'. \; Q \rightarrow \langle \alpha, Q' \rangle \land P' \sim_{S} Q'$
    by metis
qed
ultimately show $?S \land ?T$
by metis
qed

theorem $S$-transform-bisimilar-iff: $P \sim_{S} Q \iff P \sim_{S} Q$
proof
  assume $P \sim_{S} Q$
  then show $P \sim_{S} Q$
    by (metis $S$-transform-bisimilar-is-bisimulation bisimilar-def)
next
  assume $P \sim_{S} Q$
  then show $P \sim_{S} Q$
    by (metis $S$-transform-bisimilar-def bisimilar-is-S-transform-bisimulation)
qed
end
27.5 Weak Bisimilarity in the $S$-transform

context weak-nominal-ts

begin

lemma weakly-bisimilar-tau-transition-weakly-bisimilar:
assumes $P \approx R$ and $P \Rightarrow Q$ and $Q \Rightarrow R$
shows $Q \approx R$
proof
  let $\text{?bisim} = \lambda S T. S \approx T \lor \{S,T\} = \{Q,R\}$
  have is-weak-bisimulation $\text{?bisim}$
  unfolding is-weak-bisimulation-def
  proof
    show symp $\text{?bisim}$
    using weakly-bisimilar-symp
    by (simp add: insert-commute symp-def)
  next
    have $\forall S T. ?\text{bisim} S T \land S \vdash \varphi \rightarrow (\exists T'. T \Rightarrow T' \land ?\text{bisim} S T' \land T' \vdash \varphi)$ (is $\?S$)
    proof (clarify)
      fix $S T \varphi$
      assume bisim: $?\text{bisim} S T$ and valid: $S \vdash \varphi$
      from bisim show $\exists T'. T \Rightarrow T' \land ?\text{bisim} S T' \land T' \vdash \varphi$
      proof
        assume $S \approx R$
        with valid show $\?\text{thesis}$
        by (metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation)
      next
        assume $\{S, T\} = \{Q, R\}$
        then have $S = Q \land T = R \lor T = Q \land S = R$
        by (metis doubleton-eq-iff)
        then show $?\text{thesis}$
        proof
          assume $S = Q \land T = R$
          with $P \Rightarrow Q$, $P \approx R$ and valid show $\?\text{thesis}$
          by (metis is-weak-bisimulation-def weakly-bisimilar-is-weak-bisimulation weakly-bisimilar-tau-transition-trans)
        next
          assume $T = Q \land S = R$
          with $Q \Rightarrow R$ and valid show $\?\text{thesis}$
          by (meson reflpE weakly-bisimilar-reflp)
        qed
      qed
    qed
  moreover have $\forall S T. ?\text{bisim} S T \rightarrow (\forall \alpha S'. \text{bn }\alpha \not\in T \rightarrow S \rightarrow \langle\alpha,S'\rangle \rightarrow (\exists T'. T \Rightarrow \langle\alpha\rangle T' \land ?\text{bisim} S T')$ (is $\?T$)
    proof (clarify)
      fix $S T \alpha S'$
      assume bisim: $?\text{bisim} S T$ and fresh: $\text{bn }\alpha \not\in T$ and trans: $S \rightarrow \langle\alpha,S'\rangle$
      from bisim show $\exists T'. T \Rightarrow \langle\alpha\rangle T' \land ?\text{bisim} S T'$
      proof

assume $S \simeq T$

with fresh and trans show $\text{thesis}$

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

next

assume $\{S, T\} = \{Q, R\}$
then have $S = Q \land T = R \lor T = Q \land S = R$

by (metis doubleton-eq-iff)

then show $\text{thesis}$

proof

assume $S = Q \land T = R$

with $\langle P \Rightarrow Q \rangle$ and $\langle P \approx R \rangle$ and fresh and trans show $\text{thesis}$

using observable-transition-stepI tau-refl weak-transition-stepI weak-transition-weakI

weakly-bisimilar-weak-simulation-step by blast

next

assume $T = Q \land S = R$

with $\langle Q \Rightarrow R \rangle$ and trans show $\text{thesis}$

by (metis observable-transition-stepI reflpE tau-refl weak-transition-stepI

weak-transition-weakI weakly-bisimilar-reflp)

qed

qed

ultimately show $\neg S \land \neg T$

by metis

then show $\text{thesis}$

using weakly-bisimilar-def by blast

qed

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

interpretation $S$-transform: weak-nominal-ts $(\vdash_S) (\rightarrow_S) \text{Act } \sigma$

by unfold-locales (fact $S$-satisfies-eqvt, fact $S$-transition-eqvt, simp add: tau-eqvt)

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

notation $S$-transform.tau-transition (infix $\Rightarrow_S 70$)
notation $S$-transform.observe-transition (infix $\Rightarrow_S 70, 70, 71$)
notation $S$-transform.weak-transition (infix $\Rightarrow_S 70, 70, 71$)
notation $S$-transform.weakly-bisimilar (infix $\approx\cdot_S 100$)

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

proof

assume $P \Rightarrow_S P'$
then show $P \Rightarrow P'$

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

next

assume $P \Rightarrow P'$
then show $P \Rightarrow_S P'$

by induct (simp, metis $S$-transform.tau-transition.simpts $S$-transition.Act)
Weak bisimilarity is equivalent to weak bisimilarity in the S-transform.

\textbf{lemma} weakly-bisimilar-is-S-transform-weak-bisimulation: \( S \text{-transform.weakly-bisimilar-is-weak-bisimulation} \)
\begin{proof}
\begin{enumerate}
\item \textbf{show} symp weakly-bisimilar
\item \textbf{by} (\text{fact weakly-bisimilar-symp})
\end{enumerate}
\end{proof}
lemma S-transform-weakly-bisimilar-is-weak-bisimulation: is-weak-bisimulation
S-transform.weakly-bisimilar
unfolding is-weak-bisimulation-def
proof
  show symp S-transform.weakly-bisimilar
  by (fact S-transform.weakly-bisimilar-symp)
next
  have ∀ P Q. P ≈· S Q ∧ P ⊢ ϕ → (∃ Q’. Q ⊨ (αS)Q Q’ ∧ P’ ≈· Q’)
  (is ?S)
  proof (clarify)
    fix P Q ϕ
    assume bisim: P ≈· S Q and valid: P ⊢ ϕ
    from valid have P ⊨ (Pred ϕ)S P
      by (simp add: S-transition.Pred)
    moreover have bn (Pred ϕ) #* Q
      by (simp add: fresh-star-def)
    ultimately obtain Q'' where trans': Q ⊨ (Pred ϕ)S Q'' and bisim': P ≈· S Q''
    using bisim by (metis S-transform.weakly-bisimilar-weak-simulation-step)
    from trans' obtain Q' Q₁ where trans₁: Q ⊨ S Q’ and trans₂: Q’ ⊨ S (Pred ϕ, Q₁)
    (is ?S)
    proof (clarify)
      fix P Q α P' Q''
      assume eq: Q₁ = Q' and Q'' ⊨ ϕ
      using S-transition-Pred-iff by blast+
      moreover from trans₁ and trans₂ and eq and bisim and bisim' have P ≈· S Q''
      by (metis S-transform.weakly-bisimilar-equivp S-transform.weakly-bisimilar-tau-transition-weakly-bisimilar-equiv-def)
    moreover from trans₁ have Q ⇒ Q’
      by (metis S-transform-tau-transition-iff)
    ultimately show (∃ Q’. Q ⊨ S Q’ ∧ P ≈· S Q’ ⊨ ϕ)
      by metis
    qed
  moreover have ∀ P Q. P ≈· S Q → (∀ P’. bn α #* Q → P → (α,P') →
    (∃ Q’. Q ⊨ (α) Q’ ∧ P’ ≈· S Q’)) (is ?T)
  proof (clarify)
    fix P Q α P'
assume bisim: \( P \approx_S Q \) and fresh: \( bn \ \alpha \# Q \) and trans: \( P \rightarrow (\alpha, P') \)
from trans have \( P \rightarrow_S (\alpha, P') \)
by (fact S-transition.Act)
with bisim and fresh obtain \( Q' \) where trans': \( Q \Rightarrow (\alpha) Q' \) and
bisim': \( P' \approx_{S'} Q' \)
by (metis S-transform.is-weak-bisimulation-def S-transform.weakly-bisimilar-is-weak-bisimulation
bn-S-action.simps(1))
from trans' have \( Q \Rightarrow (\alpha) Q' \)
by (metis S-transform-weak-transition-iff)
with bisim' show \( \exists Q'. \ Q \Rightarrow (\alpha) Q' \land P' \approx_{S'} Q' \)
by metis
qed
ultimately show \( ?S \land ?T \)
by metis
qed

theorem S-transform-weakly-bisimilar-iff: \( P \approx_{S} Q \leftrightarrow P \approx_{S} Q \)
proof
assume \( P \approx_{S} Q \)
then show \( P \approx_{S} Q \)
by (metis S-transform-weakly-bisimilar-is-weak-bisimulation weakly-bisimilar-def)
next
assume \( P \approx_{S} Q \)
then show \( P \approx_{S} Q \)
by (metis S-transform.weakly-bisimilar-def weakly-bisimilar-is-S-transform-weak-bisimulation)
qed

end

27.6 Translation of (strong) formulas into formulas without predicates

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

The following auxiliary function returns trees (modulo \( \alpha \)-equivalence) rather than formulas. This allows us to prove equivariance for all argument trees, without an assumption that they are (hereditarily) finitely supported. Further below–after this auxiliary function has been lifted to (strong) formulas as arguments–we derive a version that returns formulas.

primrec S-transform-Tree :: ('idx,'pred::fs,'act::bn) Tree \Rightarrow ('idx, unit, ('act,'pred) S-action) Tree, where
S-transform-Tree (tConj tset) = Conj\(\alpha\) (map-bset S-transform-Tree tset)
| S-transform-Tree (tNot t) = Not\(\alpha\) (S-transform-Tree t)
| S-transform-Tree (tPred \( \varphi \)) = Act\(\alpha\) (S-action.Pred \( \varphi \)) (Conj\(\alpha\) bempty)
lemma S-transform-Tree-eqvt [eqvt]: p • S-transform-Tree t = S-transform-Tree (p • 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

S-transform-Tree respects α-equivalence.

lemma alpha-Tree-S-transform-Tree:
  assumes t1 =_α t2
  shows S-transform-Tree t1 = S-transform-Tree t2
using assms proof (induction t1 t2 rule: alpha-Tree-induct)
  case (alpha-tConj tset1 tset2)
  then have rel-bset (=) (map-bset S-transform-Tree tset1) (map-bset S-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 α 1 t1 α 2 t2)
  from ⟨tAct α 1 t1 = α tAct α 2 t2⟩
  obtain p where *: (bn α 1, t1) ≈set alpha-Tree (supp-rel alpha-Tree) p (bn α 2, t2)
    and **: (bn α 1, α 1) ≈set (=) supp p (bn α 2, α 2)
    by auto
  from * have fresh: (supp-rel alpha-Tree t1 − bn α 1) ∗ p and alpha: p • t1 =_α t2 and eq: p • bn α 1 = bn α 2
    by (auto simp add: alpha-set)
  from alpha-tAct.IH(2) have supp-rel alpha-Tree (rep-Tree α (S-transform-Tree t1)) ⊆ supp-rel alpha-Tree t1
    by (metis (no-types, lifting) infinite-mono alpha-Tree-permute-rep-commute S-transform-Tree-eqvt mem-Collect-eq subsetI supp-rel-def)
  with fresh have fresh': (supp-rel alpha-Tree (rep-Tree α (S-transform-Tree t1))) − bn α 1 ∗ p
    by (meson DiffD1 DiffD2 DiffI fresh-star-def subsetCE)
  moreover from alpha have alpha': p • rep-Tree α (S-transform-Tree t1) =_α rep-Tree α (S-transform-Tree t2)
    using alpha-tAct.IH(1) by (metis alpha-Tree-permute-rep-commute S-transform-Tree-eqvt)
  moreover from fresh' alpha' eq have supp-rel alpha-Tree (rep-Tree α (S-transform-Tree t1)) − bn α 1 = supp-rel alpha-Tree (rep-Tree α (S-transform-Tree t2)) − bn α 2
    by (metis (mono-tags) Diff-eqvt alpha-Tree-eqvt-alpha-Tree-eqvt-aux alpha-Tree-supp-rel atom-set-perm-eq)
  ultimately have (bn α 1, rep-Tree α (S-transform-Tree t1)) ≈set alpha-Tree (supp-rel alpha-Tree) p (bn α 2, rep-Tree α (S-transform-Tree t2))
    using eq by (simp add: alpha-set)
moreover from ** have \((bn \; \alpha_1, \; S\text{-}action.\;\text{Act} \; \alpha_1) \approx \text{set} \; (=) \; \text{supp} \; p \; (bn \; \alpha_2, \; S\text{-}action.\;\text{Act} \; \alpha_2)\) by (metis (mono-tags, lifting) \text{S\text{-}Transform.supp-Act alpha-set permute-S\text{-}action.simps}(1))

ultimately have \(\text{Act}_\alpha (S\text{-}action.\;\text{Act} \; \alpha_1) (S\text{-}transform\text{-Tree} \; t_1) = \text{Act}_\alpha (S\text{-}action.\;\text{Act} \; \alpha_2) (S\text{-}transform\text{-Tree} \; t_2)\)

by (auto simp add: \text{Act}_\alpha\text{-eq-iff})

then show ?case by simp

qed simp-all

\text{S\text{-}transform} \; \text{for} \; \text{trees} \; \text{modulo} \; \alpha\text{-equivalence}.

lift-definition \text{S\text{-}Transform\text{-Tree}_\alpha} :: ('idx::fs, 'act::bn) Tree_\alpha \Rightarrow ('idx, \; \text{unit} , ('act, \; \text{pred}) \text{S\text{-}action}) \; \text{Tree}_\alpha

by (fact alpha-Tree-S\text{-}Transform\text{-Tree})

lemma \text{S\text{-}transform\text{-Tree}_\alpha\text{-eqvt}} (eqvt): \(p \cdot \text{S\text{-}transform\text{-Tree}_\alpha t_\alpha = \text{S\text{-}transform\text{-Tree}_\alpha (p \cdot t_\alpha)}\)

by transfer simp

lemma \text{S\text{-}transform\text{-Tree}_\alpha\text{-Conj}_\alpha} (simp): \(\text{S\text{-}transform\text{-Tree}_\alpha (Conj_\alpha tset_\alpha) = Conj_\alpha (map\text{-}bset \; \text{S\text{-}transform\text{-Tree}_\alpha tset_\alpha})}\)

by (simp add: Conj_\alpha\text{-def'} S\text{-}transform\text{-Tree}_\alpha.abs-eq (metis (no-types, lifting) S\text{-}transform\text{-Tree}_\alpha.rep-eq bset.map-comp bset.map-comp0 comp-app))

lemma \text{S\text{-}transform\text{-Tree}_\alpha\text{-Not}_\alpha} (simp): \(S\text{-}transform\text{-Tree}_\alpha (Not_\alpha t_\alpha) = \text{Not}_\alpha (S\text{-}transform\text{-Tree}_\alpha t_\alpha)\)

by transfer simp

lemma \text{S\text{-}transform\text{-Tree}_\alpha\text{-Pred}_\alpha} (simp): \(S\text{-}transform\text{-Tree}_\alpha (Pred_\alpha \varphi) = Act_\alpha (S\text{-}action.\text{Pred} \varphi) (Conj_\alpha \text{bempty})\)

by transfer simp

lemma \text{S\text{-}transform\text{-Tree}_\alpha\text{-Act}_\alpha} (simp): \(S\text{-}transform\text{-Tree}_\alpha (Act_\alpha \alpha t_\alpha) = Act_\alpha (S\text{-}action.\text{Act} \alpha) (S\text{-}transform\text{-Tree}_\alpha t_\alpha)\)

by transfer simp

lemma \text{finite\text{-}supp\text{-}map\text{-}bset\text{-}S\text{-}transform\text{-Tree}_\alpha} (simp):

assumes finite (supp tset_\alpha)

shows finite (supp (map\text{-}bset \; S\text{-}transform\text{-Tree}_\alpha tset_\alpha))

proof -

have eqvt map\text{-}bset and eqvt S\text{-}transform\text{-Tree}_\alpha

by (simp add: eqvtI)+

then have supp (map\text{-}bset \; S\text{-}transform\text{-Tree}_\alpha) = \{\}

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

then have supp (map\text{-}bset \; S\text{-}transform\text{-Tree}_\alpha tset_\alpha) \subseteq supp tset_\alpha

using supp-fun-app by blast

with asms show finite (supp (map\text{-}bset \; S\text{-}transform\text{-Tree}_\alpha tset_\alpha))

by (metis finite-subset)
qed

lemma S-transform-Tree_α-preserved-hereditarily-fs:
  assumes hereditarily-fs t_α
  shows hereditarily-fs (S-transform-Tree_α t_α)
using assms proof (induct rule: hereditarily-fs.induct)
  case (Conj_α tset_α)
  then show ?case
    by (auto intro: hereditarily-fs.Conj_α) (metis imageE map-bset.rep-eq)
next
  case (Not_α t_α)
  then show ?case
    by simp add: hereditarily-fs.Not_α
next
  case (Pred_α ϕ)
  have finite (supp bempty) by simp add: eqvtI supp-fun-eqvt
  then show ?case
    using hereditarily-fs.Act_α finite-supp-implies-hereditarily-fs-Conj_α by fastforce
next
  case (Act_α t_α α)
  then show ?case
    by simp add: Formula.hereditarily-fs.Act_α
qed

S-transform for (strong) formulas.

lift-definition S-transform-formula :: ('idx,'pred::fs,'act::bn) formula ⇒ ('idx, unit, ('act,'pred) S-action) Tree_α is
  S-transform-Tree_α
.

lemma S-transform-formula-eqvt [eqvt]: p · S-transform-formula x = S-transform-formula (p · x)
  by transfer (simp)

lemma S-transform-formula-Conj [simp]:
  assumes finite (supp xset)
  shows S-transform-formula (Conj xset) = Conj_α (map-bset S-transform-formula xset)
  using assms by (simp add: Conj-def S-transform-formula-def bset.map-comp map-fun-def)

lemma S-transform-formula-Not [simp]: S-transform-formula (Not x) = Not_α (S-transform-formula x)
  by transfer simp

lemma S-transform-formula-Pred [simp]: S-transform-formula (Formula.Pred ϕ) = Act_α (S-action.Pred ϕ) (Conj_α bempty)
  by transfer simp
lemma S-transform-formula-Act [simp]: S-transform-formula $(\text{Formula}.\text{Act} \alpha x) = \text{Formula}.\text{Act}_\alpha (\text{S-action}.\text{Act} \alpha) (\text{S-transform-formula} x)$
  by transfer simp

lemma S-transform-formula-hereditarily-fs [simp]: hereditarily-fs $(\text{S-transform-formula} x)$
  by transfer (fact S-transform-Tree$_\alpha$-preserves-hereditarily-fs)

Finally, we define the proper S-transform, which returns formulas instead of trees.

definition S-transform :: $(\prime idx, \prime pred :: \text{fs}, \prime act :: \text{bn})$ formula $\Rightarrow (\prime idx)$ formula $\Rightarrow (\prime act, \prime pred)$ formula
  where
  $S$-transform $x = \text{Abs}$-formula $(\text{S-transform-formula} x)$

lemma S-transform-eqvt [eqvt]: $p \cdot S$-transform $x = S$-transform $(p \cdot x)$
  unfolding S-transform-def by simp

lemma finite-supp-map-bset-S-transform [simp]:
  assumes finite $(\text{supp} xset)$
  shows finite $(\text{supp} (\text{map-bset} S$-transform $xset))$
  proof –
    have eqvt map-bset and eqvt S-transform
      by (simp add: eqvtI)+
    then have supp $(\text{map-bset} S$-transform $xset) = \{}$
      using supp-fun-eqvt supp-fun-app-eqvt by blast
    then have supp $(\text{map-bset} S$-transform $xset) \subseteq \text{supp} xset$
      using supp-fun-app by blast
    with asms show finite $(\text{supp} (\text{map-bset} S$-transform $xset))$
      by (metis finite-subset)
  qed

lemma S-transform-Conj [simp]:
  assumes finite $(\text{supp} xset)$
  shows $S$-transform $(\text{Conj} xset) = \text{Conj} (\text{map-bset}$ $S$-transform $xset)$
  using asms unfolding S-transform-def by (simp, simp add: Conj-def bset.map-comp a-def)

lemma S-transform-Not [simp]: $S$-transform $(\text{Not} x) = \text{Not} (S$-transform $x)$
  unfolding S-transform-def by (simp add: Not.abs-eq eq-onp-same-args)

lemma S-transform-Pred [simp]: $S$-transform $(\text{Formula}.\text{Pred} \varphi) = \text{Formula}.\text{Act}$ $(\text{S-action}.\text{Pred} \varphi) (\text{Conj bempty})$

lemma S-transform-Act [simp]: $S$-transform $(\text{Formula}.\text{Act} \alpha x) = \text{Formula}.\text{Act}$ $(\text{S-action}.\text{Act} \alpha) (\text{S-transform} x)$
  unfolding S-transform-def by (simp, simp add: Formula.Act-def)
context nominal-ts

lemma valid-Conj-bempty [simp]: P ⊨ Conj bempty
by (simp add: bempty.rep-eq eqvtI supp-fun-eqvt)

notation S-satisfies (infix ⊨ₜ 70)

interpretation S-transform: nominal-ts (⊨ₜ) (→ₜ)
by unfold-locales (fact S-satisfies-eqvt, fact S-transition-eqvt)

notation S-transform.valid (infix |ₜ 70)

The S-transform preserves satisfaction of formulas in the following sense:

theorem valid-iff-valid-S-transform: shows P |ₜ x ←→ P |ₜ S S-transform x
proof (induct x arbitrary: P)
case (Conj xset)
then show ?case
by auto

next
case (Not x)
then show ?case
by simp

next
case (Pred ϕ)
let ?ϕ = Formula.Pred ϕ::(′idx, ′pred, (′act, ′pred) S-action) formula
have bn (S-action.Pred ϕ::(′act, ′pred) S-action) * P
by (simp add: fresh-star-def)
then show ?case
by (auto simp add: S-transform.valid-Act-fresh S-transition-Pred-iff)

next
case (Act α x)
show ?case
proof
assume P |ₜ Formula.Act α x
then obtain α' x' P' where eq: Formula.Act α x = Formula.Act α' x' and trans: P → ⟨α', P'⟩ and valid: P' |= x'
by (metis valid-Act)
from eq obtain p where p-x: p · x = x' and p-α: p · α = α'
by (metis Act-eq-iff-perm)

from valid have −p · P' |= x
using p-x by (metis valid-eqvt permute-minus-cancel(2))
then have −p · P' |=ₜ S-transform x
using Act.hyps(1) by metis
then have P' |=ₜ S-transform x'
by (metis (no-types, lifting) p-x S-transform.valid-eqvt S-transform-eqvt permute-minus-cancel(1))
with \( \text{eq and trans show } P \models_{S} S\text{-transform (Formula.Act } \alpha x) \)  
using \( S\text{-transform.valid-Act } S\text{-transition.Act by fastforce} \)

next  
assume \(*: P \models_{S} S\text{-transform (Formula.Act } \alpha x)\)  

— rename \( bn \alpha \) to avoid \( P \), without touching \( Formula.Act \alpha x \)  
obtain \( p \) where 1: \((p \cdot bn \alpha) \not\approx P \) and 2: \( \text{supp (Formula.Act } \alpha x) \not\approx p \)  
proof (rule at-set-avoiding2[| of \( bn \alpha P \) Formula.Act \( \alpha x \), THEN \( \text{exE[} \)])  
show \( \text{finite (} bn \alpha ) \) by (fact \( bn\text{-finite} \))  
next  
show \( \text{finite (} \text{supp } P \) by (fact \( \text{finite-supp} \))  
next  
show \( \text{finite (} \text{supp (Formula.Act } \alpha x) \) by (fact \( \text{finite-supp} \))  
next  
show \( bn \alpha \not\approx P \) by simp  
qed \( \text{metis} \)  
from 2 have \( \text{eq: Formula.Act } \alpha x = \text{Formula.Act (} p \cdot \alpha \) (p \cdot x) \)  
using \( \text{supp-perm-eq by fastforce} \)

with \(* \) have \( P \models_{S} \text{Formula.Act } (S\text{-action.Act } (p \cdot \alpha)) (S\text{-transform } (p \cdot x)) \)  
by simp  
with 1 obtain \( P' \) where \( \text{trans: } P \twoheadsrightarrow_{S} (S\text{-action.Act } (p \cdot \alpha),P') \) and \( \text{valid: } P' \models_{S} S\text{-transform } (p \cdot x) \)  
by (metis \( S\text{-transform.valid-Act-fresh } bn\text{-S-action.simps}(1) \) \( bn\text{-eqvt} \))

from \( \text{valid have } -p \cdot P' \models_{S} S\text{-transform x} \)  
by (metis \( \text{no-types, hide-lams} \) \( S\text{-transform.valid-eqvt } S\text{-transform-eqvt} \) \( \text{permute-minus-cancel}(1)) \)  
then have \( -p \cdot P' \models x \)  
using \( \text{Act.hyps}(1) \) by \( \text{metis} \)  
then have \( P' \models p \cdot x \)  
by (metis \( \text{permute-minus-cancel}(1) \) \( \text{valid-eqvt} \))

moreover from \( \text{trans have } P \twoheadsrightarrow (p \cdot \alpha,P') \)  
using \( \text{S-transition-Act-iff by blast} \)

ultimately show \( P \models \text{Formula.Act } \alpha x \)  
using \( \text{eq valid-Act by blast} \)  
qed  
qed

end

context indexed-nominal-ts
begin

The following (alternative) proof of the “\( \rightarrow \)” direction of theorem \( \text{nominal-ts.bisimilar} \)  
\( (\models_{S}) \rightarrow_{S} (?P \models_{S} ?Q = {?P \sim} {?Q} \) namely that bisimilarity in the \( S\text{-transform} \)  
implies bisimilarity in the original transition system, uses the fact that the
\(S\)-transform(ation) preserves satisfaction of formulas, together with the fact that bisimilarity (in the \(S\)-transform) implies logical equivalence, and equivalence (in the original transition system) implies bisimilarity. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system.

**interpretation** \(S\)-transform: indexed-nominal-ts \(\langle \tau_\langle S \rangle \rangle \rightarrow_\langle S \rangle \rangle \)

by unfold-locales (fact \(S\)-satisfies-eqvt, fact \(S\)-transition-eqvt, fact card-idx-perm, fact card-idx-state)

**notation** \(S\)-transform.bisimilar (infix \(\sim_\langle S \rangle\) 100)

**theorem** \(P \sim_\langle S \rangle Q \rightarrow P \sim_\langle S \rangle Q\)

**proof**

assume \(P \sim_\langle S \rangle Q\)

then have \(S\)-transform.logically-equivalent \(P Q\)

by (fact \(S\)-transform.bisimilarity-implies-equivalence)

with valid-iff-valid-\(S\)-transform have logically-equivalent \(P Q\)

using logically-equivalent-def \(S\)-transform.logically-equivalent-def by blast

then show \(P \sim_\langle S \rangle Q\)

by (fact equivalence-implies-bisimilarity)

qed

end

27.7 Translation of weak formulas into formulas without predicates

**context** indexed-weak-nominal-ts

begin

**notation** \(S\)-satisfies (infix \(\vdash_\langle S \rangle 70\))

**interpretation** \(S\)-transform: indexed-weak-nominal-ts Act \(\tau \langle \vdash_\langle S \rangle \rightarrow_\langle S \rangle \rangle \)


**notation** \(S\)-transform.valid (infix \(\models_\langle S \rangle 70\))

**notation** \(S\)-transform.weakly-bisimilar (infix \(\approx_\langle S \rangle 100\))

The \(S\)-transform of a weak formula is not necessarily a weak formula. However, the image of all weak formulas under the \(S\)-transform is adequate for weak bisimilarity.

**corollary** \(P \approx_\langle S \rangle Q \leftrightarrow (\forall x. \text{weak-formula } x \rightarrow P \models_\langle S \rangle \; \text{\(S\)-transform } x \leftarrow Q\)


For every weak formula, there is an equivalent weak formula over the \(S\)-
transform.

corollary
assumes weak-formula $x$
obtains $y$ where $S$-transform.weak-formula $y$ and $\forall P. P \models x \leftrightarrow P \models_S y$
proof
let $\mathcal{S} = \{ P. P \models x \}$
— $\{ P. P \models x \}$ is finitely supported
have $\text{supp } x$ supports $\mathcal{S}$
unfolding supports-def proof (clarify)
fix $a$ $b$
assume $a: a \notin \text{supp } x$ and $b: b \notin \text{supp } x$
{
fix $P$
from $a$ and $b$ have $(a \equiv b) \cdot x = x$
by (simp add: fresh-def swap-fresh-fresh)
then have $(a \equiv b) \cdot P \models x \leftrightarrow P \models x$
by (metis permute-swap-cancel valid-eqvt)
}

note $\ast = \text{this}$
show $(a \equiv b) \cdot \mathcal{S} = \mathcal{S}$
by auto (metis mem-Collect-eq mem-permute-iff permute-swap-cancel $\ast$, simp
add: Collect-eqvt permute-fun-def $\ast$)
qed
then have finite $(\text{supp } \mathcal{S})$
using finite-supp supports-finite by blast
— $\{ P. P \models x \}$ is closed under weak bisimilarity
moreover {
fix $P$ $Q$
assume $P \in \mathcal{S}$ and $P \equiv_S Q$
with (weak-formula $x$) have $Q \in \mathcal{S}$
using $S$-transform-weakly-bisimilar-iff weak-bisimilarity-implies-weak-equivalence
weakly-logically-equivalent-def by auto
}
ultimately show $\mathcal{S}$thesis
using $S$-transform.weak-expressive-completeness that by (metis (no-types,
lifting) mem-Collect-eq)
qed

end
References