Modal Logics for Nominal Transition Systems

Tjark Weber et al.

March 17, 2025

Abstract

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

Contents

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

1 Bounded Sets Equipped With a Permutation Action

Additional lemmas about bounded sets.

interpretation *bset-lifting*: *bset-lifting* ⟨*proof*⟩

lemma *Abs-bset-inverse′* [*simp*]:
assumes $|A| <o\ natLeq +c\ |UNIV :: 'k\ set|$
shows $set\text{-}bset\ (Abs\text{-}bset\ A :: 'a\ set['k]) = A$
⟨*proof*⟩

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

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

lift-definition $permute\text{-}bset :: perm \Rightarrow 'a\ set['b] \Rightarrow 'a\ set['b]$ **is**
permute
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

lemma *supp-Abs-bset* [*simp*]:
assumes $|A| <o\ natLeq +c\ |UNIV :: 'k\ set|$
shows $supp\ (Abs\text{-}bset\ A :: 'a::pt\ set['k]) = supp\ A$
⟨*proof*⟩

lemma *map-bset-permute*: $p \cdot B = map\text{-}bset\ (permute\ p)\ B$
⟨*proof*⟩

lemma *set-bset-eqvt* [*eqvt*]:
$p \cdot set\text{-}bset\ B = set\text{-}bset\ (p \cdot B)$
⟨*proof*⟩

lemma *map-bset-eqvt* [*eqvt*]:
$p \cdot$ *map-bset f B* $=$ *map-bset* $(p \cdot f)$ $(p \cdot B)$
⟨*proof*⟩

lemma *bempty-eqvt* [*eqvt*]: $p \cdot$ *bempty* $=$ *bempty*
⟨*proof*⟩

lemma *binsert-eqvt* [*eqvt*]: $p \cdot$ (*binsert x B*) $=$ *binsert* $(p \cdot x)$ $(p \cdot B)$
⟨*proof*⟩

lemma *bsingleton-eqvt* [*eqvt*]: $p \cdot$ *bsingleton x* $=$ *bsingleton* $(p \cdot x)$
⟨*proof*⟩

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

2 Lemmas about Well-Foundedness and Permutations

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

lemma *less-bool-rel-iff* [*simp*]:
$(a,b) \in$ *less-bool-rel* $\longleftrightarrow \neg\ a \wedge b$
⟨*proof*⟩

lemma *wf-less-bool-rel*: *wf less-bool-rel*
⟨*proof*⟩

2.1 Hull and well-foundedness

inductive-set *hull-rel* **where**
$(p \cdot x,\ x) \in$ *hull-rel*

lemma *hull-relp-reflp*: *reflp hull-relp*
⟨*proof*⟩

lemma *hull-relp-symp*: *symp hull-relp*
⟨*proof*⟩

lemma *hull-relp-transp*: *transp hull-relp*
⟨*proof*⟩

lemma *hull-relp-equivp*: *equivp hull-relp*

⟨*proof*⟩

lemma *hull-rel-relcomp-subset*:
  assumes *eqvt R*
  shows $R\ O\ \mathit{hull\text{-}rel} \subseteq \mathit{hull\text{-}rel}\ O\ R$
⟨*proof*⟩

lemma *wf-hull-rel-relcomp*:
  assumes *wf R* **and** *eqvt R*
  shows *wf* (*hull-rel O R*)
⟨*proof*⟩

lemma *hull-rel-relcompI* [*simp*]:
  assumes $(x, y) \in R$
  shows $(p \cdot x, y) \in \mathit{hull\text{-}rel}\ O\ R$
⟨*proof*⟩

lemma *hull-rel-relcomp-trivialI* [*simp*]:
  assumes $(x, y) \in R$
  shows $(x, y) \in \mathit{hull\text{-}rel}\ O\ R$
⟨*proof*⟩

end
theory *Residual*
imports
  Nominal2.Nominal2
begin

3 Residuals

3.1 Binding names

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

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

lemma *bn-subset-supp*: $bn\ \alpha \subseteq supp\ \alpha$
⟨*proof*⟩

3.2 Raw residuals and α-equivalence

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

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

α-equivalence is equivariant.

lemma *alpha-residual-eqvt* [*eqvt*]:
assumes *alpha-residual r1 r2*
shows *alpha-residual* (*p* · *r1*) (*p* · *r2*)
⟨*proof*⟩

α-equivalence is an equivalence relation.

lemma *alpha-residual-reflp*: *reflp alpha-residual*
⟨*proof*⟩

lemma *alpha-residual-symp*: *symp alpha-residual*
⟨*proof*⟩

lemma *alpha-residual-transp*: *transp alpha-residual*
⟨*proof*⟩

lemma *alpha-residual-equivp*: *equivp alpha-residual*
⟨*proof*⟩

3.3 Residuals

Residuals are raw residuals quotiented by α-equivalence.

quotient-type
(*'act*,*'state*) *residual* = *'act*::*bn* × *'state*::*pt* / *alpha-residual*
⟨*proof*⟩

lemma *residual-abs-rep* [*simp*]: *abs-residual* (*rep-residual res*) = *res*
⟨*proof*⟩

lemma *residual-rep-abs* [*simp*]: *alpha-residual* (*rep-residual* (*abs-residual r*)) *r*
⟨*proof*⟩

The permutation operation is lifted from raw residuals.

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

lift-definition *permute-residual* :: *perm* ⇒ (*'a*,*'b*) *residual* ⇒ (*'a*,*'b*) *residual*
is *permute*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

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

lemma *alpha-residual-permute-rep-commute* [*simp*]: *alpha-residual* (*p* • *rep-residual* *res*) (*rep-residual* (*p* • *res*))
⟨*proof*⟩

3.4 Notation for pairs as residuals

abbreviation *abs-residual-pair* :: *'act*::*bn* ⇒ *'state*::*pt* ⇒ (*'act*,*'state*) *residual* (‹⟨-,-⟩› [*0*,*0*] *1000*)
where
⟨α,*P*⟩ == *abs-residual* (α,*P*)

lemma *abs-residual-pair-eqvt* [*simp*]: *p* • ⟨α,*P*⟩ = ⟨*p* • α, *p* • *P*⟩
⟨*proof*⟩

3.5 Support of residuals

We only consider finitely supported states now.

lemma *supp-abs-residual-pair*: *supp* ⟨α, *P*::*'state*::*fs*⟩ = *supp* (α,*P*) − *bn* α
⟨*proof*⟩

lemma *bn-abs-residual-fresh* [*simp*]: *bn* α ♯∗ ⟨α,*P*::*'state*::*fs*⟩
⟨*proof*⟩

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

3.6 Equality between residuals

lemma *residual-eq-iff-perm*: ⟨α*1*,*P1*⟩ = ⟨α*2*,*P2*⟩ ⟷
(∃ *p*. *supp* (α*1*, *P1*) − *bn* α*1* = *supp* (α*2*, *P2*) − *bn* α*2* ∧ (*supp* (α*1*, *P1*) − *bn* α*1*) ♯∗ *p* ∧ *p* • (α*1*, *P1*) = (α*2*, *P2*) ∧ *p* • *bn* α*1* = *bn* α*2*)
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *residual-eq-iff-perm-renaming*: ⟨α*1*,*P1*⟩ = ⟨α*2*,*P2*⟩ ⟷
(∃ *p*. *supp* (α*1*, *P1*) − *bn* α*1* = *supp* (α*2*, *P2*) − *bn* α*2* ∧ (*supp* (α*1*, *P1*) − *bn* α*1*) ♯∗ *p* ∧ *p* • (α*1*, *P1*) = (α*2*, *P2*) ∧ *p* • *bn* α*1* = *bn* α*2* ∧ *supp* *p* ⊆ *bn* α*1* ∪ *p* • *bn* α*1*)
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

3.7 Strong induction

lemma *residual-strong-induct*:
assumes ⋀(*act*::*'act*::*bn*) (*state*::*'state*::*fs*) (*c*::*'a*::*fs*). *bn act* ♯∗ *c* ⟹ *P c* ⟨*act*,*state*⟩

shows *P c residual*
⟨*proof*⟩

3.8 Other lemmas

lemma *residual-empty-bn-eq-iff*:
  assumes *bn* $\alpha 1 = \{\}$
  shows $\langle \alpha 1, P1\rangle = \langle \alpha 2, P2\rangle \longleftrightarrow \alpha 1 = \alpha 2 \wedge P1 = P2$
⟨*proof*⟩
lemma *set-bounded-supp*:
  assumes *finite S* **and** $\bigwedge x.\ x \in X \Longrightarrow supp\ x \subseteq S$
  shows $supp\ X \subseteq S$
⟨*proof*⟩

end
theory *Transition-System*
imports
  Residual
begin

4 Nominal Transition Systems and Bisimulations

4.1 Basic Lemmas

lemma *symp-on-eqvt* [*eqvt*]:
  assumes *symp-on A R* **shows** *symp-on* $(p \cdot A)$ $(p \cdot R)$
  ⟨*proof*⟩

lemma *symp-eqvt*:
  assumes *symp R* **shows** *symp* $(p \cdot R)$
  ⟨*proof*⟩

4.2 Nominal transition systems

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

  lemma *transition-eqvt'*:
    assumes $P \to \langle \alpha, Q\rangle$ **shows** $p \cdot P \to \langle p \cdot \alpha,\ p \cdot Q\rangle$
  ⟨*proof*⟩

end

4.3 Bisimulations

context *nominal-ts*

begin

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

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

($\sim\cdot$) is an equivariant equivalence relation.

lemma *is-bisimulation-eqvt* :
assumes *is-bisimulation R* **shows** *is-bisimulation* ($p \cdot R$)
⟨*proof*⟩

lemma *bisimilar-eqvt* :
assumes $P \sim\cdot\ Q$ **shows** $(p \cdot P) \sim\cdot\ (p \cdot Q)$
⟨*proof*⟩

lemma *bisimilar-reflp*: *reflp bisimilar*
⟨*proof*⟩

lemma *bisimilar-symp*: *symp bisimilar*
⟨*proof*⟩

lemma *bisimilar-is-bisimulation*: *is-bisimulation bisimilar*
⟨*proof*⟩

lemma *bisimilar-transp*: *transp bisimilar*
⟨*proof*⟩

lemma *bisimilar-equivp*: *equivp bisimilar*
⟨*proof*⟩

lemma *bisimilar-simulation-step*:
assumes $P \sim\cdot\ Q$ **and** $bn\ \alpha\ \sharp*\ Q$ **and** $P \rightarrow \langle\alpha,P'\rangle$
obtains Q' **where** $Q \rightarrow \langle\alpha,Q'\rangle$ **and** $P' \sim\cdot\ Q'$
⟨*proof*⟩

end

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

begin

5 Infinitary Formulas

5.1 Infinitely branching trees

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

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

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

inductive-set *Tree-wf* :: (*'idx*,*'pred*,*'act*) *Tree rel* **where**
  $t \in$ *set-bset tset* $\Longrightarrow$ (t, *tConj tset*) $\in$ *Tree-wf*
| (t, *tNot* t) $\in$ *Tree-wf*
| (t, *tAct* α t) $\in$ *Tree-wf*

lemma *wf-Tree-wf*: *wf Tree-wf*
⟨*proof*⟩

We define a permutation operation on the type of trees.

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

  primrec *permute-Tree* :: *perm* ⇒ (-,-,-) *Tree* ⇒ (-,-,-) *Tree* **where**
    $p \cdot$ (*tConj tset*) = *tConj* (*map-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* φ) = *tPred* ($p \cdot \varphi$)
  | $p \cdot$ (*tAct* α t) = *tAct* ($p \cdot \alpha$) ($p \cdot t$)

  instance
  ⟨*proof*⟩

end

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

lemma *permute-Tree-tConj* [*simp*]: $p \cdot tConj\ tset = tConj\ (p \cdot tset)$
⟨*proof*⟩

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

The relation *Tree-wf* is equivariant.

lemma *Tree-wf-eqvt-aux*:
assumes $(t1, t2) \in Tree\text{-}wf$ **shows** $(p \cdot t1, p \cdot t2) \in Tree\text{-}wf$
⟨*proof*⟩

lemma *Tree-wf-eqvt* [*eqvt*, *simp*]: $p \cdot Tree\text{-}wf = Tree\text{-}wf$
⟨*proof*⟩

lemma *Tree-wf-eqvt′*: *eqvt Tree-wf*
⟨*proof*⟩

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*]: $supp\ (tConj\ tset) = supp\ tset$
⟨*proof*⟩

lemma *supp-tNot* [*simp*]: $supp\ (tNot\ t) = supp\ t$
⟨*proof*⟩

lemma *supp-tPred* [*simp*]: $supp\ (tPred\ \varphi) = supp\ \varphi$
⟨*proof*⟩

lemma *supp-tAct* [*simp*]: $supp\ (tAct\ \alpha\ t) = supp\ \alpha \cup supp\ t$
⟨*proof*⟩

5.2 Trees modulo α-equivalence

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

lemma *supp-rel-eqvt* [*eqvt*]:
$p \cdot supp\text{-}rel\ R\ x = supp\text{-}rel\ (p \cdot R)\ (p \cdot x)$
⟨*proof*⟩

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

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

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

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

lemma *supp-rel-cong* [*fundef-cong*]:
$\llbracket\ x{=}x';\ \bigwedge a\ b.\ R\ ((a \rightleftharpoons b) \cdot x')\ x' \longleftrightarrow R'\ ((a \rightleftharpoons b) \cdot x')\ x'\ \rrbracket \Longrightarrow \textit{supp-rel}\ R\ x = \textit{supp-rel}\ R'\ x'$
⟨*proof*⟩

lemma *rel-bset-cong* [*fundef-cong*]:
$\llbracket\ x{=}x';\ y{=}y';\ \bigwedge a\ b.\ a \in \textit{set-bset}\ x' \Longrightarrow b \in \textit{set-bset}\ y' \Longrightarrow R\ a\ b \longleftrightarrow R'\ a\ b\ \rrbracket \Longrightarrow \textit{rel-bset}\ R\ x\ y \longleftrightarrow \textit{rel-bset}\ R'\ x'\ y'$
⟨*proof*⟩

lemma *alpha-set-cong* [*fundef-cong*]:
$\llbracket\ bs{=}bs';\ x{=}x';\ R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y';\ f\ x' = f'\ x';\ f\ y' = f'\ y';\ p{=}p';\ cs{=}cs';\ y{=}y'\ \rrbracket \Longrightarrow$
$\textit{alpha-set}\ (bs,\ x)\ R\ f\ p\ (cs,\ y) \longleftrightarrow \textit{alpha-set}\ (bs',\ x')\ R'\ f'\ p'\ (cs',\ y')$
⟨*proof*⟩

quotient-type
$('idx,'pred,'act)\ \textit{Tree}_p = ('idx,'pred{::}pt,'act{::}bn)\ \textit{Tree}\ /\ \textit{hull-relp}$
⟨*proof*⟩

lemma $\textit{abs-Tree}_p\textit{-eq}$ [*simp*]: $\textit{abs-Tree}_p\ (p \cdot t) = \textit{abs-Tree}_p\ t$
⟨*proof*⟩

lemma $\textit{permute-rep-abs-Tree}_p$:
obtains p **where** $\textit{rep-Tree}_p\ (\textit{abs-Tree}_p\ t) = p \cdot t$
⟨*proof*⟩

lift-definition $\textit{Tree-wf}_p :: ('idx,'pred{::}pt,'act{::}bn)\ \textit{Tree}_p\ \textit{rel}$ **is**
Tree-wf ⟨*proof*⟩

lemma $\textit{Tree-wf}_p I$ [*simp*]:
assumes $(a,\ b) \in \textit{Tree-wf}$
shows $(\textit{abs-Tree}_p\ (p \cdot a),\ \textit{abs-Tree}_p\ b) \in \textit{Tree-wf}_p$
⟨*proof*⟩

lemma $\textit{Tree-wf}_p\textit{-trivialI}$ [*simp*]:
assumes $(a,\ b) \in \textit{Tree-wf}$
shows $(\textit{abs-Tree}_p\ a,\ \textit{abs-Tree}_p\ b) \in \textit{Tree-wf}_p$
⟨*proof*⟩

lemma *Tree-wf$_p$E*:
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*
⟨*proof*⟩

lemma *wf-Tree-wf$_p$*: *wf Tree-wf$_p$*
⟨*proof*⟩

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

Here it comes ...

function (*sequential*)
alpha-Tree :: ('*idx*,'*pred*::*pt*,'*act*::*bn*) *Tree* ⇒ ('*idx*,'*pred*,'*act*) *Tree* ⇒ *bool* (**infix** ‹$=_\alpha$› *50*) **where**
— ($=_\alpha$)
alpha-tConj: *tConj tset1* $=_\alpha$ *tConj tset2* ⟷ *rel-bset alpha-Tree tset1 tset2*
| *alpha-tNot*: *tNot t1* $=_\alpha$ *tNot t2* ⟷ *t1* $=_\alpha$ *t2*
| *alpha-tPred*: *tPred* $\varphi 1$ $=_\alpha$ *tPred* $\varphi 2$ ⟷ $\varphi 1 = \varphi 2$
— the action may have binding names
| *alpha-tAct*: *tAct* $\alpha 1$ *t1* $=_\alpha$ *tAct* $\alpha 2$ *t2* ⟷
(∃ *p*. (*bn* $\alpha 1$, *t1*) ≈*set alpha-Tree* (*supp-rel alpha-Tree*) *p* (*bn* $\alpha 2$, *t2*) ∧ (*bn* $\alpha 1$, $\alpha 1$) ≈*set* ((=)) *supp p* (*bn* $\alpha 2$, $\alpha 2$))
| *alpha-other*: - $=_\alpha$ - ⟷ *False*
— 254 subgoals (!)
⟨*proof*⟩
termination
⟨*proof*⟩

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

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

lemma *alpha-Tree-induct*[*case-names tConj tNot tPred tAct*, *consumes 1*]:
assumes *t1* $=_\alpha$ *t2*
and ⋀*tset1 tset2*. (⋀*a b*. *a* ∈ *set-bset tset1* ⟹ *b* ∈ *set-bset tset2* ⟹ *a* $=_\alpha$ *b* ⟹ *P a b*) ⟹
rel-bset ($=_\alpha$) *tset1 tset2* ⟹ *P* (*tConj tset1*) (*tConj tset2*)
and ⋀*t1 t2*. *t1* $=_\alpha$ *t2* ⟹ *P t1 t2* ⟹ *P* (*tNot t1*) (*tNot t2*)
and ⋀φ. *P* (*tPred* φ) (*tPred* φ)
and ⋀$\alpha 1$ *t1* $\alpha 2$ *t2*. (⋀*p*. *p* · *t1* $=_\alpha$ *t2* ⟹ *P* (*p* · *t1*) *t2*) ⟹
(⋀*a b*. ((*a* ⇌ *b*) · *t1*) $=_\alpha$ *t1* ⟹ *P* ((*a* ⇌ *b*) · *t1*) *t1*) ⟹ (⋀*a b*. ((*a* ⇌ *b*) · *t2*) $=_\alpha$ *t2* ⟹ *P* ((*a* ⇌ *b*) · *t2*) *t2*) ⟹

$(\exists p.\ (bn\ \alpha 1,\ t1) \approx set\ (=_\alpha)\ (supp\text{-}rel\ (=_\alpha))\ p\ (bn\ \alpha 2,\ t2) \wedge (bn\ \alpha 1,\ \alpha 1)$
$\approx set\ (=)\ supp\ p\ (bn\ \alpha 2,\ \alpha 2)) \Longrightarrow$
$P\ (tAct\ \alpha 1\ t1)\ (tAct\ \alpha 2\ t2)$
shows $P\ t1\ t2$
⟨*proof*⟩

α-equivalence is equivariant.

lemma *alpha-Tree-eqvt-aux*:
assumes $\bigwedge a\ b.\ (a \rightleftharpoons b) \cdot t =_\alpha t \longleftrightarrow p \cdot (a \rightleftharpoons b) \cdot t =_\alpha p \cdot t$
shows $p \cdot supp\text{-}rel\ (=_\alpha)\ t = supp\text{-}rel\ (=_\alpha)\ (p \cdot t)$
⟨*proof*⟩

lemma *alpha-Tree-eqvt'*: $t1 =_\alpha t2 \longleftrightarrow p \cdot t1 =_\alpha p \cdot t2$
⟨*proof*⟩

lemma *alpha-Tree-eqvt* [*eqvt*]: $t1 =_\alpha t2 \Longrightarrow p \cdot t1 =_\alpha p \cdot t2$
⟨*proof*⟩

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

lemma *alpha-Tree-reflp*: *reflp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-symp*: *symp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-transp*: *transp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-equivp*: *equivp alpha-Tree*
⟨*proof*⟩

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

lemma *alpha-Tree-supp-rel*:
assumes $t1 =_\alpha t2$
shows $supp\text{-}rel\ (=_\alpha)\ t1 = supp\text{-}rel\ (=_\alpha)\ t2$
⟨*proof*⟩

tAct preserves α-equivalence.

lemma *alpha-Tree-tAct*:
assumes $t1 =_\alpha t2$
shows $tAct\ \alpha\ t1 =_\alpha tAct\ \alpha\ t2$
⟨*proof*⟩

The following lemmas describe the support modulo *alpha*-equivalence.

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

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

$\langle proof \rangle$

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

lemma *infinite-mono*: $infinite\ S \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow x \in T) \Longrightarrow infinite\ T$
$\langle proof \rangle$

lemma *supp-rel-tAct* [*simp*]:
  assumes $finite\ (supp\text{-}rel\ (=_\alpha)\ t)$
  shows $supp\text{-}rel\ (=_\alpha)\ (tAct\ \alpha\ t) = supp\ \alpha \cup supp\text{-}rel\ (=_\alpha)\ t - bn\ \alpha$
$\langle proof \rangle$

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

quotient-type
  $('idx,'pred,'act)\ Tree_\alpha = ('idx,'pred{::}pt,'act{::}bn)\ Tree\ /\ alpha\text{-}Tree$
  $\langle proof \rangle$

lemma $Tree_\alpha$*-abs-rep* [*simp*]: $abs\text{-}Tree_\alpha\ (rep\text{-}Tree_\alpha\ t_\alpha) = t_\alpha$
$\langle proof \rangle$

lemma $Tree_\alpha$*-rep-abs* [*simp*]: $rep\text{-}Tree_\alpha\ (abs\text{-}Tree_\alpha\ t) =_\alpha t$
$\langle proof \rangle$

The permutation operation is lifted from trees.

instantiation $Tree_\alpha$ :: (*type*, *pt*, *bn*) *pt*
begin

  lift-definition $permute\text{-}Tree_\alpha :: perm \Rightarrow ('a,'b,'c)\ Tree_\alpha \Rightarrow ('a,'b,'c)\ Tree_\alpha$
    is *permute*
  $\langle proof \rangle$

  instance
  $\langle proof \rangle$

end

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

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

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

5.3 Constructors for trees modulo α-equivalence

The constructors are lifted from trees.

lift-definition $Conj_\alpha$ :: $('idx,'pred,'act)$ $Tree_\alpha$ $set['idx] \Rightarrow ('idx,'pred::pt,'act::bn)$ $Tree_\alpha$ **is**
tConj
⟨*proof*⟩

lemma *map-bset-abs-rep-Tree*$_\alpha$: *map-bset abs-Tree*$_\alpha$ (*map-bset rep-Tree*$_\alpha$ $tset_\alpha$) = $tset_\alpha$
⟨*proof*⟩

lemma $Conj_\alpha$*-def'*: $Conj_\alpha$ $tset_\alpha$ = *abs-Tree*$_\alpha$ (*tConj* (*map-bset rep-Tree*$_\alpha$ $tset_\alpha$))
⟨*proof*⟩

lift-definition Not_α :: $('idx,'pred,'act)$ $Tree_\alpha \Rightarrow ('idx,'pred::pt,'act::bn)$ $Tree_\alpha$ **is**
tNot
⟨*proof*⟩

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

lift-definition Act_α :: $'act \Rightarrow ('idx,'pred,'act)$ $Tree_\alpha \Rightarrow ('idx,'pred::pt,'act::bn)$ $Tree_\alpha$ **is**
tAct
⟨*proof*⟩

The lifted constructors are equivariant.

lemma $Conj_\alpha$*-eqvt* [*eqvt*, *simp*]: $p \cdot Conj_\alpha\ tset_\alpha = Conj_\alpha\ (p \cdot tset_\alpha)$
⟨*proof*⟩

lemma Not_α*-eqvt* [*eqvt*, *simp*]: $p \cdot Not_\alpha\ t_\alpha = Not_\alpha\ (p \cdot t_\alpha)$
⟨*proof*⟩

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

lemma Act_α*-eqvt* [*eqvt*, *simp*]: $p \cdot Act_\alpha\ \alpha\ t_\alpha = Act_\alpha\ (p \cdot \alpha)\ (p \cdot t_\alpha)$
⟨*proof*⟩

The lifted constructors are injective (except for Act_α).

lemma $Conj_\alpha$*-eq-iff* [*simp*]: $Conj_\alpha\ tset1_\alpha = Conj_\alpha\ tset2_\alpha \longleftrightarrow tset1_\alpha = tset2_\alpha$
⟨*proof*⟩

lemma Not_α*-eq-iff* [*simp*]: $Not_\alpha\ t1_\alpha = Not_\alpha\ t2_\alpha \longleftrightarrow t1_\alpha = t2_\alpha$
⟨*proof*⟩

lemma $Pred_\alpha$*-eq-iff* [*simp*]: $Pred_\alpha\ \varphi 1 = Pred_\alpha\ \varphi 2 \longleftrightarrow \varphi 1 = \varphi 2$
⟨*proof*⟩

lemma Act_α*-eq-iff*: $Act_\alpha\ \alpha 1\ t1 = Act_\alpha\ \alpha 2\ t2 \longleftrightarrow tAct\ \alpha 1\ (rep\text{-}Tree_\alpha\ t1) =_\alpha$

tAct α*2* (*rep-Tree*$_\alpha$ *t2*)
⟨*proof*⟩

The lifted constructors are free (except for Act_α).

lemma *Tree*$_\alpha$*-free* [*simp*]:
shows $Conj_\alpha\ tset_\alpha \neq Not_\alpha\ t_\alpha$
and $Conj_\alpha\ tset_\alpha \neq Pred_\alpha\ \varphi$
and $Conj_\alpha\ tset_\alpha \neq Act_\alpha\ \alpha\ t_\alpha$
and $Not_\alpha\ t_\alpha \neq Pred_\alpha\ \varphi$
and $Not_\alpha\ t1_\alpha \neq Act_\alpha\ \alpha\ t2_\alpha$
and $Pred_\alpha\ \varphi \neq Act_\alpha\ \alpha\ t_\alpha$
⟨*proof*⟩

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

lemma *supp-alpha-supp-rel*: $supp\ t_\alpha = supp\text{-}rel\ (=_\alpha)\ (rep\text{-}Tree_\alpha\ t_\alpha)$
⟨*proof*⟩

lemma *supp-Conj*$_\alpha$ [*simp*]: $supp\ (Conj_\alpha\ tset_\alpha) = supp\ tset_\alpha$
⟨*proof*⟩

lemma *supp-Not*$_\alpha$ [*simp*]: $supp\ (Not_\alpha\ t_\alpha) = supp\ t_\alpha$
⟨*proof*⟩

lemma *supp-Pred*$_\alpha$ [*simp*]: $supp\ (Pred_\alpha\ \varphi) = supp\ \varphi$
⟨*proof*⟩

lemma *supp-Act*$_\alpha$ [*simp*]:
assumes $finite\ (supp\ t_\alpha)$
shows $supp\ (Act_\alpha\ \alpha\ t_\alpha) = supp\ \alpha \cup supp\ t_\alpha - bn\ \alpha$
⟨*proof*⟩

5.4 Induction over trees modulo α-equivalence

lemma *Tree*$_\alpha$*-induct* [*case-names* $Conj_\alpha$ Not_α $Pred_\alpha$ Act_α Env_α, *induct type*: $Tree_\alpha$]:
fixes t_α
assumes $\bigwedge tset_\alpha.\ (\bigwedge x.\ x \in set\text{-}bset\ tset_\alpha \Longrightarrow P\ x) \Longrightarrow P\ (Conj_\alpha\ tset_\alpha)$
and $\bigwedge t_\alpha.\ P\ t_\alpha \Longrightarrow P\ (Not_\alpha\ t_\alpha)$
and $\bigwedge pred.\ P\ (Pred_\alpha\ pred)$
and $\bigwedge act\ t_\alpha.\ P\ t_\alpha \Longrightarrow P\ (Act_\alpha\ act\ t_\alpha)$
shows $P\ t_\alpha$
⟨*proof*⟩

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

5.5 Hereditarily finitely supported trees

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

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

inductive *hereditarily-fs* :: $('idx,'pred::fs,'act::bn)$ $Tree_\alpha \Rightarrow bool$ **where**
  $Conj_\alpha$: $finite\ (supp\ tset_\alpha) \Longrightarrow (\bigwedge t_\alpha.\ t_\alpha \in set\text{-}bset\ tset_\alpha \Longrightarrow hereditarily\text{-}fs\ t_\alpha)$
$\Longrightarrow hereditarily\text{-}fs\ (Conj_\alpha\ tset_\alpha)$
| Not_α: $hereditarily\text{-}fs\ t_\alpha \Longrightarrow hereditarily\text{-}fs\ (Not_\alpha\ t_\alpha)$
| $Pred_\alpha$: $hereditarily\text{-}fs\ (Pred_\alpha\ \varphi)$
| Act_α: $hereditarily\text{-}fs\ t_\alpha \Longrightarrow hereditarily\text{-}fs\ (Act_\alpha\ \alpha\ t_\alpha)$

hereditarily-fs is equivariant.

lemma *hereditarily-fs-eqvt* [*eqvt*]:
  assumes $hereditarily\text{-}fs\ t_\alpha$
  shows $hereditarily\text{-}fs\ (p \cdot t_\alpha)$
⟨*proof*⟩

hereditarily-fs is preserved under α-renaming.

lemma *hereditarily-fs-alpha-renaming*:
  assumes $Act_\alpha\ \alpha\ t_\alpha = Act_\alpha\ \alpha'\ t_\alpha'$
  shows $hereditarily\text{-}fs\ t_\alpha \longleftrightarrow hereditarily\text{-}fs\ t_\alpha'$
⟨*proof*⟩

Hereditarily finitely supported trees have finite support.

lemma *hereditarily-fs-implies-finite-supp*:
  assumes $hereditarily\text{-}fs\ t_\alpha$
  shows $finite\ (supp\ t_\alpha)$
⟨*proof*⟩

5.6 Infinitary formulas

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

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

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
⟨*proof*⟩

lemma *Rep-formula′* [*simp*]: *hereditarily-fs* (*Rep-formula* *x*)
⟨*proof*⟩

Now we lift the permutation operation.

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

  lift-definition *permute-formula* :: *perm* ⇒ (′*a*,′*b*,′*c*) *formula* ⇒ (′*a*,′*b*,′*c*) *formula*
    is *permute*
  ⟨*proof*⟩

  instance
  ⟨*proof*⟩

end

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

lemma *Abs-formula-eqvt* [*simp*]:
  assumes *hereditarily-fs* t_α
  shows $p \cdot$ *Abs-formula* t_α = *Abs-formula* ($p \cdot t_\alpha$)
⟨*proof*⟩

lemma *supp-Abs-formula* [*simp*]:
  assumes *hereditarily-fs* t_α
  shows *supp* (*Abs-formula* t_α) = *supp* t_α
⟨*proof*⟩

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

lemma *supp-Rep-formula* [*simp*]: *supp* (*Rep-formula* *x*) = *supp* *x*
⟨*proof*⟩

lemma *supp-map-bset-Rep-formula* [*simp*]: *supp* (*map-bset* *Rep-formula* *xset*) = *supp* *xset*
⟨*proof*⟩

Formulas are in fact finitely supported.

instance *formula* :: (*type*, *fs*, *bn*) *fs*
⟨*proof*⟩

5.7 Constructors for infinitary formulas

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

definition *Conj* :: ('*idx*,'*pred*,'*act*) *formula set*['*idx*] ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* **where**
Conj xset = *Abs-formula* ($Conj_\alpha$ (*map-bset Rep-formula xset*))

lemma *finite-supp-implies-hereditarily-fs-*$Conj_\alpha$ [*simp*]:
assumes *finite* (*supp xset*)
shows *hereditarily-fs* ($Conj_\alpha$ (*map-bset Rep-formula xset*))
⟨*proof*⟩

lemma *Conj-rep-eq*:
assumes *finite* (*supp xset*)
shows *Rep-formula* (*Conj xset*) = $Conj_\alpha$ (*map-bset Rep-formula xset*)
⟨*proof*⟩

lift-definition *Not* :: ('*idx*,'*pred*,'*act*) *formula* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* **is**
Not_α
⟨*proof*⟩

lift-definition *Pred* :: '*pred* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* **is**
$Pred_\alpha$
⟨*proof*⟩

lift-definition *Act* :: '*act* ⇒ ('*idx*,'*pred*,'*act*) *formula* ⇒ ('*idx*,'*pred*::*fs*,'*act*::*bn*) *formula* **is**
Act_α
⟨*proof*⟩

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

lemma *Conj-eqvt* [*simp*]:
assumes *finite* (*supp xset*)
shows $p \cdot$ *Conj xset* = *Conj* ($p \cdot$ *xset*)
⟨*proof*⟩

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

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

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

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows *supp* (*Conj xset*) = *supp xset*
⟨*proof*⟩

lemma *supp-Not* [*simp*]: *supp* (*Not x*) = *supp x*
⟨*proof*⟩

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

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

lemma *bn-fresh-Act* [*simp*]: $bn\ \alpha \sharp* Act\ \alpha\ x$
⟨*proof*⟩

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

lemma *Conj-eq-iff* [*simp*]:
assumes *finite* (*supp xset1*) **and** *finite* (*supp xset2*)
shows $Conj\ xset1 = Conj\ xset2 \longleftrightarrow xset1 = xset2$
⟨*proof*⟩

lemma *Not-eq-iff* [*simp*]: $Not\ x1 = Not\ x2 \longleftrightarrow x1 = x2$
⟨*proof*⟩

lemma *Pred-eq-iff* [*simp*]: $Pred\ \varphi 1 = Pred\ \varphi 2 \longleftrightarrow \varphi 1 = \varphi 2$
⟨*proof*⟩

lemma *Act-eq-iff*: $Act\ \alpha 1\ x1 = Act\ \alpha 2\ x2 \longleftrightarrow Act_\alpha\ \alpha 1\ (Rep\text{-}formula\ x1) = Act_\alpha\ \alpha 2\ (Rep\text{-}formula\ x2)$
⟨*proof*⟩

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: $Act\ \alpha 1\ x1 = Act\ \alpha 2\ x2 \longleftrightarrow$
$(\exists p.\ supp\ x1 - bn\ \alpha 1 = supp\ x2 - bn\ \alpha 2 \wedge (supp\ x1 - bn\ \alpha 1) \sharp* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha 1 - bn\ \alpha 1 = supp\ \alpha 2 - bn\ \alpha 2 \wedge (supp\ \alpha 1 - bn\ \alpha 1) \sharp* p \wedge p \cdot \alpha 1 = \alpha 2)$
(**is** $?l \longleftrightarrow ?r$)
⟨*proof*⟩

lemma *Act-eq-iff-perm-renaming*: $Act\ \alpha 1\ x1 = Act\ \alpha 2\ x2 \longleftrightarrow$
$(\exists p.\ supp\ x1 - bn\ \alpha 1 = supp\ x2 - bn\ \alpha 2 \wedge (supp\ x1 - bn\ \alpha 1) \sharp* p \wedge p \cdot x1 = x2 \wedge supp\ \alpha 1 - bn\ \alpha 1 = supp\ \alpha 2 - bn\ \alpha 2 \wedge (supp\ \alpha 1 - bn\ \alpha 1) \sharp* p \wedge p \cdot$

$\alpha 1 = \alpha 2 \wedge supp\ p \subseteq bn\ \alpha 1 \cup p \cdot bn\ \alpha 1$)
(**is** $?l \longleftrightarrow ?r$)
⟨*proof*⟩

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

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

5.8 Induction over infinitary formulas

lemma *formula-induct* [*case-names Conj Not Pred Act*, *induct type*: *formula*]:
fixes *x*
assumes $\bigwedge xset.\ finite\ (supp\ xset) \Longrightarrow (\bigwedge x.\ x \in set\text{-}bset\ xset \Longrightarrow P\ x) \Longrightarrow P\ (Conj\ xset)$
and $\bigwedge formula.\ P\ formula \Longrightarrow P\ (Not\ formula)$
and $\bigwedge pred.\ P\ (Pred\ pred)$
and $\bigwedge act\ formula.\ P\ formula \Longrightarrow P\ (Act\ act\ formula)$
shows *P x*
⟨*proof*⟩

5.9 Strong induction over infinitary formulas

lemma *formula-strong-induct-aux*:
fixes *x*
assumes $\bigwedge xset\ c.\ finite\ (supp\ xset) \Longrightarrow (\bigwedge x.\ x \in set\text{-}bset\ xset \Longrightarrow (\bigwedge c.\ P\ c\ x)) \Longrightarrow P\ c\ (Conj\ xset)$
and $\bigwedge formula\ c.\ (\bigwedge c.\ P\ c\ formula) \Longrightarrow P\ c\ (Not\ formula)$
and $\bigwedge pred\ c.\ P\ c\ (Pred\ pred)$
and $\bigwedge act\ formula\ c.\ bn\ act\ \sharp\!*\ c \Longrightarrow (\bigwedge c.\ P\ c\ formula) \Longrightarrow P\ c\ (Act\ act\ formula)$
shows $\bigwedge (c :: 'd::fs)\ p.\ P\ c\ (p \cdot x)$
⟨*proof*⟩

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 ≡ $\{(x,y).\ x =_\alpha y\}$

lemma *alpha-Tree-relI* [*simp*]:
assumes $x =_\alpha y$ **shows** $(x,y) \in$ *alpha-Tree-rel*
⟨*proof*⟩

lemma *alpha-Tree-relE*:
assumes $(x,y) \in$ *alpha-Tree-rel* **and** $x =_\alpha y \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
wf (*alpha-Tree-rel O hull-rel O Tree-wf*)
⟨*proof*⟩

lemma *alpha-Tree-rel-relcomp-trivialI* [*simp*]:
assumes $(x, y) \in R$
shows $(x, y) \in$ *alpha-Tree-rel O R*
⟨*proof*⟩

lemma *alpha-Tree-rel-relcompI* [*simp*]:
assumes $x =_\alpha x'$ **and** $(x', y) \in R$
shows $(x, y) \in$ *alpha-Tree-rel O R*
⟨*proof*⟩

6.1 Validity for infinitely branching trees

context *nominal-ts*
begin

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

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

function *valid-Tree* :: *'state* ⇒ (*'idx*,*'pred*,*'act*) *Tree* ⇒ *bool* **where**
valid-Tree P (*tConj tset*) ⟷ (∀ *t*∈*set-bset tset*. *valid-Tree P t*)
| *valid-Tree P* (*tNot t*) ⟷ ¬ *valid-Tree P t*
| *valid-Tree P* (*tPred* φ) ⟷ $P \vdash \varphi$
| *valid-Tree P* (*tAct* α *t*) ⟷ (∃ α' t' P'. *tAct* α $t =_\alpha$ *tAct* α' t' ∧ $P \rightarrow \langle\alpha',P'\rangle$ ∧ *valid-Tree* P' t')
⟨*proof*⟩

termination $\langle proof \rangle$

valid-Tree is equivariant.

lemma *valid-Tree-eqvt'*: *valid-Tree P t* $\longleftrightarrow$ *valid-Tree* $(p \cdot P)$ $(p \cdot t)$
$\langle proof \rangle$

lemma *valid-Tree-eqvt* :
assumes *valid-Tree P t* **shows** *valid-Tree* $(p \cdot P)$ $(p \cdot t)$
$\langle proof \rangle$

α-equivalent trees validate the same states.

lemma *alpha-Tree-valid-Tree*:
assumes $t1 =_\alpha t2$
shows *valid-Tree P t1* $\longleftrightarrow$ *valid-Tree P t2*
$\langle proof \rangle$

6.2 Validity for trees modulo α-equivalence

lift-definition $valid\text{-}Tree_\alpha$ :: $'state \Rightarrow ('idx,'pred,'act)\ Tree_\alpha \Rightarrow bool$ **is**
valid-Tree
$\langle proof \rangle$

lemma $valid\text{-}Tree_\alpha\text{-}eqvt$:
assumes $valid\text{-}Tree_\alpha\ P\ t$ **shows** $valid\text{-}Tree_\alpha\ (p \cdot P)\ (p \cdot t)$
$\langle proof \rangle$

lemma $valid\text{-}Tree_\alpha\text{-}Conj_\alpha$ [*simp*]: $valid\text{-}Tree_\alpha\ P\ (Conj_\alpha\ tset_\alpha) \longleftrightarrow (\forall\ t_\alpha \in set\text{-}bset\ tset_\alpha.\ valid\text{-}Tree_\alpha\ P\ t_\alpha)$
$\langle proof \rangle$

lemma $valid\text{-}Tree_\alpha\text{-}Not_\alpha$ [*simp*]: $valid\text{-}Tree_\alpha\ P\ (Not_\alpha\ t_\alpha) \longleftrightarrow \neg\ valid\text{-}Tree_\alpha\ P\ t_\alpha$
$\langle proof \rangle$

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

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

6.3 Validity for infinitary formulas

lift-definition *valid* :: $'state \Rightarrow ('idx,'pred,'act)\ formula \Rightarrow bool$ (**infix** ‹⊨› *70*) **is**
$valid\text{-}Tree_\alpha$
$\langle proof \rangle$

lemma *valid-eqvt* :

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

lemma *valid-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows $P \models$ *Conj xset* $\longleftrightarrow$ ($\forall x \in$*set-bset xset*. $P \models x$)
⟨*proof*⟩

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

lemma *valid-Pred* [*simp*]: $P \models$ *Pred* $\varphi \longleftrightarrow P \vdash \varphi$
⟨*proof*⟩

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

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

lemma *valid-Act-strong*:
assumes *finite* (*supp* X)
shows $P \models$ *Act* $\alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.$ *Act* $\alpha\ x =$ *Act* $\alpha'\ x' \wedge P \rightarrow \langle\alpha',P'\rangle \wedge P' \models x' \wedge$ *bn* $\alpha' \sharp* X)$
⟨*proof*⟩

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

end

end
theory *Logical-Equivalence*
imports
Validity
begin

7 (Strong) Logical Equivalence

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

locale *indexed-nominal-ts* = *nominal-ts satisfies transition*
for *satisfies* :: *'state::fs* ⇒ *'pred::fs* ⇒ *bool* (**infix** ‹⊢› *70*)
and *transition* :: *'state* ⇒ (*'act::bn*,*'state*) *residual* ⇒ *bool* (**infix** ‹→› *70*) +
assumes *card-idx-perm*: |*UNIV*::*perm set*| <*o* |*UNIV*::*'idx set*|
and *card-idx-state*: |*UNIV*::*'state set*| <*o* |*UNIV*::*'idx set*|

begin

definition *logically-equivalent* :: $'state \Rightarrow 'state \Rightarrow bool$ **where**
logically-equivalent $P\ Q \equiv (\forall\, x{::}('idx,'pred,'act)\ formula.\ P \models x \longleftrightarrow Q \models x)$

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

lemma *logically-equivalent-eqvt*:
assumes $P =\cdot\ Q$ **shows** $p \cdot P =\cdot\ p \cdot Q$
⟨*proof*⟩

end

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

8 Bisimilarity Implies Logical Equivalence

context *indexed-nominal-ts*
begin

lemma *bisimilarity-implies-equivalence-Act*:
assumes $\bigwedge P\ Q.\ P \sim\cdot\ Q \Longrightarrow P \models x \longleftrightarrow Q \models x$
and $P \sim\cdot\ Q$
and $P \models Act\ \alpha\ x$
shows $Q \models Act\ \alpha\ x$
⟨*proof*⟩

theorem *bisimilarity-implies-equivalence*: **assumes** $P \sim\cdot\ Q$ **shows** $P =\cdot\ Q$
⟨*proof*⟩

end

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

9 Logical Equivalence Implies Bisimilarity

context *indexed-nominal-ts*
begin

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

$'state \Rightarrow bool$
  (‹- *distinguishes* - *from* -› [*100*,*100*,*100*] *100*)
  where
  x *distinguishes* P *from* $Q \equiv P \models x \wedge \neg\ Q \models x$

lemma *is-distinguishing-formula-eqvt* :
  assumes *x distinguishes P from Q* **shows** $(p \cdot x)$ *distinguishes* $(p \cdot P)$ *from* $(p \cdot Q)$
  ⟨*proof*⟩

lemma *equivalent-iff-not-distinguished*: $(P =\cdot\ Q) \longleftrightarrow \neg(\exists\, x.\ x$ *distinguishes* P *from* $Q)$
  ⟨*proof*⟩

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

lemma *distinguished-bounded-support*:
  assumes *x distinguishes P from Q*
  obtains *y* **where** *supp y* $\subseteq$ *supp P* **and** *y distinguishes P from Q*
  ⟨*proof*⟩

lemma *equivalence-is-bisimulation*: *is-bisimulation logically-equivalent*
  ⟨*proof*⟩

theorem *equivalence-implies-bisimilarity*: **assumes** $P =\cdot\ Q$ **shows** $P \sim\cdot\ Q$
  ⟨*proof*⟩

end

end
theory *Disjunction*
imports
  Formula
  Validity
begin

10 Disjunction

definition *Disj* :: ($'idx$,$'pred$::*fs*,$'act$::*bn*) *formula set*[$'idx$] $\Rightarrow$ ($'idx$,$'pred$,$'act$) *formula* **where**
  Disj xset = *Not* (*Conj* (*map-bset Not xset*))

lemma *finite-supp-map-bset-Not* [*simp*]:
  assumes *finite* (*supp xset*)
  shows *finite* (*supp* (*map-bset Not xset*))
⟨*proof*⟩

lemma *Disj-eqvt* [*simp*]:
  assumes *finite* (*supp xset*)

shows $p \cdot \mathit{Disj}\ \mathit{xset} = \mathit{Disj}\ (p \cdot \mathit{xset})$
⟨*proof*⟩

lemma *Disj-eq-iff* [*simp*]:
assumes *finite* (*supp xset1*) **and** *finite* (*supp xset2*)
shows $\mathit{Disj}\ \mathit{xset1} = \mathit{Disj}\ \mathit{xset2} \longleftrightarrow \mathit{xset1} = \mathit{xset2}$
⟨*proof*⟩

context *nominal-ts*
begin

lemma *valid-Disj* [*simp*]:
assumes *finite* (*supp xset*)
shows $P \models \mathit{Disj}\ \mathit{xset} \longleftrightarrow (\exists x{\in}\mathit{set\text{-}bset}\ \mathit{xset}.\ P \models x)$
⟨*proof*⟩

end

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

11 Expressive Completeness

context *indexed-nominal-ts*
begin

11.1 Distinguishing formulas

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

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

definition *distinguishing-formula* :: $'\mathit{state} \Rightarrow {}'\mathit{state} \Rightarrow ('\mathit{idx}, '\mathit{pred}, '\mathit{act})\ \mathit{formula}$
where
$\mathit{distinguishing\text{-}formula}\ P\ Q \equiv \mathit{Conj}\ (\mathit{Abs\text{-}bset}\ \{-p \cdot (\epsilon\, x.\ \mathit{supp}\ x \subseteq \mathit{supp}\ (p \cdot P) \wedge x\ \mathit{distinguishes}\ (p \cdot P)\ \mathit{from}\ (p \cdot Q))|p.\ \mathit{True}\})$

— just an auxiliary lemma that will be useful further below
lemma *distinguishing-formula-card-aux*:
$|\{-p \cdot (\epsilon\, x.\ \mathit{supp}\ x \subseteq \mathit{supp}\ (p \cdot P) \wedge x\ \mathit{distinguishes}\ (p \cdot P)\ \mathit{from}\ (p \cdot Q))|p.$

$True\}| <o\ natLeq +c\ |UNIV :: 'idx\ set|$
⟨proof⟩
lemma *distinguishing-formula-supp-aux*:
assumes $\neg (P =\cdot Q)$
shows $supp\ (Abs\text{-}bset\ \{-p \cdot (\epsilon x.\ supp\ x \subseteq supp\ (p \cdot P) \wedge x\ distinguishes\ (p \cdot P)\ from\ (p \cdot Q))|p.\ True\} :: \text{-}\ set['idx]) \subseteq supp\ P$
⟨proof⟩

lemma *distinguishing-formula-eqvt* [*simp*]:
assumes $\neg (P =\cdot Q)$
shows $p \cdot distinguishing\text{-}formula\ P\ Q = distinguishing\text{-}formula\ (p \cdot P)\ (p \cdot Q)$
⟨proof⟩

lemma *supp-distinguishing-formula*:
assumes $\neg (P =\cdot Q)$
shows $supp\ (distinguishing\text{-}formula\ P\ Q) \subseteq supp\ P$
⟨proof⟩

lemma *distinguishing-formula-distinguishes*:
assumes $\neg (P =\cdot Q)$
shows $(distinguishing\text{-}formula\ P\ Q)\ distinguishes\ P\ from\ Q$
⟨proof⟩

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

— just an auxiliary lemma that will be useful further below
lemma *characteristic-formula-card-aux*:
$|\{distinguishing\text{-}formula\ P\ Q|Q.\ \neg (P =\cdot Q)\}| <o\ natLeq +c\ |UNIV :: 'idx\ set|$
⟨proof⟩
lemma *characteristic-formula-supp-aux*:
shows $supp\ (Abs\text{-}bset\ \{distinguishing\text{-}formula\ P\ Q|Q.\ \neg (P =\cdot Q)\} :: \text{-}\ set['idx]) \subseteq supp\ P$
⟨proof⟩

lemma *characteristic-formula-eqvt* [*simp*]:
$p \cdot characteristic\text{-}formula\ P = characteristic\text{-}formula\ (p \cdot P)$
⟨proof⟩

lemma *characteristic-formula-eqvt-raw* [*simp*]:
$p \cdot characteristic\text{-}formula = characteristic\text{-}formula$
⟨proof⟩

lemma *characteristic-formula-is-characteristic'*:

$Q \models$ *characteristic-formula* $P \longleftrightarrow P =\cdot Q$
⟨*proof*⟩

lemma *characteristic-formula-is-characteristic*:
$Q \models$ *characteristic-formula* $P \longleftrightarrow P \sim\cdot Q$
⟨*proof*⟩

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 $\bigwedge P\ Q.\ P \in S \Longrightarrow P \sim\cdot Q \Longrightarrow Q \in S$
shows $P \models$ *Disj* (*Abs-bset* (*characteristic-formula* ‘ S)) $\longleftrightarrow P \in S$
⟨*proof*⟩

end

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

12 Finitely Supported Sets

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

typedef $'a$ *fs-set* = $\{x::'a::pt\ set.\ finite\ (supp\ x)\}$
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

instance
⟨*proof*⟩

end

Set membership.

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

notation
member-fs-set (‹'($\in_{fs}$')›) **and**
member-fs-set (‹(-/ $\in_{fs}$ -)› [*51*, *51*] *50*)

lemma *member-fs-set-permute-iff* [*simp*]: $p \cdot x \in_{fs} p \cdot X \longleftrightarrow x \in_{fs} X$
⟨*proof*⟩

lemma *member-fs-set-eqvt* [*eqvt*]: $x \in_{fs} X \Longrightarrow p \cdot x \in_{fs} p \cdot X$
⟨*proof*⟩

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

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

13.1 Nominal transition systems with effects

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

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

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

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

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

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

locale *effect-nominal-ts* = *nominal-ts satisfies transition*
for *satisfies* :: $'state::fs \Rightarrow 'pred::fs \Rightarrow bool$ (**infix** ‹⊢› *70*)
and *transition* :: $'state \Rightarrow ('act::bn,'state)\ residual \Rightarrow bool$ (**infix** ‹→› *70*) +
fixes *effect-apply* :: $'effect::fs \Rightarrow 'state \Rightarrow 'state$ (‹⟨-⟩-› [*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*]: $p \cdot effect\text{-}apply = effect\text{-}apply$
⟨*proof*⟩

lemma *effect-apply-eqvt′* [*eqvt*]: $p \cdot \langle f\rangle P = \langle p \cdot f\rangle(p \cdot P)$
⟨*proof*⟩

lemma *L-eqvt-aux* [*simp*]: $p \cdot L = L$
⟨*proof*⟩

lemma *L-eqvt′* [*eqvt*]: $p \cdot L\ (\alpha,\ P,\ f) = L\ (p \cdot \alpha,\ p \cdot P,\ p \cdot f)$
⟨*proof*⟩

end

13.2 L-bisimulations and F/L-bisimilarity

context *effect-nominal-ts*
begin

definition *is-L-bisimulation*:: $('effect\ first \Rightarrow 'state \Rightarrow 'state \Rightarrow bool) \Rightarrow bool$
where
is-L-bisimulation $R \equiv$
$\forall F.\ symp\ (R\ F) \wedge$
$(\forall P\ Q.\ R\ F\ P\ Q \longrightarrow (\forall f.\ f \in_{fs} F \longrightarrow (\forall \varphi.\ \langle f\rangle P \vdash \varphi \longrightarrow \langle f\rangle Q \vdash \varphi))) \wedge$
$(\forall P\ Q.\ R\ F\ P\ Q \longrightarrow (\forall f.\ f \in_{fs} F \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha \sharp{*}\ (\langle f\rangle Q,\ F,\ f) \longrightarrow$
$\langle f\rangle P \rightarrow \langle \alpha,P'\rangle \longrightarrow (\exists Q'.\ \langle f\rangle Q \rightarrow \langle \alpha,Q'\rangle \wedge R\ (L\ (\alpha,F,f))\ P'\ Q'))))$

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

abbreviation *FL-bisimilar′* (‹- ∼·[-] -› [*51*,*0*,*51*] *50*) **where**
$P \sim\cdot[F]\ Q \equiv FL\text{-}bisimilar\ F\ P\ Q$

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

lemma *is-L-bisimulation-eqvt* [*eqvt*]:
assumes *is-L-bisimulation* R **shows** *is-L-bisimulation* $(p \cdot R)$
⟨*proof*⟩

lemma *FL-bisimilar-eqvt*:
assumes $P \sim\cdot[F]\ Q$ **shows** $(p \cdot P) \sim\cdot[p \cdot F]\ (p \cdot Q)$
⟨*proof*⟩

lemma *FL-bisimilar-reflp*: *reflp* (*FL-bisimilar F*)
⟨*proof*⟩

lemma *FL-bisimilar-symp*: *symp* (*FL-bisimilar F*)
⟨*proof*⟩

lemma *FL-bisimilar-is-L-bisimulation*: *is-L-bisimulation FL-bisimilar*
⟨*proof*⟩

lemma *FL-bisimilar-simulation-step*:
assumes $P \sim\cdot[F]\ Q$ **and** $f \in_{fs} F$ **and** $bn\ \alpha \sharp* (\langle f\rangle Q,\ F,\ f)$ **and** $\langle f\rangle P \to \langle \alpha, P'\rangle$
obtains Q' **where** $\langle f\rangle Q \to \langle \alpha, Q'\rangle$ **and** $P' \sim\cdot[L\ (\alpha, F, f)]\ Q'$
⟨*proof*⟩

lemma *FL-bisimilar-transp*: *transp* (*FL-bisimilar F*)
⟨*proof*⟩

lemma *FL-bisimilar-equivp*: *equivp* (*FL-bisimilar F*)
⟨*proof*⟩

end

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

14 Infinitary Formulas With Effects

14.1 Infinitely branching trees

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

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

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

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

inductive-set *Tree-wf* :: ('*idx*,'*pred*,'*act*,'*eff*) *Tree rel* **where**
  $t \in$ *set-bset tset* $\Longrightarrow$ (t, *tConj tset*) $\in$ *Tree-wf*
| (t, *tNot* t) $\in$ *Tree-wf*
| (t, *tAct* f α t) $\in$ *Tree-wf*

lemma *wf-Tree-wf*: *wf Tree-wf*
⟨*proof*⟩

We define a permutation operation on the type of trees.

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

  primrec *permute-Tree* :: *perm* ⇒ (-,-,-,-) *Tree* ⇒ (-,-,-,-) *Tree* **where**
    $p \cdot$ (*tConj tset*) = *tConj* (*map-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 φ) = *tPred* ($p \cdot f$) ($p \cdot \varphi$)
  | $p \cdot$ (*tAct* f α t) = *tAct* ($p \cdot f$) ($p \cdot \alpha$) ($p \cdot t$)

  instance
  ⟨*proof*⟩

end

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

lemma *permute-Tree-tConj* [*simp*]: $p \cdot$ *tConj tset* = *tConj* ($p \cdot$ *tset*)
⟨*proof*⟩

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

The relation *Tree-wf* is equivariant.

lemma *Tree-wf-eqvt-aux*:
  assumes (*t1*, *t2*) $\in$ *Tree-wf* **shows** ($p \cdot$ *t1*, $p \cdot$ *t2*) $\in$ *Tree-wf*
⟨*proof*⟩

lemma *Tree-wf-eqvt* [*eqvt*, *simp*]: $p \cdot \mathit{Tree\text{-}wf} = \mathit{Tree\text{-}wf}$
⟨*proof*⟩

lemma *Tree-wf-eqvt'*: *eqvt Tree-wf*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *supp-tNot* [*simp*]: *supp* (*tNot t*) = *supp t*
⟨*proof*⟩

lemma *supp-tPred* [*simp*]: *supp* (*tPred f* φ) = *supp f* $\cup$ *supp* φ
⟨*proof*⟩

lemma *supp-tAct* [*simp*]: *supp* (*tAct f* α *t*) = *supp f* $\cup$ *supp* α $\cup$ *supp t*
⟨*proof*⟩

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*]:
 $p \cdot$ *supp-rel R x* = *supp-rel* ($p \cdot R$) ($p \cdot x$)
⟨*proof*⟩

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

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

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

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

lemma *supp-rel-cong* [*fundef-cong*]:
$\llbracket$ $x{=}x'$; $\bigwedge a\ b.\ R\ ((a \rightleftharpoons b) \cdot x')\ x' \longleftrightarrow R'\ ((a \rightleftharpoons b) \cdot x')\ x'$ $\rrbracket \Longrightarrow$ *supp-rel* $R\ x$ = *supp-rel* $R'\ x'$
⟨*proof*⟩

lemma *rel-bset-cong* [*fundef-cong*]:
$\llbracket$ $x{=}x'$; $y{=}y'$; $\bigwedge a\ b.\ a \in$ *set-bset* $x' \Longrightarrow b \in$ *set-bset* $y' \Longrightarrow R\ a\ b \longleftrightarrow R'\ a\ b$ $\rrbracket$ $\Longrightarrow$ *rel-bset* $R\ x\ y \longleftrightarrow$ *rel-bset* $R'\ x'\ y'$
⟨*proof*⟩

lemma *alpha-set-cong* [*fundef-cong*]:
$\llbracket$ $bs{=}bs'$; $x{=}x'$; $R\ (p' \cdot x')\ y' \longleftrightarrow R'\ (p' \cdot x')\ y'$; $f\ x' = f'\ x'$; $f\ y' = f'\ y'$; $p{=}p'$; $cs{=}cs'$; $y{=}y'$ $\rrbracket \Longrightarrow$
alpha-set $(bs,\ x)\ R\ f\ p\ (cs,\ y) \longleftrightarrow$ *alpha-set* $(bs',\ x')\ R'\ f'\ p'\ (cs',\ y')$
⟨*proof*⟩

quotient-type
('*idx*,'*pred*,'*act*,'*eff*) $Tree_p$ = ('*idx*,'*pred*::*pt*,'*act*::*bn*,'*eff*::*fs*) *Tree* / *hull-relp*
⟨*proof*⟩

lemma *abs-*$Tree_p$*-eq* [*simp*]: *abs-*$Tree_p$ $(p \cdot t)$ = *abs-*$Tree_p$ t
⟨*proof*⟩

lemma *permute-rep-abs-*$Tree_p$:
obtains p **where** *rep-*$Tree_p$ (*abs-*$Tree_p$ t) = $p \cdot t$
⟨*proof*⟩

lift-definition *Tree-wf*$_p$ :: ('*idx*,'*pred*::*pt*,'*act*::*bn*,'*eff*::*fs*) $Tree_p$ *rel* **is**
Tree-wf ⟨*proof*⟩

lemma *Tree-wf*$_p$*I* [*simp*]:
assumes $(a,\ b) \in$ *Tree-wf*
shows (*abs-*$Tree_p$ $(p \cdot a)$, *abs-*$Tree_p$ b) $\in$ *Tree-wf*$_p$
⟨*proof*⟩

lemma *Tree-wf*$_p$*-trivialI* [*simp*]:
assumes $(a,\ b) \in$ *Tree-wf*
shows (*abs-*$Tree_p$ a, *abs-*$Tree_p$ b) $\in$ *Tree-wf*$_p$
⟨*proof*⟩

lemma *Tree-wf*$_p$*E*:
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*
⟨*proof*⟩

lemma *wf-Tree-wf*$_p$: *wf Tree-wf*$_p$
⟨*proof*⟩

fun *alpha-Tree-termination* :: ('*a*, '*b*, '*c*, '*d*) *Tree* × ('*a*, '*b*, '*c*, '*d*) *Tree* ⇒ ('*a*,

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

Here it comes ...

function (*sequential*)
alpha-Tree :: ('*idx*,'*pred*::*pt*,'*act*::*bn*,'*eff*::*fs*) *Tree* ⇒ ('*idx*,'*pred*,'*act*,'*eff*) *Tree* ⇒ *bool* (**infix** ‹$=_\alpha$› *50*) **where**
— ($=_\alpha$)
alpha-tConj: *tConj tset1* $=_\alpha$ *tConj tset2* ⟷ *rel-bset alpha-Tree tset1 tset2*
| *alpha-tNot*: *tNot t1* $=_\alpha$ *tNot t2* ⟷ *t1* $=_\alpha$ *t2*
| *alpha-tPred*: *tPred f1* φ*1* $=_\alpha$ *tPred f2* φ*2* ⟷ *f1* = *f2* ∧ φ*1* = φ*2*
— the action may have binding names
| *alpha-tAct*: *tAct f1* α*1 t1* $=_\alpha$ *tAct f2* α*2 t2* ⟷
f1 = *f2* ∧ (∃ *p*. (*bn* α*1*, *t1*) ≈*set alpha-Tree* (*supp-rel alpha-Tree*) *p* (*bn* α*2*, *t2*) ∧ (*bn* α*1*, α*1*) ≈*set* ((=)) *supp p* (*bn* α*2*, α*2*))
| *alpha-other*: - $=_\alpha$ - ⟷ *False*
— 254 subgoals (!)
⟨*proof*⟩
termination
⟨*proof*⟩

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

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

lemma *alpha-Tree-induct*[*case-names tConj tNot tPred tAct*, *consumes 1*]:
assumes *t1* $=_\alpha$ *t2*
and ⋀*tset1 tset2*. (⋀*a b*. *a* ∈ *set-bset tset1* ⟹ *b* ∈ *set-bset tset2* ⟹ *a* $=_\alpha$ *b* ⟹ *P a b*) ⟹
rel-bset ($=_\alpha$) *tset1 tset2* ⟹ *P* (*tConj tset1*) (*tConj tset2*)
and ⋀*t1 t2*. *t1* $=_\alpha$ *t2* ⟹ *P t1 t2* ⟹ *P* (*tNot t1*) (*tNot t2*)
and ⋀*f* φ. *P* (*tPred f* φ) (*tPred f* φ)
and ⋀*f1* α*1 t1 f2* α*2 t2*. (⋀*p*. *p* · *t1* $=_\alpha$ *t2* ⟹ *P* (*p* · *t1*) *t2*) ⟹ *f1* = *f2* ⟹
(∃ *p*. (*bn* α*1*, *t1*) ≈*set* ($=_\alpha$) (*supp-rel* ($=_\alpha$)) *p* (*bn* α*2*, *t2*) ∧ (*bn* α*1*, α*1*) ≈*set* (=) *supp p* (*bn* α*2*, α*2*)) ⟹
P (*tAct f1* α*1 t1*) (*tAct f2* α*2 t2*)
shows *P t1 t2*
⟨*proof*⟩

α-equivalence is equivariant.

lemma *alpha-Tree-eqvt-aux*:
assumes ⋀*a b*. (*a* ⇌ *b*) · *t* $=_\alpha$ *t* ⟷ *p* · (*a* ⇌ *b*) · *t* $=_\alpha$ *p* · *t*
shows *p* · *supp-rel* ($=_\alpha$) *t* = *supp-rel* ($=_\alpha$) (*p* · *t*)
⟨*proof*⟩

lemma *alpha-Tree-eqvt'*: $t1 =_\alpha t2 \longleftrightarrow p \cdot t1 =_\alpha p \cdot t2$
⟨*proof*⟩

lemma *alpha-Tree-eqvt* [*eqvt*]: $t1 =_\alpha t2 \Longrightarrow p \cdot t1 =_\alpha p \cdot t2$
⟨*proof*⟩

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

lemma *alpha-Tree-reflp*: *reflp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-symp*: *symp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-transp*: *transp alpha-Tree*
⟨*proof*⟩

lemma *alpha-Tree-equivp*: *equivp alpha-Tree*
⟨*proof*⟩

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

lemma *alpha-Tree-supp-rel*:
  assumes $t1 =_\alpha t2$
  shows $\textit{supp-rel}\ (=_\alpha)\ t1 = \textit{supp-rel}\ (=_\alpha)\ t2$
⟨*proof*⟩

tAct preserves α-equivalence.

lemma *alpha-Tree-tAct*:
  assumes $t1 =_\alpha t2$
  shows $\textit{tAct}\ f\ \alpha\ t1 =_\alpha \textit{tAct}\ f\ \alpha\ t2$
⟨*proof*⟩

The following lemmas describe the support modulo α-equivalence.

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

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

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

lemma *infinite-mono*: $\textit{infinite}\ S \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow x \in T) \Longrightarrow \textit{infinite}\ T$
⟨*proof*⟩

lemma *supp-rel-tAct* [*simp*]:
  assumes $\textit{finite}\ (\textit{supp-rel}\ (=_\alpha)\ t)$
  shows $\textit{supp-rel}\ (=_\alpha)\ (\textit{tAct}\ f\ \alpha\ t) = \textit{supp}\ f \cup (\textit{supp}\ \alpha \cup \textit{supp-rel}\ (=_\alpha)\ t - \textit{bn}\ \alpha)$

⟨*proof*⟩

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

quotient-type
 $('idx,'pred,'act,'eff)\ Tree_\alpha = ('idx,'pred::pt,'act::bn,'eff::fs)\ Tree\ /\ alpha\text{-}Tree$
 ⟨*proof*⟩

lemma $Tree_\alpha\text{-}abs\text{-}rep$ [*simp*]: $abs\text{-}Tree_\alpha\ (rep\text{-}Tree_\alpha\ t_\alpha) = t_\alpha$
⟨*proof*⟩

lemma $Tree_\alpha\text{-}rep\text{-}abs$ [*simp*]: $rep\text{-}Tree_\alpha\ (abs\text{-}Tree_\alpha\ t) =_\alpha t$
⟨*proof*⟩

The permutation operation is lifted from trees.

instantiation $Tree_\alpha$ :: (*type*, *pt*, *bn*, *fs*) *pt*
begin

 lift-definition $permute\text{-}Tree_\alpha :: perm \Rightarrow ('a,'b,'c,'d)\ Tree_\alpha \Rightarrow ('a,'b,'c,'d)\ Tree_\alpha$
 is *permute*
 ⟨*proof*⟩

 instance
 ⟨*proof*⟩

end

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

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

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

14.3 Constructors for trees modulo α-equivalence

The constructors are lifted from trees.

lift-definition $Conj_\alpha :: ('idx,'pred,'act,'eff)\ Tree_\alpha\ set['idx] \Rightarrow ('idx,'pred::pt,'act::bn,'eff::fs)\ Tree_\alpha$ **is**
 tConj
⟨*proof*⟩

lemma $map\text{-}bset\text{-}abs\text{-}rep\text{-}Tree_\alpha$: $map\text{-}bset\ abs\text{-}Tree_\alpha\ (map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha) = tset_\alpha$
⟨*proof*⟩

lemma $Conj_\alpha\text{-}def'$: $Conj_\alpha\ tset_\alpha = abs\text{-}Tree_\alpha\ (tConj\ (map\text{-}bset\ rep\text{-}Tree_\alpha\ tset_\alpha))$
⟨*proof*⟩

lift-definition Not_α :: ('idx,'pred,'act,'eff) $Tree_\alpha \Rightarrow$ ('idx,'pred::pt,'act::bn,'eff::fs) $Tree_\alpha$ **is**
tNot
⟨*proof*⟩

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

lift-definition Act_α :: 'eff $\Rightarrow$ 'act $\Rightarrow$ ('idx,'pred,'act,'eff) $Tree_\alpha \Rightarrow$ ('idx,'pred::pt,'act::bn,'eff::fs) $Tree_\alpha$ **is**
tAct
⟨*proof*⟩

The lifted constructors are equivariant.

lemma $Conj_\alpha$*-eqvt* [*eqvt, simp*]: $p \cdot Conj_\alpha\ tset_\alpha = Conj_\alpha\ (p \cdot tset_\alpha)$
⟨*proof*⟩

lemma Not_α*-eqvt* [*eqvt, simp*]: $p \cdot Not_\alpha\ t_\alpha = Not_\alpha\ (p \cdot t_\alpha)$
⟨*proof*⟩

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

lemma Act_α*-eqvt* [*eqvt, simp*]: $p \cdot Act_\alpha\ f\ \alpha\ t_\alpha = Act_\alpha\ (p \cdot f)\ (p \cdot \alpha)\ (p \cdot t_\alpha)$
⟨*proof*⟩

The lifted constructors are injective (except for Act_α).

lemma $Conj_\alpha$*-eq-iff* [*simp*]: $Conj_\alpha\ tset1_\alpha = Conj_\alpha\ tset2_\alpha \longleftrightarrow tset1_\alpha = tset2_\alpha$
⟨*proof*⟩

lemma Not_α*-eq-iff* [*simp*]: $Not_\alpha\ t1_\alpha = Not_\alpha\ t2_\alpha \longleftrightarrow t1_\alpha = t2_\alpha$
⟨*proof*⟩

lemma $Pred_\alpha$*-eq-iff* [*simp*]: $Pred_\alpha\ f1\ \varphi 1 = Pred_\alpha\ f2\ \varphi 2 \longleftrightarrow f1 = f2 \wedge \varphi 1 = \varphi 2$
⟨*proof*⟩

lemma Act_α*-eq-iff*: $Act_\alpha\ f1\ \alpha 1\ t1 = Act_\alpha\ f2\ \alpha 2\ t2 \longleftrightarrow tAct\ f1\ \alpha 1\ (rep\text{-}Tree_\alpha\ t1) =_\alpha tAct\ f2\ \alpha 2\ (rep\text{-}Tree_\alpha\ t2)$
⟨*proof*⟩

The lifted constructors are free (except for Act_α).

lemma $Tree_\alpha$*-free* [*simp*]:
shows $Conj_\alpha\ tset_\alpha \neq Not_\alpha\ t_\alpha$
and $Conj_\alpha\ tset_\alpha \neq Pred_\alpha\ f\ \varphi$
and $Conj_\alpha\ tset_\alpha \neq Act_\alpha\ f\ \alpha\ t_\alpha$
and $Not_\alpha\ t_\alpha \neq Pred_\alpha\ f\ \varphi$
and $Not_\alpha\ t1_\alpha \neq Act_\alpha\ f\ \alpha\ t2_\alpha$

and $Pred_{\alpha}\ f1\ \varphi \neq Act_{\alpha}\ f2\ \alpha\ t_{\alpha}$
⟨*proof*⟩

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

lemma *supp-alpha-supp-rel*: $supp\ t_{\alpha} = supp\text{-}rel\ (=_{\alpha})\ (rep\text{-}Tree_{\alpha}\ t_{\alpha})$
⟨*proof*⟩

lemma *supp-Conj*$_{\alpha}$ [*simp*]: $supp\ (Conj_{\alpha}\ tset_{\alpha}) = supp\ tset_{\alpha}$
⟨*proof*⟩

lemma *supp-Not*$_{\alpha}$ [*simp*]: $supp\ (Not_{\alpha}\ t_{\alpha}) = supp\ t_{\alpha}$
⟨*proof*⟩

lemma *supp-Pred*$_{\alpha}$ [*simp*]: $supp\ (Pred_{\alpha}\ f\ \varphi) = supp\ f \cup supp\ \varphi$
⟨*proof*⟩

lemma *supp-Act*$_{\alpha}$ [*simp*]:
assumes $finite\ (supp\ t_{\alpha})$
shows $supp\ (Act_{\alpha}\ f\ \alpha\ t_{\alpha}) = supp\ f \cup (supp\ \alpha \cup supp\ t_{\alpha} - bn\ \alpha)$
⟨*proof*⟩

14.4 Induction over trees modulo α-equivalence

lemma *Tree*$_{\alpha}$*-induct* [*case-names* $Conj_{\alpha}$ Not_{α} $Pred_{\alpha}$ Act_{α} Env_{α}, *induct type*: $Tree_{\alpha}$]:
fixes t_{α}
assumes $\bigwedge tset_{\alpha}.\ (\bigwedge x.\ x \in set\text{-}bset\ tset_{\alpha} \Longrightarrow P\ x) \Longrightarrow P\ (Conj_{\alpha}\ tset_{\alpha})$
and $\bigwedge t_{\alpha}.\ P\ t_{\alpha} \Longrightarrow P\ (Not_{\alpha}\ t_{\alpha})$
and $\bigwedge f\ pred.\ P\ (Pred_{\alpha}\ f\ pred)$
and $\bigwedge f\ act\ t_{\alpha}.\ P\ t_{\alpha} \Longrightarrow P\ (Act_{\alpha}\ f\ act\ t_{\alpha})$
shows $P\ t_{\alpha}$
⟨*proof*⟩

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

14.5 Hereditarily finitely supported trees

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

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

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

hereditarily-fs is equivariant.

lemma *hereditarily-fs-eqvt* [*eqvt*]:
  assumes *hereditarily-fs* t_α
  shows *hereditarily-fs* ($p \cdot t_\alpha$)
⟨*proof*⟩

hereditarily-fs is preserved under α-renaming.

lemma *hereditarily-fs-alpha-renaming*:
  assumes Act_α f α t_α = Act_α f' α' t_α'
  shows *hereditarily-fs* $t_\alpha \longleftrightarrow$ *hereditarily-fs* t_α'
⟨*proof*⟩

Hereditarily finitely supported trees have finite support.

lemma *hereditarily-fs-implies-finite-supp*:
  assumes *hereditarily-fs* t_α
  shows *finite* (*supp* t_α)
⟨*proof*⟩

14.6 Infinitary formulas

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

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

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
⟨*proof*⟩

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

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

lemma *Abs-formula-eqvt* [*simp*]:
assumes *hereditarily-fs* t_α
shows *p* · *Abs-formula* t_α = *Abs-formula* (*p* · t_α)
⟨*proof*⟩

lemma *supp-Abs-formula* [*simp*]:
assumes *hereditarily-fs* t_α
shows *supp* (*Abs-formula* t_α) = *supp* t_α
⟨*proof*⟩

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

lemma *supp-Rep-formula* [*simp*]: *supp* (*Rep-formula x*) = *supp x*
⟨*proof*⟩

lemma *supp-map-bset-Rep-formula* [*simp*]: *supp* (*map-bset Rep-formula xset*) = *supp xset*
⟨*proof*⟩

Formulas are in fact finitely supported.

instance *formula* :: (*type*, *fs*, *bn*, *fs*) *fs*
⟨*proof*⟩

14.7 Constructors for infinitary formulas

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

definition *Conj* :: (*'idx*,*'pred*,*'act*,*'eff*) *formula set*[*'idx*] ⇒ (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* **where**

Conj xset = *Abs-formula* (*Conj*$_\alpha$ (*map-bset Rep-formula xset*))

lemma *finite-supp-implies-hereditarily-fs-Conj*$_\alpha$ [*simp*]:
assumes *finite* (*supp xset*)
shows *hereditarily-fs* (*Conj*$_\alpha$ (*map-bset Rep-formula xset*))
⟨*proof*⟩

lemma *Conj-rep-eq*:
assumes *finite* (*supp xset*)
shows *Rep-formula* (*Conj xset*) = *Conj*$_\alpha$ (*map-bset Rep-formula xset*)
⟨*proof*⟩

lift-definition *Not* :: (*'idx*,*'pred*,*'act*,*'eff*) *formula* ⇒ (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* **is**
Not$_\alpha$
⟨*proof*⟩

lift-definition *Pred* :: *'eff* ⇒ *'pred* ⇒ (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* **is**
Pred$_\alpha$
⟨*proof*⟩

lift-definition *Act* :: *'eff* ⇒ *'act* ⇒ (*'idx*,*'pred*,*'act*,*'eff*) *formula* ⇒ (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* **is**
Act$_\alpha$
⟨*proof*⟩

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

lemma *Conj-eqvt* [*simp*]:
assumes *finite* (*supp xset*)
shows *p* • *Conj xset* = *Conj* (*p* • *xset*)
⟨*proof*⟩

lemma *Not-eqvt* [*eqvt*, *simp*]: *p* • *Not x* = *Not* (*p* • *x*)
⟨*proof*⟩

lemma *Pred-eqvt* [*eqvt*, *simp*]: *p* • *Pred f* φ = *Pred* (*p* • *f*) (*p* • φ)
⟨*proof*⟩

lemma *Act-eqvt* [*eqvt*, *simp*]: *p* • *Act f* α *x* = *Act* (*p* • *f*) (*p* • α) (*p* • *x*)
⟨*proof*⟩

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows *supp* (*Conj xset*) = *supp xset*
⟨*proof*⟩

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

⟨proof⟩

lemma *supp-Pred* [*simp*]: *supp* (*Pred f* φ) = *supp f* $\cup$ *supp* φ
⟨proof⟩

lemma *supp-Act* [*simp*]: *supp* (*Act f* α *x*) = *supp f* $\cup$ (*supp* α $\cup$ *supp x* $-$ *bn* α)
⟨proof⟩

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

lemma *Conj-eq-iff* [*simp*]:
assumes *finite* (*supp xset1*) **and** *finite* (*supp xset2*)
shows *Conj xset1* = *Conj xset2* $\longleftrightarrow$ *xset1* = *xset2*
⟨proof⟩

lemma *Not-eq-iff* [*simp*]: *Not x1* = *Not x2* $\longleftrightarrow$ *x1* = *x2*
⟨proof⟩

lemma *Pred-eq-iff* [*simp*]: *Pred f1* φ*1* = *Pred f2* φ*2* $\longleftrightarrow$ *f1* = *f2* $\wedge$ φ*1* = φ*2*
⟨proof⟩

lemma *Act-eq-iff*: *Act f1* α*1 x1* = *Act f2* α*2 x2* $\longleftrightarrow$ Act_α *f1* α*1* (*Rep-formula x1*) = Act_α *f2* α*2* (*Rep-formula x2*)
⟨proof⟩

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: *Act f1* α*1 x1* = *Act f2* α*2 x2* $\longleftrightarrow$
f1 = *f2* $\wedge$ ($\exists$ *p*. *supp x1* $-$ *bn* α*1* = *supp x2* $-$ *bn* α*2* $\wedge$ (*supp x1* $-$ *bn* α*1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ *x1* = *x2* $\wedge$ *supp* α*1* $-$ *bn* α*1* = *supp* α*2* $-$ *bn* α*2* $\wedge$ (*supp* α*1* $-$ *bn* α*1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ α*1* = α*2*)
(**is** *?l* $\longleftrightarrow$ *?r*)
⟨proof⟩

lemma *Act-eq-iff-perm-renaming*: *Act f1* α*1 x1* = *Act f2* α*2 x2* $\longleftrightarrow$
f1 = *f2* $\wedge$ ($\exists$ *p*. *supp x1* $-$ *bn* α*1* = *supp x2* $-$ *bn* α*2* $\wedge$ (*supp x1* $-$ *bn* α*1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ *x1* = *x2* $\wedge$ *supp* α*1* $-$ *bn* α*1* = *supp* α*2* $-$ *bn* α*2* $\wedge$ (*supp* α*1* $-$ *bn* α*1*) $\sharp*$ *p* $\wedge$ *p* $\cdot$ α*1* = α*2* $\wedge$ *supp p* $\subseteq$ *bn* α*1* $\cup$ *p* $\cdot$ *bn* α*1*)
(**is** *?l* $\longleftrightarrow$ *?r*)
⟨proof⟩

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

lemma *Tree-free* [*simp*]:
shows *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Not x*
and *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Pred f* φ
and *finite* (*supp xset*) $\Longrightarrow$ *Conj xset* $\neq$ *Act f* α *x*
and *Not x* $\neq$ *Pred f* φ
and *Not x1* $\neq$ *Act f* α *x2*
and *Pred f1* φ $\neq$ *Act f2* α *x*
⟨proof⟩

14.8 F/L-formulas

context *effect-nominal-ts*
begin

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

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

abbreviation *in-𝒜* :: (*'idx*,*'pred*,*'act*,*'effect*) *formula* ⇒ *'effect first* ⇒ *bool*
(‹- ∈ 𝒜[-]› [*51*,*0*] *50*) **where**
x ∈ 𝒜[*F*] ≡ *is-FL-formula F x*

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

lemma *is-FL-formula-eqvt* [*eqvt*]: *x* ∈ 𝒜[*F*] ⟹ *p* · *x* ∈ 𝒜[*p* · *F*]
⟨*proof*⟩

end

14.9 Induction over infinitary formulas

14.10 Strong induction over infinitary formulas

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

15 Validity With Effects

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

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

lemma *alpha-Tree-relI* [*simp*]:
assumes $x =_\alpha y$ **shows** (*x*,*y*) ∈ *alpha-Tree-rel*
⟨*proof*⟩

lemma *alpha-Tree-relE*:
  assumes $(x,y) \in$ *alpha-Tree-rel* **and** $x =_\alpha y \Longrightarrow P$
  shows P
⟨*proof*⟩

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
  wf (*alpha-Tree-rel* O *hull-rel* O *Tree-wf*)
⟨*proof*⟩

lemma *alpha-Tree-rel-relcomp-trivialI* [*simp*]:
  assumes $(x, y) \in R$
  shows $(x, y) \in$ *alpha-Tree-rel* O R
⟨*proof*⟩

lemma *alpha-Tree-rel-relcompI* [*simp*]:
  assumes $x =_\alpha x'$ **and** $(x', y) \in R$
  shows $(x, y) \in$ *alpha-Tree-rel* O R
⟨*proof*⟩

15.1 Validity for infinitely branching trees

context *effect-nominal-ts*
begin

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

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

  function (*sequential*) *FL-valid-Tree* :: *'state* ⇒ (*'idx*,*'pred*,*'act*,*'effect*) *Tree* ⇒ *bool* **where**
    FL-valid-Tree P (*tConj tset*) ⟷ (∀ *t*∈*set-bset tset*. *FL-valid-Tree P t*)
  | *FL-valid-Tree P* (*tNot t*) ⟷ ¬ *FL-valid-Tree P t*
  | *FL-valid-Tree P* (*tPred f* φ) ⟷ ⟨*f*⟩*P* ⊢ φ
  | *FL-valid-Tree P* (*tAct f* α *t*) ⟷ (∃ α' *t'* *P'*. *tAct f* α *t* $=_\alpha$ *tAct f* α' *t'* ∧ ⟨*f*⟩*P* → ⟨α',*P'*⟩ ∧ *FL-valid-Tree P' t'*)
  ⟨*proof*⟩
  termination ⟨*proof*⟩

FL-valid-Tree is equivariant.

  lemma *FL-valid-Tree-eqvt'*: *FL-valid-Tree P t* ⟷ *FL-valid-Tree* (*p* · *P*) (*p* · *t*)
  ⟨*proof*⟩

  lemma *FL-valid-Tree-eqvt* [*eqvt*]:
    assumes *FL-valid-Tree P t* **shows** *FL-valid-Tree* (*p* · *P*) (*p* · *t*)
  ⟨*proof*⟩

α-equivalent trees validate the same states.

lemma *alpha-Tree-FL-valid-Tree*:
assumes $t1 =_\alpha t2$
shows *FL-valid-Tree P t1* $\longleftrightarrow$ *FL-valid-Tree P t2*
⟨*proof*⟩

15.2 Validity for trees modulo α-equivalence

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

lemma $FL\text{-}valid\text{-}Tree_\alpha\text{-}eqvt$ [*eqvt*]:
assumes $FL\text{-}valid\text{-}Tree_\alpha$ *P t* **shows** $FL\text{-}valid\text{-}Tree_\alpha$ $(p \cdot P)$ $(p \cdot t)$
⟨*proof*⟩

lemma $FL\text{-}valid\text{-}Tree_\alpha\text{-}Conj_\alpha$ [*simp*]: $FL\text{-}valid\text{-}Tree_\alpha$ P $(Conj_\alpha\ tset_\alpha) \longleftrightarrow (\forall\, t_\alpha \in set\text{-}bset\ tset_\alpha.\ FL\text{-}valid\text{-}Tree_\alpha\ P\ t_\alpha)$
⟨*proof*⟩

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

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

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

15.3 Validity for infinitary formulas

lift-definition *FL-valid* :: *'state* $\Rightarrow$ (*'idx,'pred,'act,'effect*) *formula* $\Rightarrow$ *bool* (**infix** ‹$\models$› *70*) **is**
$FL\text{-}valid\text{-}Tree_\alpha$
⟨*proof*⟩

lemma *FL-valid-eqvt* [*eqvt*]:
assumes $P \models x$ **shows** $(p \cdot P) \models (p \cdot x)$
⟨*proof*⟩

lemma *FL-valid-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows $P \models Conj\ xset \longleftrightarrow (\forall\, x \in set\text{-}bset\ xset.\ P \models x)$
⟨*proof*⟩

lemma *FL-valid-Not* [*simp*]: $P \models \mathit{Not}\ x \longleftrightarrow \neg\ P \models x$
⟨*proof*⟩

lemma *FL-valid-Pred* [*simp*]: $P \models \mathit{Pred}\ f\ \varphi \longleftrightarrow \langle f\rangle P \vdash \varphi$
⟨*proof*⟩

lemma *FL-valid-Act*: $P \models \mathit{Act}\ f\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f\rangle P \rightarrow \langle \alpha',P'\rangle \wedge P' \models x')$
⟨*proof*⟩

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

lemma *FL-valid-Act-strong*:
assumes *finite* (*supp* X)
shows $P \models \mathit{Act}\ f\ \alpha\ x \longleftrightarrow (\exists \alpha'\ x'\ P'.\ \mathit{Act}\ f\ \alpha\ x = \mathit{Act}\ f\ \alpha'\ x' \wedge \langle f\rangle P \rightarrow \langle \alpha',P'\rangle \wedge P' \models x' \wedge \mathit{bn}\ \alpha' \sharp* X)$
⟨*proof*⟩

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

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 ⇒ 'pred::fs ⇒ bool* (**infix** ‹⊢› *70*)
and *transition* :: *'state ⇒ ('act::bn,'state) residual ⇒ bool* (**infix** ‹→› *70*)
and *effect-apply* :: *'effect::fs ⇒ 'state ⇒ 'state* (‹⟨-⟩-› [*0*,*101*] *100*) +
assumes *card-idx-perm*: $|\mathit{UNIV}::\mathit{perm\ set}| <o |\mathit{UNIV}::'\mathit{idx\ set}|$
and *card-idx-state*: $|\mathit{UNIV}::'\mathit{state\ set}| <o |\mathit{UNIV}::'\mathit{idx\ set}|$
begin

definition *FL-logically-equivalent* :: *'effect first ⇒ 'state ⇒ 'state ⇒ bool* **where**
FL-logically-equivalent F P Q ≡
$\forall x::('\mathit{idx},'\mathit{pred},'\mathit{act},'\mathit{effect})\ \mathit{formula}.\ x \in \mathcal{A}[F] \longrightarrow (P \models x \longleftrightarrow Q \models x)$

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

end

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

17 F/L-Bisimilarity Implies Logical Equivalence

context *indexed-effect-nominal-ts*
begin

lemma *FL-bisimilarity-implies-equivalence-Act*:
assumes $f \in_{fs} F$
and $bn\ \alpha\ \sharp*\ (F, f)$
and $x \in \mathcal{A}[L\ (\alpha, F, f)]$
and $\bigwedge P\ Q.\ P \sim\cdot[L\ (\alpha, F, f)]\ Q \Longrightarrow P \models x \longleftrightarrow Q \models x$
and $P \sim\cdot[F]\ Q$
and $P \models Act\ f\ \alpha\ x$
shows $Q \models Act\ f\ \alpha\ x$
⟨*proof*⟩

theorem *FL-bisimilarity-implies-equivalence*: **assumes** $P \sim\cdot[F]\ Q$ **shows** *FL-logically-equivalent* $F\ P\ Q$
⟨*proof*⟩

end

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

18 Logical Equivalence Implies F/L-Bisimilarity

context *indexed-effect-nominal-ts*
begin

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

lemma *is-distinguishing-formula-eqvt* :

assumes *x distinguishes P from Q* **shows** $(p \cdot x)$ *distinguishes* $(p \cdot P)$ *from* $(p \cdot Q)$
⟨*proof*⟩

lemma *FL-equivalent-iff-not-distinguished*:
FL-logically-equivalent F P Q $\longleftrightarrow \neg(\exists x.\ x \in \mathcal{A}[F] \wedge$ *x distinguishes P from Q*$)$
⟨*proof*⟩

There exists a distinguishing formula for P and Q in $\mathcal{A}[F]$ whose support is contained in *supp* $(F,\ P)$.

lemma *FL-distinguished-bounded-support*:
assumes $x \in \mathcal{A}[F]$ **and** *x distinguishes P from Q*
obtains *y* **where** $y \in \mathcal{A}[F]$ **and** *supp y* $\subseteq$ *supp* (F,P) **and** *y distinguishes P from Q*
⟨*proof*⟩

lemma *FL-equivalence-is-L-bisimulation*: *is-L-bisimulation FL-logically-equivalent*
⟨*proof*⟩

theorem *FL-equivalence-implies-bisimilarity*: **assumes** *FL-logically-equivalent F P Q* **shows** $P \sim\cdot[F]\ Q$
⟨*proof*⟩

end

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

19 *L*-Transform

19.1 States

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

datatype *('state,'effect) L-state* =
AC 'effect × *'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**
$p \cdot (AC\ x) = AC\ (p \cdot x)$

| $p \cdot (EF\ x) = EF\ (p \cdot x)$

instance
⟨*proof*⟩

end

declare *permute-L-state.simps* [*eqvt*]

lemma *supp-AC* [*simp*]: $supp\ (AC\ x) = supp\ x$
⟨*proof*⟩

lemma *supp-EF* [*simp*]: $supp\ (EF\ x) = supp\ x$
⟨*proof*⟩

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

instance
⟨*proof*⟩

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**
$p \cdot (Act\ \alpha) = Act\ (p \cdot \alpha)$
| $p \cdot (Eff\ f) = Eff\ (p \cdot f)$

instance
⟨*proof*⟩

end

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

lemma *supp-Act* [*simp*]: $supp\ (Act\ \alpha) = supp\ \alpha$
⟨*proof*⟩

lemma *supp-Eff* [*simp*]: $supp\ (Eff\ f) = supp\ f$
⟨*proof*⟩

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

instance
⟨*proof*⟩

end

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

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

instance
⟨*proof*⟩

end

19.3 Satisfaction

context *effect-nominal-ts*
begin

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

lemma *L-satisfies-eqvt*: **assumes** $P_L \vdash_L \varphi$ **shows** $(p \cdot P_L) \vdash_L (p \cdot \varphi)$
⟨*proof*⟩

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** ‹$\rightarrow_L$› *70*) **where**
AC (f,F,P) $\rightarrow_L \alpha P' \longleftrightarrow (\exists \alpha\ P'.\ P \rightarrow \langle\alpha,P'\rangle \wedge \alpha P' = \langle$*Act* α, *EF* (*L* (α,F,f), P')⟩ ∧ *bn* α ♯∗ (F,f)) — note the freshness condition
| *EF* (F,P) $\rightarrow_L \alpha P' \longleftrightarrow (\exists f.\ f \in_{fs} F \wedge \alpha P' = \langle$*Eff* f, *AC* (f, F, $\langle f\rangle P$)⟩)

lemma *L-transition-eqvt*: **assumes** $P_L \rightarrow_L \alpha_L P_L{}'$ **shows** $(p \cdot P_L) \rightarrow_L (p \cdot \alpha_L P_L{}')$
⟨*proof*⟩

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

lemma *L-transition-AC-strong*:
assumes *finite* (*supp X*) **and** $AC\ (f,F,P) \rightarrow_L \langle \alpha_L,P_L{}' \rangle$
shows $\exists\, \alpha\ P'.\ P \rightarrow \langle \alpha,P' \rangle \wedge \langle \alpha_L,P_L{}' \rangle = \langle Act\ \alpha,\ EF\ (L\ (\alpha,F,f),\ P') \rangle \wedge bn\ \alpha\ \sharp\!*\ X$
⟨*proof*⟩

lemma *L-transition-AC-fresh*:
assumes $bn\ \alpha\ \sharp\!*\ (F,f,P)$
shows $AC\ (f,F,P) \rightarrow_L \langle Act\ \alpha,\ P_L{}' \rangle \longleftrightarrow (\exists\, P'.\ P_L{}' = EF\ (L\ (\alpha,F,f),\ P') \wedge P \rightarrow \langle \alpha,P' \rangle)$
⟨*proof*⟩

end

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

Since we defined formulas via a manual quotient construction, we also need to define the L-transform via lifting from the underlying type of infinitely branching trees. As before, we cannot use **nominal_function** because that generates proof obligations where, for formulas of the form *FL-Formula.Conj xset*, the assumption that *xset* has finite support is missing.

The following auxiliary function returns trees (modulo α-equivalence) rather than formulas. This allows us to prove equivariance for *all* argument trees, without an assumption that they are (hereditarily) finitely supported. Further below–after this auxiliary function has been lifted to F/L-formulas as arguments–we derive a version that returns formulas.

primrec *L-transform-Tree* :: ('*idx*,'*pred*::*fs*,'*act*::*bn*,'*eff*::*fs*) *Tree* ⇒ ('*idx*, '*pred*, ('*act*,'*eff*) *L-action*) $Formula.Tree_\alpha$ **where**
L-transform-Tree (*tConj tset*) = $Formula.Conj_\alpha$ (*map-bset L-transform-Tree tset*)
| *L-transform-Tree* (*tNot t*) = $Formula.Not_\alpha$ (*L-transform-Tree t*)
| *L-transform-Tree* (*tPred f* φ) = $Formula.Act_\alpha$ (*Eff f*) ($Formula.Pred_\alpha$ φ)
| *L-transform-Tree* (*tAct f* α *t*) = $Formula.Act_\alpha$ (*Eff f*) ($Formula.Act_\alpha$ (*Act* α) (*L-transform-Tree t*))

lemma *L-transform-Tree-eqvt* [*eqvt*]: *p* • *L-transform-Tree t* = *L-transform-Tree* (*p* • *t*)
⟨*proof*⟩

L-transform-Tree respects α-equivalence.

lemma *alpha-Tree-L-transform-Tree*:
assumes *alpha-Tree t1 t2*
shows *L-transform-Tree t1* = *L-transform-Tree t2*

⟨*proof*⟩

L-transform for trees modulo α-equivalence.

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

lemma *L-transform-Tree*$_\alpha$*-eqvt* [*eqvt*]: $p \cdot$ *L-transform-Tree*$_\alpha$ t_α = *L-transform-Tree*$_\alpha$ ($p \cdot t_\alpha$)
⟨*proof*⟩

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*$_\alpha$)
⟨*proof*⟩

lemma *L-transform-Tree*$_\alpha$*-Not*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Not*$_\alpha$ t_α) = *Formula.Not*$_\alpha$ (*L-transform-Tree*$_\alpha$ t_α)
⟨*proof*⟩

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

lemma *L-transform-Tree*$_\alpha$*-Act*$_\alpha$ [*simp*]: *L-transform-Tree*$_\alpha$ (*Act*$_\alpha$ f α t_α) = *Formula.Act*$_\alpha$ (*Eff* f) (*Formula.Act*$_\alpha$ (*Act* α) (*L-transform-Tree*$_\alpha$ t_α))
⟨*proof*⟩

lemma *finite-supp-map-bset-L-transform-Tree*$_\alpha$ [*simp*]:
assumes *finite* (*supp tset*$_\alpha$)
shows *finite* (*supp* (*map-bset L-transform-Tree*$_\alpha$ *tset*$_\alpha$))
⟨*proof*⟩

lemma *L-transform-Tree*$_\alpha$*-preserves-hereditarily-fs*:
assumes *hereditarily-fs* t_α
shows *Formula.hereditarily-fs* (*L-transform-Tree*$_\alpha$ t_α)
⟨*proof*⟩

L-transform for F/L-formulas.

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

lemma *L-transform-formula-eqvt* [*eqvt*]: $p \cdot$ *L-transform-formula* x = *L-transform-formula* ($p \cdot x$)
⟨*proof*⟩

lemma *L-transform-formula-Conj* [*simp*]:

assumes *finite* (*supp xset*)
shows *L-transform-formula* (*Conj xset*) = *Formula.Conj*$_\alpha$ (*map-bset L-transform-formula xset*)
⟨*proof*⟩

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

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

lemma *L-transform-formula-Act* [*simp*]: *L-transform-formula* (*FL-Formula.Act f* α *x*) = *Formula.Act*$_\alpha$ (*Eff f*) (*Formula.Act*$_\alpha$ (*Act* α) (*L-transform-formula x*))
⟨*proof*⟩

lemma *L-transform-formula-hereditarily-fs* [*simp*]: *Formula.hereditarily-fs* (*L-transform-formula x*)
⟨*proof*⟩

Finally, we define the proper L-transform, which returns formulas instead of trees.

definition *L-transform* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*,*'eff*::*fs*) *formula* ⇒ (*'idx*, *'pred*, (*'act*,*'eff*) *L-action*) *Formula.formula* **where**
L-transform x = *Formula.Abs-formula* (*L-transform-formula x*)

lemma *L-transform-eqvt* [*eqvt*]: *p* · *L-transform x* = *L-transform* (*p* · *x*)
⟨*proof*⟩

lemma *finite-supp-map-bset-L-transform* [*simp*]:
assumes *finite* (*supp xset*)
shows *finite* (*supp* (*map-bset L-transform xset*))
⟨*proof*⟩

lemma *L-transform-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows *L-transform* (*Conj xset*) = *Formula.Conj* (*map-bset L-transform xset*)
⟨*proof*⟩

lemma *L-transform-Not* [*simp*]: *L-transform* (*Not x*) = *Formula.Not* (*L-transform x*)
⟨*proof*⟩

lemma *L-transform-Pred* [*simp*]: *L-transform* (*Pred f* φ) = *Formula.Act* (*Eff f*) (*Formula.Pred* φ)
⟨*proof*⟩

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

(*Eff f*) (*Formula.Act* (*Act* α) (*L-transform x*))
⟨*proof*⟩

context *effect-nominal-ts*
begin

interpretation *L-transform*: *nominal-ts* ($\vdash_L$) ($\rightarrow_L$)
⟨*proof*⟩

The L-transform preserves satisfaction of formulas in the following sense:

theorem *FL-valid-iff-valid-L-transform*:
assumes (*x*::(*'idx,'pred,'act,'effect*) *formula*) $\in \mathcal{A}[F]$
shows *FL-valid P x* $\longleftrightarrow$ *L-transform.valid* (*EF* (*F*, *P*)) (*L-transform x*)
⟨*proof*⟩

end

19.6 Bisimilarity in the L-transform

context *effect-nominal-ts*
begin

interpretation *L-transform*: *nominal-ts* ($\vdash_L$) ($\rightarrow_L$)
⟨*proof*⟩

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

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

inductive *L-bisimilar* :: (*'state,'effect*) *L-state* ⇒ (*'state,'effect*) *L-state* ⇒ *bool*
where
P $\sim\cdot[F]$ *Q* $\Longrightarrow$ *L-bisimilar* (*EF* (*F*,*P*)) (*EF* (*F*,*Q*))
| *P* $\sim\cdot[F]$ *Q* $\Longrightarrow$ $f \in_{fs} F$ $\Longrightarrow$ *L-bisimilar* (*AC* (*f*, *F*, ⟨*f*⟩*P*)) (*AC* (*f*, *F*, ⟨*f*⟩*Q*))

lemma *L-bisimilar-is-L-transform-bisimulation*: *L-transform.is-bisimulation L-bisimilar*
⟨*proof*⟩

definition *invL-FL-bisimilar* :: *'effect first* ⇒ *'state* ⇒ *'state* ⇒ *bool* **where**
invL-FL-bisimilar F P Q ≡ *EF* (*F*,*P*) $\sim\cdot_L$ *EF*(*F*,*Q*)

lemma *invL-FL-bisimilar-is-L-bisimulation*: *is-L-bisimulation invL-FL-bisimilar*
⟨*proof*⟩

theorem *P* $\sim\cdot[F]$ *Q* $\longleftrightarrow$ *EF* (*F*,*P*) $\sim\cdot_L$ *EF*(*F*,*Q*)
⟨*proof*⟩

end

The following (alternative) proof of the "←" direction of this equivalence, namely that bisimilarity in the L-transform implies F/L-bisimilarity, uses the fact that the L-transform preserves satisfaction of formulas, together with the fact that bisimilarity (in the L-transform) implies logical equivalence. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system with effects where, additionally, the cardinality of the state set of the L-transform is bounded. We could re-organize our formalization to remove this assumption: the proof of ⟦*indexed-nominal-ts TYPE(?'idx) ?satisfies ?transition*; *nominal-ts.bisimilar ?satisfies ?transition ?P ?Q*⟧ ⟹ *indexed-nominal-ts.logically-equivalent TYPE(?'idx) ?satisfies ?transition ?P ?Q* does not actually make use of the cardinality assumptions provided by indexed nominal transition systems.

locale *L-transform-indexed-effect-nominal-ts* = *indexed-effect-nominal-ts L satisfies transition effect-apply*
for *L* :: (*'act*::*bn*) × (*'effect*::*fs*) *fs-set* × *'effect* ⇒ *'effect fs-set*
and *satisfies* :: *'state*::*fs* ⇒ *'pred*::*fs* ⇒ *bool* (**infix** ‹⊢› *70*)
and *transition* :: *'state* ⇒ (*'act*,*'state*) *residual* ⇒ *bool* (**infix** ‹→› *70*)
and *effect-apply* :: *'effect* ⇒ *'state* ⇒ *'state* (‹⟨-⟩-› [*0*,*101*] *100*) +
assumes *card-idx-L-transform-state*: |*UNIV*::(*'state*, *'effect*) *L-state set*| <*o* |*UNIV*::*'idx set*|
begin

interpretation *L-transform*: *indexed-nominal-ts* ($\vdash_L$) ($\rightarrow_L$)
⟨*proof*⟩

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

theorem *EF* (*F*,*P*) $\sim\cdot_L$ *EF*(*F*,*Q*) ⟶ *P* ∼·[*F*] *Q*
⟨*proof*⟩

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*]: $p \cdot \tau = \tau$
begin

lemma *bn-tau-empty* [*simp*]: $bn\ \tau = \{\}$
⟨*proof*⟩

lemma *bn-tau-fresh* [*simp*]: $bn\ \tau\ \sharp *\ P$
⟨*proof*⟩

inductive *tau-transition* :: $'state \Rightarrow 'state \Rightarrow bool$ (**infix** ‹⇒› *70*) **where**
tau-refl [*simp*]: $P \Rightarrow P$
| *tau-step*: $[\![P \rightarrow \langle\tau, P'\rangle; P' \Rightarrow P'']\!] \Longrightarrow P \Rightarrow P''$

definition *observable-transition* :: $'state \Rightarrow 'act \Rightarrow 'state \Rightarrow bool$ (‹-/ ⇒{-}/ -› [*70*, *70*, *71*] *71*) **where**
$P \Rightarrow\{\alpha\}\ P' \equiv \exists\, Q\ Q'.\ P \Rightarrow Q \wedge Q \rightarrow \langle\alpha, Q'\rangle \wedge Q' \Rightarrow P'$

definition *weak-transition* :: $'state \Rightarrow 'act \Rightarrow 'state \Rightarrow bool$ (‹-/ ⇒⟨-⟩/ -› [*70*, *70*, *71*] *71*) **where**
$P \Rightarrow\langle\alpha\rangle\ P' \equiv if\ \alpha = \tau\ then\ P \Rightarrow P'\ else\ P \Rightarrow\{\alpha\}\ P'$

The transition relations defined above are equivariant.

lemma *tau-transition-eqvt* :
assumes $P \Rightarrow P'$ **shows** $p \cdot P \Rightarrow p \cdot P'$
⟨*proof*⟩

lemma *observable-transition-eqvt* :
assumes $P \Rightarrow\{\alpha\}\ P'$ **shows** $p \cdot P \Rightarrow\{p \cdot \alpha\}\ p \cdot P'$
⟨*proof*⟩

lemma *weak-transition-eqvt* :
assumes $P \Rightarrow\langle\alpha\rangle\ P'$ **shows** $p \cdot P \Rightarrow\langle p \cdot \alpha\rangle\ p \cdot P'$
⟨*proof*⟩

Additional lemmas about (⇒), *observable-transition* and *weak-transition*.

lemma *tau-transition-trans*:
assumes $P \Rightarrow Q$ **and** $Q \Rightarrow R$
shows $P \Rightarrow R$
⟨*proof*⟩

lemma *observable-transitionI*:
assumes $P \Rightarrow Q$ **and** $Q \rightarrow \langle\alpha, Q'\rangle$ **and** $Q' \Rightarrow P'$
shows $P \Rightarrow\{\alpha\}\ P'$
⟨*proof*⟩

lemma *observable-transition-stepI* [*simp*]:
assumes $P \rightarrow \langle\alpha, P'\rangle$
shows $P \Rightarrow\{\alpha\}\ P'$
⟨*proof*⟩

lemma *observable-tau-transition*:
  assumes $P \Rightarrow\{\tau\}\ P'$
  shows $P \Rightarrow\ P'$
⟨*proof*⟩

lemma *weak-transition-tau-iff* [*simp*]:
  $P \Rightarrow\langle\tau\rangle\ P' \longleftrightarrow P \Rightarrow\ P'$
⟨*proof*⟩

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

lemma *weak-transition-stepI* [*simp*]:
  assumes $P \Rightarrow\{\alpha\}\ P'$
  shows $P \Rightarrow\langle\alpha\rangle\ P'$
⟨*proof*⟩

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*⟩

end

20.2 Weak bisimulations

context *weak-nominal-ts*
begin

definition *is-weak-bisimulation* :: $('state \Rightarrow 'state \Rightarrow bool) \Rightarrow bool$ **where**
  is-weak-bisimulation $R \equiv$
    symp $R\ \wedge$
    — weak static implication
    $(\forall P\ Q\ \varphi.\ R\ P\ Q \wedge P \vdash \varphi \longrightarrow (\exists Q'.\ Q \Rightarrow\ Q' \wedge R\ P\ Q' \wedge Q' \vdash \varphi)) \wedge$
    — weak simulation
    $(\forall P\ Q.\ R\ P\ Q \longrightarrow (\forall \alpha\ P'.\ bn\ \alpha\ \sharp{*}\ Q \longrightarrow P \to \langle\alpha,P'\rangle \longrightarrow (\exists Q'.\ Q \Rightarrow\langle\alpha\rangle$
$Q' \wedge R\ P'\ Q')))$

definition *weakly-bisimilar* :: $'state \Rightarrow 'state \Rightarrow bool$ (**infix** ‹≈·› *100*) **where**
  $P \approx\cdot\ Q \equiv \exists R.$ *is-weak-bisimulation* $R \wedge R\ P\ Q$

$(\approx\cdot)$ is an equivariant equivalence relation.

lemma *is-weak-bisimulation-eqvt* :
  assumes *is-weak-bisimulation* R **shows** *is-weak-bisimulation* $(p \cdot R)$
⟨*proof*⟩

lemma *weakly-bisimilar-eqvt* :
assumes $P \approx\cdot Q$ **shows** $(p \cdot P) \approx\cdot (p \cdot Q)$
⟨*proof*⟩

lemma *weakly-bisimilar-reflp*: *reflp weakly-bisimilar*
⟨*proof*⟩

lemma *weakly-bisimilar-symp*: *symp weakly-bisimilar*
⟨*proof*⟩

lemma *weakly-bisimilar-is-weak-bisimulation*: *is-weak-bisimulation weakly-bisimilar*
⟨*proof*⟩

lemma *weakly-bisimilar-tau-simulation-step*:
assumes $P \approx\cdot Q$ **and** $P \Rightarrow P'$
obtains Q' **where** $Q \Rightarrow Q'$ **and** $P' \approx\cdot Q'$
⟨*proof*⟩

lemma *weakly-bisimilar-weak-simulation-step*:
assumes $P \approx\cdot Q$ **and** $bn\ \alpha \sharp* Q$ **and** $P \Rightarrow\langle\alpha\rangle\ P'$
obtains Q' **where** $Q \Rightarrow\langle\alpha\rangle\ Q'$ **and** $P' \approx\cdot Q'$
⟨*proof*⟩

lemma *weakly-bisimilar-transp*: *transp weakly-bisimilar*
⟨*proof*⟩

lemma *weakly-bisimilar-equivp*: *equivp weakly-bisimilar*
⟨*proof*⟩

end

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

21 Weak Formulas

21.1 Lemmas about α-equivalence involving τ

context *weak-nominal-ts*
begin

lemma *Act-tau-eq-iff* [*simp*]:
$Act\ \tau\ x1 = Act\ \alpha\ x2 \longleftrightarrow \alpha = \tau \wedge x2 = x1$
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

end

21.2 Weak action modality

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

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

locale *indexed-weak-nominal-ts* = *weak-nominal-ts satisfies transition*
  for *satisfies* :: *'state::fs* ⇒ *'pred::fs* ⇒ *bool* (**infix** ‹⊢› *70*)
  and *transition* :: *'state* ⇒ (*'act::bn*,*'state*) *residual* ⇒ *bool* (**infix** ‹→› *70*) +
  assumes *card-idx-perm*: |*UNIV::perm set*| <*o* |*UNIV::'idx set*|
    and *card-idx-state*: |*UNIV::'state set*| <*o* |*UNIV::'idx set*|
    and *card-idx-nat*: |*UNIV::nat set*| <*o* |*UNIV::'idx set*|
begin

The assumption |*UNIV*| <*o* |*UNIV*| is redundant: it is already implied by |*UNIV*| <*o* |*UNIV*|. A formal proof of this fact is left for future work.

lemma *card-idx-nat'* [*simp*]:
  |*UNIV::nat set*| <*o natLeq* +*c* |*UNIV::'idx set*|
⟨*proof*⟩

primrec *tau-steps* :: (*'idx*,*'pred::fs*,*'act::bn*) *formula* ⇒ *nat* ⇒ (*'idx*,*'pred*,*'act*) *formula*
  where
    tau-steps x 0 = *x*
  | *tau-steps x* (*Suc n*) = *Act* τ (*tau-steps x n*)

lemma *tau-steps-eqvt* [*simp*]:
  p · *tau-steps x n* = *tau-steps* (*p* · *x*) (*p* · *n*)
  ⟨*proof*⟩

lemma *tau-steps-eqvt'* [*simp*]:
  p · *tau-steps x* = *tau-steps* (*p* · *x*)
  ⟨*proof*⟩

lemma *tau-steps-eqvt-raw* [*simp*]:
  p · *tau-steps* = *tau-steps*
  ⟨*proof*⟩

lemma *tau-steps-add* [*simp*]:
  tau-steps (*tau-steps x m*) *n* = *tau-steps x* (*m* + *n*)
  ⟨*proof*⟩

lemma *tau-steps-not-self*:
  x = *tau-steps x n* ⟷ *n* = *0*
⟨*proof*⟩

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

lemma *finite-supp-map-bset-tau-steps* [*simp*]:
finite (*supp* (*map-bset* (*tau-steps x*) (*Abs-bset UNIV* :: *nat set*[*'idx*])))
⟨*proof*⟩

lemma *weak-tau-modality-eqvt* [*simp*]:
p · *weak-tau-modality x* = *weak-tau-modality* (*p* · *x*)
⟨*proof*⟩

lemma *weak-tau-modality-eq-iff* [*simp*]:
weak-tau-modality x = *weak-tau-modality y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *supp-weak-tau-modality* [*simp*]:
supp (*weak-tau-modality x*) = *supp x*
⟨*proof*⟩

lemma *Act-weak-tau-modality-eq-iff* [*simp*]:
Act α*1* (*weak-tau-modality x1*) = *Act* α*2* (*weak-tau-modality x2*) ⟷ *Act* α*1 x1* = *Act* α*2 x2*
⟨*proof*⟩

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

lemma *weak-action-modality-eqvt* [*simp*]:
p · (⟨⟨α⟩⟩*x*) = ⟨⟨*p* · α⟩⟩(*p* · *x*)
⟨*proof*⟩

lemma *weak-action-modality-tau*:
(⟨⟨τ⟩⟩*x*) = *weak-tau-modality x*
⟨*proof*⟩

lemma *weak-action-modality-not-tau*:
assumes $\alpha \neq \tau$
shows (⟨⟨α⟩⟩*x*) = *weak-tau-modality* (*Act* α (*weak-tau-modality x*))
⟨*proof*⟩

Equality is modulo α-equivalence.

Note that the converse of the following lemma does not hold. For instance, for $\alpha \neq \tau$ we have ⟨⟨τ⟩⟩*Act* α (*weak-tau-modality x*) = ⟨⟨α⟩⟩*x* by definition, but clearly not *Act* τ (*Act* α (*weak-tau-modality x*)) = *Act* α *x*.

lemma *weak-action-modality-eq*:
assumes *Act* $\alpha 1$ *x1* = *Act* $\alpha 2$ *x2*
shows $(\langle\langle\alpha 1\rangle\rangle x1) = (\langle\langle\alpha 2\rangle\rangle x2)$
⟨*proof*⟩

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* $(\langle\langle\alpha\rangle\rangle x)$
| *wf-Pred*: *weak-formula x* ⟹ *weak-formula* $(\langle\langle\tau\rangle\rangle$(*Conj* (*binsert* (*Pred* φ) (*bsingleton x*))))

lemma *finite-supp-wf-Pred* [*simp*]: *finite* (*supp* (*binsert* (*Pred* φ) (*bsingleton x*)))
⟨*proof*⟩

weak-formula is equivariant.

lemma *weak-formula-eqvt* [*simp*]: *weak-formula x* ⟹ *weak-formula* (*p* · *x*)
⟨*proof*⟩

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 ⊨ *weak-tau-modality x* ⟷ (∃ *n*. *P* ⊨ *tau-steps x n*)
⟨*proof*⟩

lemma *tau-steps-iff-tau-transition*:
(∃ *n*. *P* ⊨ *tau-steps x n*) ⟷ (∃ *P'*. *P* ⇒ *P'* ∧ *P'* ⊨ *x*)
⟨*proof*⟩

lemma *valid-weak-tau-modality*:
$P \models$ *weak-tau-modality* $x \longleftrightarrow (\exists P'. P \Rightarrow P' \wedge P' \models x)$
⟨*proof*⟩

lemma *valid-weak-action-modality*:
$P \models (\langle\langle\alpha\rangle\rangle x) \longleftrightarrow (\exists \alpha'\ x'\ P'.\ Act\ \alpha\ x = Act\ \alpha'\ x' \wedge P \Rightarrow\langle\alpha'\rangle\ P' \wedge P' \models x')$
(**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

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

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

lemma *valid-weak-action-modality-fresh*:
assumes $bn\ \alpha \sharp* P$
shows $P \models (\langle\langle\alpha\rangle\rangle x) \longleftrightarrow (\exists P'.\ P \Rightarrow\langle\alpha\rangle\ P' \wedge P' \models x)$
⟨*proof*⟩

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

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

lemma *weakly-logically-equivalent-eqvt*:
assumes $P \equiv\cdot Q$ **shows** $p \cdot P \equiv\cdot p \cdot Q$
⟨*proof*⟩

end

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

24 Weak Bisimilarity Implies Weak Logical Equivalence

context *indexed-weak-nominal-ts*
begin

lemma *weak-bisimilarity-implies-weak-equivalence-Act*:
assumes $\bigwedge P\ Q.\ P \approx\cdot\ Q \Longrightarrow P \models x \longleftrightarrow Q \models x$
and $P \approx\cdot\ Q$
— not needed: and *weak-formula x*
and $P \models \langle\langle\alpha\rangle\rangle x$
shows $Q \models \langle\langle\alpha\rangle\rangle x$
⟨*proof*⟩

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

theorem *weak-bisimilarity-implies-weak-equivalence*: **assumes** $P \approx\cdot\ Q$ **shows** $P \equiv\cdot\ Q$
⟨*proof*⟩

end

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

25 Weak Logical Equivalence Implies Weak Bisimilarity

context *indexed-weak-nominal-ts*

begin

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

lemma *is-distinguishing-formula-eqvt* [*simp*]:
assumes *x distinguishes P from Q* **shows** $(p \cdot x)$ *distinguishes* $(p \cdot P)$ *from* $(p \cdot Q)$
⟨*proof*⟩

lemma *weakly-equivalent-iff-not-distinguished*: $(P \equiv\cdot\, Q) \longleftrightarrow \neg(\exists\, x.$ *weak-formula* $x \wedge x$ *distinguishes P from Q*)
⟨*proof*⟩

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

lemma *distinguished-bounded-support*:
assumes *weak-formula x* **and** *x distinguishes P from Q*
obtains *y* **where** *weak-formula y* **and** *supp y* $\subseteq$ *supp P* **and** *y distinguishes P from Q*
⟨*proof*⟩

lemma *weak-equivalence-is-weak-bisimulation*: *is-weak-bisimulation weakly-logically-equivalent*
⟨*proof*⟩

theorem *weak-equivalence-implies-weak-bisimilarity*: **assumes** $P \equiv\cdot\, Q$ **shows** $P \approx\cdot\, Q$
⟨*proof*⟩

end

end
theory *Weak-Expressive-Completeness*
imports
Weak-Bisimilarity-Implies-Equivalence
Weak-Equivalence-Implies-Bisimilarity
Disjunction
begin

26 Weak Expressive Completeness

context *indexed-weak-nominal-ts*
begin

26.1 Distinguishing weak formulas

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

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

definition *distinguishing-weak-formula* :: *'state* ⇒ *'state* ⇒ (*'idx*, *'pred*, *'act*) *formula* **where**
distinguishing-weak-formula P Q ≡ *Conj* (*Abs-bset* {−*p* · (ϵ *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* · (ϵ *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*⟩
lemma *distinguishing-weak-formula-supp-aux*:
assumes ¬ (*P* ≡· *Q*)
shows *supp* (*Abs-bset* {−*p* · (ϵ *x*. *weak-formula x* ∧ *supp x* ⊆ *supp* (*p* · *P*) ∧ *x distinguishes* (*p* · *P*) *from* (*p* · *Q*))|*p*. *True*} :: - *set*[*'idx*]) ⊆ *supp P*
⟨*proof*⟩

lemma *distinguishing-weak-formula-eqvt* [*simp*]:
assumes ¬ (*P* ≡· *Q*)
shows *p* · *distinguishing-weak-formula P Q* = *distinguishing-weak-formula* (*p* · *P*) (*p* · *Q*)
⟨*proof*⟩

lemma *supp-distinguishing-weak-formula*:
assumes ¬ (*P* ≡· *Q*)
shows *supp* (*distinguishing-weak-formula P Q*) ⊆ *supp P*
⟨*proof*⟩

lemma *distinguishing-weak-formula-distinguishes*:
assumes ¬ (*P* ≡· *Q*)
shows (*distinguishing-weak-formula P Q*) *distinguishes P from Q*
⟨*proof*⟩

lemma *distinguishing-weak-formula-is-weak*:
assumes ¬ (*P* ≡· *Q*)
shows *weak-formula* (*distinguishing-weak-formula P Q*)
⟨*proof*⟩

26.2 Characteristic weak formulas

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

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

— just an auxiliary lemma that will be useful further below
lemma *characteristic-weak-formula-card-aux*:
|{*distinguishing-weak-formula* P $Q|Q.\ \neg\ (P \equiv\cdot\ Q)$}| $<o$ *natLeq* $+c$ |*UNIV* :: $'idx$ *set*|
⟨*proof*⟩
lemma *characteristic-weak-formula-supp-aux*:
shows *supp* (*Abs-bset* {*distinguishing-weak-formula* P $Q|Q.\ \neg\ (P \equiv\cdot\ Q)$} :: -*set*[$'idx$]) $\subseteq$ *supp* P
⟨*proof*⟩

lemma *characteristic-weak-formula-eqvt* [*simp*]:
$p \cdot$ *characteristic-weak-formula* P = *characteristic-weak-formula* $(p \cdot P)$
⟨*proof*⟩

lemma *characteristic-weak-formula-eqvt-raw* [*simp*]:
$p \cdot$ *characteristic-weak-formula* = *characteristic-weak-formula*
⟨*proof*⟩

lemma *characteristic-weak-formula-is-weak*:
weak-formula (*characteristic-weak-formula* P)
⟨*proof*⟩

lemma *characteristic-weak-formula-is-characteristic'*:
$Q \models$ *characteristic-weak-formula* $P \longleftrightarrow P \equiv\cdot\ Q$
⟨*proof*⟩

lemma *characteristic-weak-formula-is-characteristic*:
$Q \models$ *characteristic-weak-formula* $P \longleftrightarrow P \approx\cdot\ Q$
⟨*proof*⟩

26.3 Weak expressive completeness

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

theorem *weak-expressive-completeness*:
assumes *finite* (*supp* S)
and $\bigwedge P\ Q.\ P \in S \Longrightarrow P \approx\cdot\ Q \Longrightarrow Q \in S$
shows $P \models$ *Disj* (*Abs-bset* (*characteristic-weak-formula* ‘ S)) $\longleftrightarrow P \in S$

and *weak-formula* (*Disj* (*Abs-bset* (*characteristic-weak-formula* ‘ *S*)))
⟨*proof*⟩

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**
$p \cdot (Act\ \alpha) = Act\ (p \cdot \alpha)$
| $p \cdot (Pred\ \varphi) = Pred\ (p \cdot \varphi)$

instance
⟨*proof*⟩

end

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

lemma *supp-Act* [*simp*]: $supp\ (Act\ \alpha) = supp\ \alpha$
⟨*proof*⟩

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

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

instance
⟨*proof*⟩

end

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

fun *bn-S-action* :: ($'a$,$'b$) *S-action* ⇒ *atom set* **where**
bn-S-action (*Act* α) = *bn* α
| *bn-S-action* (*Pred* -) = {}

instance
⟨*proof*⟩

end

27.2 Satisfaction

context *nominal-ts*
begin

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

definition *S-satisfies* :: $'state \Rightarrow unit \Rightarrow bool$ (**infix** ‹$\vdash_S$› *70*) **where**
$P \vdash_S \varphi \longleftrightarrow False$

lemma *S-satisfies-eqvt*: **assumes** $P \vdash_S \varphi$ **shows** $(p \cdot P) \vdash_S (p \cdot \varphi)$
⟨*proof*⟩

end

27.3 Transitions

context *nominal-ts*
begin

inductive *S-transition* :: $'state \Rightarrow ((' act,' pred)$ *S-action*, $'state)$ *residual* ⇒ *bool* (**infix** ‹$\rightarrow_S$› *70*) **where**
Act: $P \rightarrow \langle\alpha,P'\rangle \Longrightarrow P \rightarrow_S \langle Act\ \alpha,P'\rangle$
| *Pred*: $P \vdash \varphi \Longrightarrow P \rightarrow_S \langle Pred\ \varphi,P\rangle$

lemma *S-transition-eqvt*: **assumes** $P \rightarrow_S \alpha_S P'$ **shows** $(p \cdot P) \rightarrow_S (p \cdot \alpha_S P')$
⟨*proof*⟩

If there is an S-transition, there is an ordinary transition with the same residual—it is not necessary to consider alpha-variants.

lemma *S-transition-cases* [*case-names Act Pred*, *consumes 1*]: **assumes** $P \rightarrow_S \langle\alpha_S,P'\rangle$

and $\bigwedge\alpha.\ \alpha_S = \mathit{Act}\ \alpha \Longrightarrow P \to \langle\alpha,P'\rangle \Longrightarrow R$
and $\bigwedge\varphi.\ \alpha_S = \mathit{Pred}\ \varphi \Longrightarrow P' = P \Longrightarrow P \vdash \varphi \Longrightarrow R$
shows R
⟨*proof*⟩

lemma *S-transition-Act-iff*: $P \to_S \langle \mathit{Act}\ \alpha,P'\rangle \longleftrightarrow P \to \langle\alpha,P'\rangle$
⟨*proof*⟩

lemma *S-transition-Pred-iff*: $P \to_S \langle \mathit{Pred}\ \varphi,P'\rangle \longleftrightarrow P' = P \wedge P \vdash \varphi$
⟨*proof*⟩

end

27.4 Strong Bisimilarity in the S-transform

context *nominal-ts*
begin

interpretation *S-transform*: *nominal-ts* $(\vdash_S)$ $(\to_S)$
⟨*proof*⟩

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

notation *S-transform.bisimilar* (**infix** ‹$\sim\cdot_S$› *100*)

Bisimilarity is equivalent to bisimilarity in the S-transform.

lemma *bisimilar-is-S-transform-bisimulation*: *S-transform.is-bisimulation bisimilar*
⟨*proof*⟩

lemma *S-transform-bisimilar-is-bisimulation*: *is-bisimulation S-transform.bisimilar*
⟨*proof*⟩

theorem *S-transform-bisimilar-iff*: $P \sim\cdot_S Q \longleftrightarrow P \sim\cdot\ Q$
⟨*proof*⟩

end

27.5 Weak Bisimilarity in the S-transform

context *weak-nominal-ts*
begin

lemma *weakly-bisimilar-tau-transition-weakly-bisimilar*:
assumes $P \approx\cdot\ R$ **and** $P \Rightarrow Q$ **and** $Q \Rightarrow R$
shows $Q \approx\cdot\ R$
⟨*proof*⟩

notation *S-satisfies* (**infix** ‹$\vdash_S$› *70*)

interpretation *S-transform*: *weak-nominal-ts* ($\vdash_S$) ($\rightarrow_S$) *Act* τ
⟨*proof*⟩

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

notation *S-transform.tau-transition* (**infix** ‹$\Rightarrow_S$› *70*)
notation *S-transform.observable-transition* (‹-/ $\Rightarrow${-}$_S$/ -› [*70*, *70*, *71*] *71*)
notation *S-transform.weak-transition* (‹-/ $\Rightarrow\langle$-$\rangle_S$/ -› [*70*, *70*, *71*] *71*)
notation *S-transform.weakly-bisimilar* (**infix** ‹$\approx\cdot_S$› *100*)

lemma *S-transform-tau-transition-iff*: $P \Rightarrow_S P' \longleftrightarrow P \Rightarrow P'$
⟨*proof*⟩

lemma *S-transform-observable-transition-iff*: $P \Rightarrow\{Act\ \alpha\}_S\ P' \longleftrightarrow P \Rightarrow\{\alpha\}\ P'$
⟨*proof*⟩

lemma *S-transform-weak-transition-iff*: $P \Rightarrow\langle Act\ \alpha\rangle_S\ P' \longleftrightarrow P \Rightarrow\langle\alpha\rangle\ P'$
⟨*proof*⟩

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

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

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

theorem *S-transform-weakly-bisimilar-iff*: $P \approx\cdot_S Q \longleftrightarrow P \approx\cdot Q$
⟨*proof*⟩

end

27.6 Translation of (strong) formulas into formulas without predicates

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

The following auxiliary function returns trees (modulo α-equivalence) rather than formulas. This allows us to prove equivariance for *all* argument trees, without an assumption that they are (hereditarily) finitely supported. Further below–after this auxiliary function has been lifted to (strong) formulas as arguments–we derive a version that returns formulas.

primrec *S-transform-Tree* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*) *Tree* ⇒ (*'idx*, *unit*, (*'act*,*'pred*) *S-action*) $Tree_\alpha$ **where**
S-transform-Tree (*tConj tset*) = $Conj_\alpha$ (*map-bset S-transform-Tree tset*)
| *S-transform-Tree* (*tNot t*) = Not_α (*S-transform-Tree t*)
| *S-transform-Tree* (*tPred* φ) = Act_α (*S-action.Pred* φ) ($Conj_\alpha$ *bempty*)
| *S-transform-Tree* (*tAct* α *t*) = Act_α (*S-action.Act* α) (*S-transform-Tree t*)

lemma *S-transform-Tree-eqvt* [*eqvt*]: *p* • *S-transform-Tree t* = *S-transform-Tree* (*p* • *t*)
⟨*proof*⟩

S-transform-Tree respects α-equivalence.

lemma *alpha-Tree-S-transform-Tree*:
assumes *t1* $=_\alpha$ *t2*
shows *S-transform-Tree t1* = *S-transform-Tree t2*
⟨*proof*⟩

S-transform for trees modulo α-equivalence.

lift-definition $S\text{-}transform\text{-}Tree_\alpha$ :: (*'idx*,*'pred*::*fs*,*'act*::*bn*) $Tree_\alpha$ ⇒ (*'idx*, *unit*, (*'act*,*'pred*) *S-action*) $Tree_\alpha$ **is**
S-transform-Tree
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}eqvt$ [*eqvt*]: *p* • $S\text{-}transform\text{-}Tree_\alpha$ t_α = $S\text{-}transform\text{-}Tree_\alpha$ (*p* • t_α)
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}Conj_\alpha$ [*simp*]: $S\text{-}transform\text{-}Tree_\alpha$ ($Conj_\alpha$ $tset_\alpha$) = $Conj_\alpha$ (*map-bset* $S\text{-}transform\text{-}Tree_\alpha$ $tset_\alpha$)
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}Not_\alpha$ [*simp*]: $S\text{-}transform\text{-}Tree_\alpha$ (Not_α t_α) = Not_α ($S\text{-}transform\text{-}Tree_\alpha$ t_α)
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}Pred_\alpha$ [*simp*]: $S\text{-}transform\text{-}Tree_\alpha$ ($Pred_\alpha$ φ) = Act_α (*S-action.Pred* φ) ($Conj_\alpha$ *bempty*)
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}Act_\alpha$ [*simp*]: $S\text{-}transform\text{-}Tree_\alpha$ (Act_α α t_α) = Act_α (*S-action.Act* α) ($S\text{-}transform\text{-}Tree_\alpha$ t_α)
⟨*proof*⟩

lemma $finite\text{-}supp\text{-}map\text{-}bset\text{-}S\text{-}transform\text{-}Tree_\alpha$ [*simp*]:
assumes *finite* (*supp* $tset_\alpha$)
shows *finite* (*supp* (*map-bset* $S\text{-}transform\text{-}Tree_\alpha$ $tset_\alpha$))
⟨*proof*⟩

lemma $S\text{-}transform\text{-}Tree_\alpha\text{-}preserves\text{-}hereditarily\text{-}fs$:

assumes *hereditarily-fs* t_α
shows *hereditarily-fs* (*S-transform-Tree*$_\alpha$ t_α)
⟨*proof*⟩

S-transform for (strong) formulas.

lift-definition *S-transform-formula* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*) *formula* ⇒ (*'idx*, *unit*, (*'act*,*'pred*) *S-action*) *Tree*$_\alpha$ **is**
S-transform-Tree$_\alpha$
⟨*proof*⟩

lemma *S-transform-formula-eqvt* [*eqvt*]: *p* · *S-transform-formula x* = *S-transform-formula* (*p* · *x*)
⟨*proof*⟩

lemma *S-transform-formula-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows *S-transform-formula* (*Conj xset*) = *Conj*$_\alpha$ (*map-bset S-transform-formula xset*)
⟨*proof*⟩

lemma *S-transform-formula-Not* [*simp*]: *S-transform-formula* (*Not x*) = *Not*$_\alpha$ (*S-transform-formula x*)
⟨*proof*⟩

lemma *S-transform-formula-Pred* [*simp*]: *S-transform-formula* (*Formula.Pred* φ) = *Act*$_\alpha$ (*S-action.Pred* φ) (*Conj*$_\alpha$ *bempty*)
⟨*proof*⟩

lemma *S-transform-formula-Act* [*simp*]: *S-transform-formula* (*Formula.Act* α *x*) = *Formula.Act*$_\alpha$ (*S-action.Act* α) (*S-transform-formula x*)
⟨*proof*⟩

lemma *S-transform-formula-hereditarily-fs* [*simp*]: *hereditarily-fs* (*S-transform-formula x*)
⟨*proof*⟩

Finally, we define the proper *S*-transform, which returns formulas instead of trees.

definition *S-transform* :: (*'idx*,*'pred*::*fs*,*'act*::*bn*) *formula* ⇒ (*'idx*, *unit*, (*'act*,*'pred*) *S-action*) *formula* **where**
S-transform x = *Abs-formula* (*S-transform-formula x*)

lemma *S-transform-eqvt* [*eqvt*]: *p* · *S-transform x* = *S-transform* (*p* · *x*)
⟨*proof*⟩

lemma *finite-supp-map-bset-S-transform* [*simp*]:
assumes *finite* (*supp xset*)
shows *finite* (*supp* (*map-bset S-transform xset*))
⟨*proof*⟩

lemma *S-transform-Conj* [*simp*]:
  assumes *finite* (*supp xset*)
  shows *S-transform* (*Conj xset*) = *Conj* (*map-bset S-transform xset*)
  ⟨*proof*⟩

lemma *S-transform-Not* [*simp*]: *S-transform* (*Not x*) = *Not* (*S-transform x*)
  ⟨*proof*⟩

lemma *S-transform-Pred* [*simp*]: *S-transform* (*Formula.Pred* φ) = *Formula.Act* (*S-action.Pred* φ) (*Conj bempty*)
  ⟨*proof*⟩

lemma *S-transform-Act* [*simp*]: *S-transform* (*Formula.Act* α *x*) = *Formula.Act* (*S-action.Act* α) (*S-transform x*)
  ⟨*proof*⟩

context *nominal-ts*
begin

  lemma *valid-Conj-bempty* [*simp*]: *P* ⊨ *Conj bempty*
  ⟨*proof*⟩

  notation *S-satisfies* (**infix** ‹$\vdash_S$› *70*)

  interpretation *S-transform*: *nominal-ts* ($\vdash_S$) ($\rightarrow_S$)
  ⟨*proof*⟩

  notation *S-transform.valid* (**infix** ‹$\models_S$› *70*)

The *S*-transform preserves satisfaction of formulas in the following sense:

  theorem *valid-iff-valid-S-transform*: **shows** *P* ⊨ *x* ⟷ *P* $\models_S$ *S-transform x*
  ⟨*proof*⟩

end

context *indexed-nominal-ts*
begin

The following (alternative) proof of the "→" direction of theorem *nominal-ts.bisimilar* ($\vdash_S$) ($\rightarrow_S$) *?P ?Q* = *?P* ∼· *?Q*, namely that bisimilarity in the *S*-transform implies bisimilarity in the original transition system, uses the fact that the *S*-transform(ation) preserves satisfaction of formulas, together with the fact that bisimilarity (in the *S*-transform) implies logical equivalence, and equivalence (in the original transition system) implies bisimilarity. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system.

interpretation *S-transform*: *indexed-nominal-ts* $(\vdash_S)$ $(\rightarrow_S)$
⟨*proof*⟩

notation *S-transform.bisimilar* (**infix** ‹$\sim\cdot_S$› *100*)

theorem $P \sim\cdot_S Q \longrightarrow P \sim\cdot Q$
⟨*proof*⟩

end

27.7 Translation of weak formulas into formulas without predicates

context *indexed-weak-nominal-ts*
begin

notation *S-satisfies* (**infix** ‹$\vdash_S$› *70*)

interpretation *S-transform*: *indexed-weak-nominal-ts S-action.Act* τ $(\vdash_S)$ $(\rightarrow_S)$
⟨*proof*⟩

notation *S-transform.valid* (**infix** ‹$\models_S$› *70*)
notation *S-transform.weakly-bisimilar* (**infix** ‹$\approx\cdot_S$› *100*)

The S-transform of a weak formula is not necessarily a weak formula. However, the image of all weak formulas under the S-transform is adequate for weak bisimilarity.

corollary $P \approx\cdot_S Q \longleftrightarrow (\forall x.$ *weak-formula* $x \longrightarrow P \models_S$ *S-transform* $x \longleftrightarrow Q \models_S$ *S-transform* $x)$
⟨*proof*⟩

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** $\forall P.\ P \models x \longleftrightarrow P \models_S y$
⟨*proof*⟩

end

end

References

[1] J. Parrow, J. Borgström, L. Eriksson, R. Gutkovas, and T. Weber. Modal logics for nominal transition systems. In L. Aceto and D. de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory,*

CONCUR 2015, Madrid, Spain, September 1-4, 2015, volume 42 of *LIPIcs*, pages 198–211. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.