

Modal Logics for Nominal Transition Systems

Tjark Weber et al.

May 21, 2019

Abstract

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

Contents

| | | |
|----------|---|-----------|
| 1 | Bounded Sets Equipped With a Permutation Action | 4 |
| 2 | Lemmas about Well-Foundedness and Permutations | 5 |
| 2.1 | Hull and well-foundedness | 5 |
| 3 | Residuals | 6 |
| 3.1 | Binding names | 6 |
| 3.2 | Raw residuals and α -equivalence | 6 |
| 3.3 | Residuals | 7 |
| 3.4 | Notation for pairs as residuals | 8 |
| 3.5 | Support of residuals | 8 |
| 3.6 | Equality between residuals | 8 |
| 3.7 | Strong induction | 8 |
| 3.8 | Other lemmas | 9 |
| 4 | Nominal Transition Systems and Bisimulations | 9 |
| 4.1 | Basic Lemmas | 9 |
| 4.2 | Nominal transition systems | 9 |
| 4.3 | Bisimulations | 9 |
| 5 | Infinitary Formulas | 11 |
| 5.1 | Infinitely branching trees | 11 |
| 5.2 | Trees modulo α -equivalence | 12 |
| 5.3 | Constructors for trees modulo α -equivalence | 16 |
| 5.4 | Induction over trees modulo α -equivalence | 18 |
| 5.5 | Hereditarily finitely supported trees | 19 |

| | | |
|-----------|--|-----------|
| 5.6 | Infinitary formulas | 19 |
| 5.7 | Constructors for infinitary formulas | 21 |
| 5.8 | Induction over infinitary formulas | 23 |
| 5.9 | Strong induction over infinitary formulas | 23 |
| 6 | Validity | 24 |
| 6.1 | Validity for infinitely branching trees | 24 |
| 6.2 | Validity for trees modulo α -equivalence | 25 |
| 6.3 | Validity for infinitary formulas | 25 |
| 7 | (Strong) Logical Equivalence | 26 |
| 8 | Bisimilarity Implies Logical Equivalence | 27 |
| 9 | Logical Equivalence Implies Bisimilarity | 27 |
| 10 | Disjunction | 28 |
| 11 | Expressive Completeness | 29 |
| 11.1 | Distinguishing formulas | 29 |
| 11.2 | Characteristic formulas | 30 |
| 11.3 | Expressive completeness | 31 |
| 12 | Finitely Supported Sets | 31 |
| 13 | Nominal Transition Systems with Effects and F/L-Bisimilarity | 32 |
| 13.1 | Nominal transition systems with effects | 32 |
| 13.2 | L -bisimulations and F/L -bisimilarity | 33 |
| 14 | Infinitary Formulas With Effects | 34 |
| 14.1 | Infinitely branching trees | 34 |
| 14.2 | Trees modulo α -equivalence | 36 |
| 14.3 | Constructors for trees modulo α -equivalence | 40 |
| 14.4 | Induction over trees modulo α -equivalence | 42 |
| 14.5 | Hereditarily finitely supported trees | 42 |
| 14.6 | Infinitary formulas | 43 |
| 14.7 | Constructors for infinitary formulas | 44 |
| 14.8 | F/L -formulas | 47 |
| 14.9 | Induction over infinitary formulas | 47 |
| 14.10 | Strong induction over infinitary formulas | 47 |
| 15 | Validity With Effects | 47 |
| 15.1 | Validity for infinitely branching trees | 48 |
| 15.2 | Validity for trees modulo α -equivalence | 49 |
| 15.3 | Validity for infinitary formulas | 49 |

| | |
|---|-----------|
| 16 (Strong) Logical Equivalence | 50 |
| 17 F/L-Bisimilarity Implies Logical Equivalence | 51 |
| 18 Logical Equivalence Implies F/L-Bisimilarity | 51 |
| 19 L-Transform | 52 |
| 19.1 States | 52 |
| 19.2 Actions and binding names | 53 |
| 19.3 Satisfaction | 54 |
| 19.4 Transitions | 54 |
| 19.5 Translation of F/L -formulas into formulas without effects . . | 55 |
| 19.6 Bisimilarity in the L -transform | 58 |
| 20 Nominal Transition Systems and Bisimulations with Unob- | |
| servable Transitions | 60 |
| 20.1 Nominal transition systems with unobservable transitions . . | 60 |
| 20.2 Weak bisimulations | 61 |
| 21 Weak Formulas | 63 |
| 21.1 Lemmas about α -equivalence involving τ | 63 |
| 21.2 Weak action modality | 63 |
| 21.3 Weak formulas | 65 |
| 22 Weak Validity | 65 |
| 23 Weak Logical Equivalence | 66 |
| 24 Weak Bisimilarity Implies Weak Logical Equivalence | 67 |
| 25 Weak Logical Equivalence Implies Weak Bisimilarity | 68 |
| 26 Weak Expressive Completeness | 69 |
| 26.1 Distinguishing weak formulas | 69 |
| 26.2 Characteristic weak formulas | 70 |
| 26.3 Weak expressive completeness | 71 |
| 27 S-Transform: State Predicates as Actions | 71 |
| 27.1 Actions and binding names | 71 |
| 27.2 Satisfaction | 72 |
| 27.3 Transitions | 72 |
| 27.4 Strong Bisimilarity in the S -transform | 73 |
| 27.5 Weak Bisimilarity in the S -transform | 73 |
| 27.6 Translation of (strong) formulas into formulas without pred- | |
| icates | 74 |
| 27.7 Translation of weak formulas into formulas without predicates | 78 |

```

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  $|A| < o \text{ natLeq } +c \mid UNIV :: 'k \text{ set}$ 
  shows set-bset (Abs-bset  $A :: 'a \text{ set}['k]$ ) =  $A$ 
  <proof>

```

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

```

instantiation bset :: (pt,type) pt
begin

```

```

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

```

```

  instance
  <proof>

```

```

end

```

```

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

```

```

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

```

```

lemma map-bset-permute:  $p \cdot B = \text{map-bset } (\text{permute } p) B$ 
  <proof>

```

```

lemma set-bset-eqv [eqvt]:
   $p \cdot \text{set-bset } B = \text{set-bset } (p \cdot B)$ 
  <proof>

```

lemma *map-bset-eqv* [*eqvt*]:
 $p \cdot \text{map-bset } f \ B = \text{map-bset } (p \cdot f) \ (p \cdot B)$
 $\langle \text{proof} \rangle$

lemma *bempty-eqv* [*eqvt*]: $p \cdot \text{bempty} = \text{bempty}$
 $\langle \text{proof} \rangle$

lemma *binsert-eqv* [*eqvt*]: $p \cdot (\text{binsert } x \ B) = \text{binsert } (p \cdot x) \ (p \cdot B)$
 $\langle \text{proof} \rangle$

lemma *bsingleton-eqv* [*eqvt*]: $p \cdot \text{bsingleton } x = \text{bsingleton } (p \cdot x)$
 $\langle \text{proof} \rangle$

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

2 Lemmas about Well-Foundedness and Permutations

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

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

lemma *wf-less-bool-rel*: *wf less-bool-rel*
 $\langle \text{proof} \rangle$

2.1 Hull and well-foundedness

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

lemma *hull-relp-reflp*: *reflp hull-relp*
 $\langle \text{proof} \rangle$

lemma *hull-relp-symp*: *symp hull-relp*
 $\langle \text{proof} \rangle$

lemma *hull-relp-transp*: *transp hull-relp*
 $\langle \text{proof} \rangle$

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

<proof>

lemma *hull-rel-relcomp-subset*:
 assumes *eqvt R*
 shows $R \circ \text{hull-rel} \subseteq \text{hull-rel} \circ R$
<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 $(p \cdot x, y) \in \text{hull-rel} \circ R$
<proof>

lemma *hull-rel-relcomp-trivialI [simp]*:
 assumes $(x, y) \in R$
 shows $(x, y) \in \text{hull-rel} \circ R$
<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: $p \cdot (\text{bn } \alpha) = \text{bn } (p \cdot \alpha)$*
 and *bn-finite: finite (bn α)*

lemma *bn-subset-supp: bn $\alpha \subseteq \text{supp } \alpha$*
<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 (α1,P1) (α2,P2) ↔ [bn α1]set. (α1, P1) = [bn α2]set. (α2,
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* ($p \cdot \text{rep-residual } res$) (*rep-residual* ($p \cdot res$))
 $\langle \text{proof} \rangle$

3.4 Notation for pairs as residuals

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

where

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

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

3.5 Support of residuals

We only consider finitely supported states now.

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

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

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

3.6 Equality between residuals

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

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

3.7 Strong induction

lemma *residual-strong-induct*:

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

shows P *c residual*
 ⟨proof⟩

3.8 Other lemmas

lemma *residual-empty-bn-eq-iff*:
 assumes $bn\ \alpha1 = \{\}$
 shows $\langle \alpha1, P1 \rangle = \langle \alpha2, P2 \rangle \longleftrightarrow \alpha1 = \alpha2 \wedge P1 = P2$
 ⟨proof⟩

lemma *set-bounded-supp*:
 assumes *finite* S and $\bigwedge x. x \in X \implies supp\ x \subseteq S$
 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-eqt* [*eqt*]:
 assumes *symp* R shows *symp* $(p \cdot R)$
 ⟨proof⟩

4.2 Nominal transition systems

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

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

end

4.3 Bisimulations

context *nominal-ts*
begin

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

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

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

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

lemma $is\text{-bisimulation}\text{-eqvt}$:
assumes $is\text{-bisimulation } R$ **shows** $is\text{-bisimulation } (p \cdot R)$
 $\langle proof \rangle$

lemma $bisimilar\text{-eqvt}$:
assumes $P \sim \cdot Q$ **shows** $(p \cdot P) \sim \cdot (p \cdot Q)$
 $\langle proof \rangle$

lemma $bisimilar\text{-reflp}$: $reflp\ bisimilar$
 $\langle proof \rangle$

lemma $bisimilar\text{-symp}$: $symp\ bisimilar$
 $\langle proof \rangle$

lemma $bisimilar\text{-is-bisimulation}$: $is\text{-bisimulation } bisimilar$
 $\langle proof \rangle$

lemma $bisimilar\text{-transp}$: $transp\ bisimilar$
 $\langle proof \rangle$

lemma $bisimilar\text{-equivp}$: $equivp\ bisimilar$
 $\langle proof \rangle$

lemma $bisimilar\text{-simulation}\text{-step}$:
assumes $P \sim \cdot Q$ **and** $bn\ \alpha\ \#* Q$ **and** $P \rightarrow \langle \alpha, P' \rangle$
obtains Q' **where** $Q \rightarrow \langle \alpha, Q' \rangle$ **and** $P' \sim \cdot Q'$
 $\langle proof \rangle$

end

end

theory $Formula$

imports

$Nominal\text{-Bounded}\text{-Set}$

$Nominal\text{-Wellfounded}$

$Residual$

begin

5 Infinitary Formulas

5.1 Infinitely branching trees

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

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

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

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

```
lemma wf-Tree-wf: wf Tree-wf
⟨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 · (tConj tset) = tConj (map-bset (permute p) tset) — neat trick to get
    around the fact that tset is not of permutation type yet
  | p · (tNot t) = tNot (p · t)
  | p · (tPred φ) = tPred (p · φ)
  | p · (tAct α t) = tAct (p · α) (p · t)
```

```
instance
⟨proof⟩
```

```
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 · tConj tset = tConj (p · tset)
⟨proof⟩
```

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

The relation *Tree-wf* is equivariant.

lemma *Tree-wf-eqvt-aux*:

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

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

<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*]: $\text{supp } (t\text{Conj } tset) = \text{supp } tset$

<proof>

lemma *supp-tNot* [*simp*]: $\text{supp } (t\text{Not } t) = \text{supp } t$

<proof>

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

<proof>

lemma *supp-tAct* [*simp*]: $\text{supp } (t\text{Act } \alpha t) = \text{supp } \alpha \cup \text{supp } t$

<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*]:

$p \cdot \text{supp-rel } R x = \text{supp-rel } (p \cdot R) (p \cdot x)$

<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* $(t\text{Conj } tset) = \text{supp-rel } \alpha\text{-Tree } (t\text{Conj } tset)$.

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

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

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

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

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

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

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

$\llbracket bs=bs'; x=x'; R (p' \cdot x') y' \longleftrightarrow R' (p' \cdot x') y'; f x' = f' x'; f y' = f' y'; p=p'; cs=cs'; y=y' \rrbracket \implies \text{alpha-set } (bs, x) R f p (cs, y) \longleftrightarrow \text{alpha-set } (bs', x') R' f' p' (cs', y')$
 $\langle \text{proof} \rangle$

quotient-type

$(\text{'id}x, \text{'pred}, \text{'act}) \text{Tree}_p = (\text{'id}x, \text{'pred}::\text{pt}, \text{'act}::\text{bn}) \text{Tree} / \text{hull-rel}_p$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *permute-rep-abs-Tree_p*:

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

$\langle \text{proof} \rangle$

lift-definition *Tree-wf_p* :: $(\text{'id}x, \text{'pred}::\text{pt}, \text{'act}::\text{bn}) \text{Tree}_p \text{ rel is}$

Tree-wf $\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

lemma *Tree-wf_pE*:

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

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

lemma $\text{wf-Tree-wf}_p: \text{wf Tree-wf}_p$
 ⟨proof⟩

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

Here it comes ...

function (*sequential*)

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

— $(=_\alpha)$

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

| $\text{alpha-tNot}: \text{tNot}\ t1 =_\alpha\ \text{tNot}\ t2 \iff t1 =_\alpha\ t2$

| $\text{alpha-tPred}: \text{tPred}\ \varphi1 =_\alpha\ \text{tPred}\ \varphi2 \iff \varphi1 = \varphi2$

— the action may have binding names

| $\text{alpha-tAct}: \text{tAct}\ \alpha1\ t1 =_\alpha\ \text{tAct}\ \alpha2\ t2 \iff$

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

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

— 254 subgoals (!)

⟨proof⟩

termination

⟨proof⟩

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

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

lemma $\text{alpha-Tree-induct}[\text{case-names}\ \text{tConj}\ \text{tNot}\ \text{tPred}\ \text{tAct}, \text{consumes}\ 1]:$

assumes $t1 =_\alpha t2$

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

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

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

and $\bigwedge \varphi. P\ (\text{tPred}\ \varphi)\ (\text{tPred}\ \varphi)$

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

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

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

$P\ (\text{tAct}\ \alpha1\ t1)\ (\text{tAct}\ \alpha2\ t2)$

shows $P\ t1\ t2$
<proof>

α -equivalence is equivariant.

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

lemma *alpha-Tree-eqvt'*: $t1 =_{\alpha} t2 \longleftrightarrow p \cdot t1 =_{\alpha} p \cdot t2$
<proof>

lemma *alpha-Tree-eqvt [eqvt]*: $t1 =_{\alpha} t2 \implies p \cdot t1 =_{\alpha} p \cdot t2$
<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 $\text{supp-rel } (=_{\alpha})\ t1 = \text{supp-rel } (=_{\alpha})\ t2$
<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]*: $\text{supp-rel } (=_{\alpha})\ (tNot\ t) = \text{supp-rel } (=_{\alpha})\ t$
<proof>

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

The support modulo α -equivalence of $tAct\ \alpha\ t$ is not easily described: when t has infinite support (modulo α -equivalence), the support (modulo

α -equivalence) of $tAct \alpha t$ may still contain names in $bn \alpha$. This incongruity is avoided when t has finite support modulo α -equivalence.

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

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

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 $permute\text{-Tree}_{\alpha} :: perm \Rightarrow (\text{'a}, \text{'b}, \text{'c}) Tree_{\alpha} \Rightarrow (\text{'a}, \text{'b}, \text{'c}) Tree_{\alpha}$
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-Tree $_{\alpha}$.abs-eq [eqvt, simp]*

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

5.3 Constructors for trees modulo α -equivalence

The constructors are lifted from trees.

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

tConj
<proof>

lemma *map-bset-abs-rep-Tree_α*: *map-bset abs-Tree_α (map-bset rep-Tree_α tset_α) = tset_α*
<proof>

lemma *Conj_α-def'*: *Conj_α tset_α = abs-Tree_α (tConj (map-bset rep-Tree_α tset_α))*
<proof>

lift-definition *Not_α* :: *('idx, 'pred, 'act) Tree_α ⇒ ('idx, 'pred::pt, 'act::bn) Tree_α is*
tNot
<proof>

lift-definition *Pred_α* :: *'pred ⇒ ('idx, 'pred::pt, 'act::bn) Tree_α is*
tPred
<proof>

lift-definition *Act_α* :: *'act ⇒ ('idx, 'pred, 'act) Tree_α ⇒ ('idx, 'pred::pt, 'act::bn) Tree_α is*
tAct
<proof>

The lifted constructors are equivariant.

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

lemma *Not_α-eqvt* [*eqvt, simp*]: *p · Not_α t_α = Not_α (p · t_α)*
<proof>

lemma *Pred_α-eqvt* [*eqvt, simp*]: *p · Pred_α φ = Pred_α (p · φ)*
<proof>

lemma *Act_α-eqvt* [*eqvt, simp*]: *p · Act_α α t_α = Act_α (p · α) (p · t_α)*
<proof>

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

lemma *Conj_α-eq-iff* [*simp*]: *Conj_α tset1_α = Conj_α tset2_α ⟷ tset1_α = tset2_α*
<proof>

lemma *Not_α-eq-iff* [*simp*]: *Not_α t1_α = Not_α t2_α ⟷ t1_α = t2_α*
<proof>

lemma *Pred_α-eq-iff* [*simp*]: *Pred_α φ1 = Pred_α φ2 ⟷ φ1 = φ2*
<proof>

lemma *Act_α-eq-iff*: *Act_α α1 t1 = Act_α α2 t2 ⟷ tAct α1 (rep-Tree_α t1) =_α tAct α2 (rep-Tree_α 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 $Conj_\alpha\ Not_\alpha\ Pred_\alpha\ Act_\alpha\ Env_\alpha$, induct type: $Tree_\alpha$]:
fixes t_α
assumes $\bigwedge tset_\alpha. (\bigwedge x. x \in set\text{-}bset\ tset_\alpha \implies P\ x) \implies P\ (Conj_\alpha\ tset_\alpha)$
and $\bigwedge t_\alpha. P\ t_\alpha \implies P\ (Not_\alpha\ t_\alpha)$
and $\bigwedge pred. P\ (Pred_\alpha\ pred)$
and $\bigwedge act\ t_\alpha. P\ t_\alpha \implies P\ (Act_\alpha\ act\ t_\alpha)$
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 ($\bigwedge t_{\alpha}. t_{\alpha} \in \text{set-bset } tset_{\alpha} \Longrightarrow \text{hereditarily-fs } t_{\alpha}$)
 \Longrightarrow hereditarily-fs (Conj $_{\alpha}$ tset $_{\alpha}$)
 | Not $_{\alpha}$: hereditarily-fs $t_{\alpha} \Longrightarrow$ hereditarily-fs (Not $_{\alpha}$ t_{α})
 | Pred $_{\alpha}$: hereditarily-fs (Pred $_{\alpha}$ φ)
 | Act $_{\alpha}$: hereditarily-fs $t_{\alpha} \Longrightarrow$ hereditarily-fs (Act $_{\alpha}$ 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 $_{\alpha}$ α $t_{\alpha} =$ Act $_{\alpha}$ 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>

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 $_{\alpha}. \text{hereditarily-fs } t_{\alpha}$ }
<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
 \langle *proof* \rangle

lemma *Rep-formula'* [simp]: *hereditarily-fs* (*Rep-formula* x)
 \langle *proof* \rangle

Now we lift the permutation operation.

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

lift-definition *permute-formula* :: *perm* \Rightarrow (*a*,*b*,*c*) *formula* \Rightarrow (*a*,*b*,*c*) *formula*
is *permute*
 \langle *proof* \rangle

instance
 \langle *proof* \rangle

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* ($p \cdot t_\alpha$)
 \langle *proof* \rangle

lemma *supp-Abs-formula* [simp]:
assumes *hereditarily-fs* t_α
shows *supp* (*Abs-formula* t_α) = *supp* t_α
 \langle *proof* \rangle

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

lemma *supp-Rep-formula* [simp]: *supp* (*Rep-formula* x) = *supp* x
 \langle *proof* \rangle

lemma *supp-map-bset-Rep-formula* [simp]: *supp* (*map-bset* *Rep-formula* $xset$) =
supp $xset$
 \langle *proof* \rangle

Formulas are in fact finitely supported.

instance *formula* :: (*type*, *fs*, *bn*) *fs*
 \langle *proof* \rangle

5.7 Constructors for infinitary formulas

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

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

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

lemma $Conj\ rep\ eq$:
assumes $finite\ (supp\ xset)$
shows $Rep\ formula\ (Conj\ xset) = Conj_\alpha\ (map\ bset\ Rep\ formula\ xset)$
 $\langle proof \rangle$

lift-definition $Not :: ('idx, 'pred, 'act)\ formula \Rightarrow ('idx, 'pred::fs, 'act::bn)\ formula$ **is**
 Not_α
 $\langle proof \rangle$

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

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

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

lemma $Conj\ eqvt$ [simp]:
assumes $finite\ (supp\ xset)$
shows $p \cdot Conj\ xset = Conj\ (p \cdot xset)$
 $\langle proof \rangle$

lemma $Not\ eqvt$ [eqvt, simp]: $p \cdot Not\ x = Not\ (p \cdot x)$
 $\langle proof \rangle$

lemma $Pred\ eqvt$ [eqvt, simp]: $p \cdot Pred\ \varphi = Pred\ (p \cdot \varphi)$
 $\langle proof \rangle$

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

The following lemmas describe the support of constructed formulas.

lemma *supp-Conj* [*simp*]:
assumes *finite* (*supp xset*)
shows $\text{supp } (\text{Conj } xset) = \text{supp } xset$
 ⟨*proof*⟩

lemma *supp-Not* [*simp*]: $\text{supp } (\text{Not } x) = \text{supp } x$
 ⟨*proof*⟩

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

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

lemma *bn-fresh-Act* [*simp*]: $\text{bn } \alpha \#* \text{Act } \alpha x$
 ⟨*proof*⟩

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

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

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

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

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

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: $\text{Act } \alpha1 x1 = \text{Act } \alpha2 x2 \longleftrightarrow$
 $(\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2)$
(is ?l \longleftrightarrow ?r)
 ⟨*proof*⟩

lemma *Act-eq-iff-perm-renaming*: $\text{Act } \alpha1 x1 = \text{Act } \alpha2 x2 \longleftrightarrow$
 $(\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot$

$\alpha 1 = \alpha 2 \wedge \text{supp } p \subseteq \text{bn } \alpha 1 \cup p \cdot \text{bn } \alpha 1$
 (is ?l \longleftrightarrow ?r)
 <proof>

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

lemma *Tree-free [simp]:*
 shows $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Not } x$
 and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Pred } \varphi$
 and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Act } \alpha x$
 and $\text{Not } x \neq \text{Pred } \varphi$
 and $\text{Not } x1 \neq \text{Act } \alpha x2$
 and $\text{Pred } \varphi \neq \text{Act } \alpha x$
 <proof>

5.8 Induction over infinitary formulas

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

5.9 Strong induction over infinitary formulas

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

lemmas *formula-strong-induct = formula-strong-induct-aux[where p=0, simplified]*

declare *formula-strong-induct [case-names Conj Not Pred Act]*

end

theory *Validity*

imports

Transition-System

Formula

begin

6 Validity

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

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

lemma *alpha-Tree-relI* [*simp*]:
assumes $x =_\alpha y$ **shows** $(x,y) \in \text{alpha-Tree-rel}$
 $\langle \text{proof} \rangle$

lemma *alpha-Tree-relE*:
assumes $(x,y) \in \text{alpha-Tree-rel}$ **and** $x =_\alpha y \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
 $wf (\text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf})$
 $\langle \text{proof} \rangle$

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

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

6.1 Validity for infinitely branching trees

context *nominal-ts*
begin

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

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

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

termination $\langle proof \rangle$

valid-Tree is equivariant.

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

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

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

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

lemma *valid-Tree $_{\alpha}$ -Not $_{\alpha}$* [*simp*]: $valid-Tree_{\alpha} P (Not_{\alpha} t_{\alpha}) \longleftrightarrow \neg valid-Tree_{\alpha} P t_{\alpha}$
 $\langle proof \rangle$

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

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

6.3 Validity for infinitary formulas

lift-definition *valid* :: $'state \Rightarrow ('idx, 'pred, 'act) formula \Rightarrow bool$ (**infix** \models 70) is
valid-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 $P \models \text{Conj } xset \longleftrightarrow (\forall x \in \text{set-bset } xset. P \models x)$
<proof>

lemma *valid-Not* [*simp*]: $P \models \text{Not } x \longleftrightarrow \neg P \models x$
<proof>

lemma *valid-Pred* [*simp*]: $P \models \text{Pred } \varphi \longleftrightarrow P \vdash \varphi$
<proof>

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

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

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

lemma *valid-Act-fresh*:
 assumes $\text{bn } \alpha \ \sharp^* P$
 shows $P \models \text{Act } \alpha \ x \longleftrightarrow (\exists P'. P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x)$
<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* :: $'state::fs \Rightarrow 'pred::fs \Rightarrow \text{bool}$ (**infix** \vdash 70)
 and *transition* :: $'state \Rightarrow ('act::bn, 'state) \text{residual} \Rightarrow \text{bool}$ (**infix** \rightarrow 70) +
 assumes *card-idx-perm*: $|UNIV::\text{perm set}| < o \ |UNIV::'idx set|$
 and *card-idx-state*: $|UNIV::'state set| < o \ |UNIV::'idx set|$
begin

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

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 $\bigwedge P\ Q. P \sim\cdot Q \implies P \models x \longleftrightarrow Q \models x$
and $P \sim\cdot Q$
and $P \models \text{Act}\ \alpha\ x$
shows $Q \models \text{Act}\ \alpha\ x$
 ⟨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* $(p \cdot Q)$
 \langle *proof* \rangle

lemma *equivalent-iff-not-distinguished*: $(P =\cdot Q) \longleftrightarrow \neg(\exists x. x \text{ distinguishes } P \text{ from } Q)$
 \langle *proof* \rangle

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

lemma *distinguished-bounded-support*:
assumes *x distinguishes P from Q*
obtains y **where** $\text{supp } y \subseteq \text{supp } P$ **and** *y distinguishes P from Q*
 \langle *proof* \rangle

lemma *equivalence-is-bisimulation*: *is-bisimulation logically-equivalent*
 \langle *proof* \rangle

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

end

end

theory *Disjunction*

imports
Formula
Validity

begin

10 Disjunction

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

lemma *finite-supp-map-bset-Not* [*simp*]:
assumes *finite (supp xset)*
shows *finite (supp (map-bset Not xset))*
 \langle *proof* \rangle

lemma *Disj-eqvt* [*simp*]:
assumes *finite (supp xset)*
shows $p \cdot \text{Disj } xset = \text{Disj } (p \cdot xset)$

<proof>

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

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

shows $Disj\ xset1 = Disj\ xset2 \longleftrightarrow xset1 = xset2$

<proof>

context *nominal-ts*

begin

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

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

shows $P \models Disj\ xset \longleftrightarrow (\exists x \in set\ bset\ xset. P \models x)$

<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* $:: 'state \Rightarrow 'state \Rightarrow ('idx, 'pred, 'act)\ formula$
where

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

— just an auxiliary lemma that will be useful further below

lemma *distinguishing-formula-card-aux*:

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

<proof>

lemma *distinguishing-formula-supp-aux*:

assumes $\neg (P =\cdot Q)$

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

<proof>

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

assumes $\neg (P =\cdot Q)$

shows $p \cdot \text{distinguishing-formula } P \ Q = \text{distinguishing-formula } (p \cdot P) \ (p \cdot Q)$

<proof>

lemma *supp-distinguishing-formula*:

assumes $\neg (P =\cdot Q)$

shows $\text{supp } (\text{distinguishing-formula } P \ Q) \subseteq \text{supp } P$

<proof>

lemma *distinguishing-formula-distinguishes*:

assumes $\neg (P =\cdot Q)$

shows $(\text{distinguishing-formula } P \ Q) \text{ distinguishes } P \text{ from } Q$

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

— just an auxiliary lemma that will be useful further below

lemma *characteristic-formula-card-aux*:

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

<proof>

lemma *characteristic-formula-supp-aux*:

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

<proof>

lemma *characteristic-formula-eqt [simp]*:

$p \cdot \text{characteristic-formula } P = \text{characteristic-formula } (p \cdot P)$

<proof>

lemma *characteristic-formula-eqt-raw [simp]*:

$p \cdot \text{characteristic-formula} = \text{characteristic-formula}$

<proof>

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

$Q \models \text{characteristic-formula } P \longleftrightarrow P =\cdot Q$

<proof>

lemma *characteristic-formula-is-characteristic:*

$Q \models \text{characteristic-formula } P \iff P \sim Q$

<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 $\bigwedge P Q. P \in S \implies P \sim Q \implies Q \in S$

shows $P \models \text{Disj } (\text{Abs-bset } (\text{characteristic-formula } 'S)) \iff P \in S$

<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 $'a \text{ fs-set} = \{x :: 'a :: \text{pt set. finite (supp } x)\}$

<proof>

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

Type $'a \text{ fs-set}$ is a finitely supported permutation type.

instantiation $\text{fs-set} :: (\text{pt}) \text{ pt}$

begin

lift-definition $\text{permute-fs-set} :: \text{perm} \Rightarrow 'a \text{ fs-set} \Rightarrow 'a \text{ fs-set}$ **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 ('(∈_{*fs*}')) **and**
member-fs-set ((-/ ∈_{*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-eqt* [*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 × 'eff first × 'eff ⇒ 'eff first

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

begin

lemma effect-apply-eqt-aux [simp]: p · effect-apply = effect-apply
 ⟨proof⟩

lemma effect-apply-eqt' [eqvt]: p · ⟨f⟩P = ⟨p · f⟩(p · P)
 ⟨proof⟩

lemma L-eqt-aux [simp]: p · L = L
 ⟨proof⟩

lemma L-eqt' [eqvt]: p · L (α, P, f) = L (p · α, p · P, p · f)
 ⟨proof⟩

end

13.2 L-bisimulations and F/L-bisimilarity

context effect-nominal-ts

begin

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

is-L-bisimulation R ≡
 ∀ F. symp (R F) ∧
 (∀ P Q. R F P Q → (∀ f. f ∈_{fs} F → (∀ φ. ⟨f⟩P ⊢ φ → ⟨f⟩Q ⊢ φ))) ∧
 (∀ P Q. R F P Q → (∀ f. f ∈_{fs} F → (∀ α P'. bn α †* (⟨f⟩Q, F, f) →
 ⟨f⟩P → ⟨α,P'⟩ → (∃ Q'. ⟨f⟩Q → ⟨α,Q'⟩ ∧ R (L (α,F,f)) P' Q'))))

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

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

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

lemma is-L-bisimulation-eqt [eqvt]:
assumes is-L-bisimulation R **shows** is-L-bisimulation (p · R)
 ⟨proof⟩

```

lemma FL-bisimilar-eqt:
  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  $\text{bn } \alpha \#* (\langle f \rangle Q, F, f)$  and  $\langle f \rangle P \rightarrow$ 
   $\langle \alpha, P \rangle$ 
  obtains  $Q'$  where  $\langle f \rangle Q \rightarrow \langle \alpha, Q \rangle$  and  $P' \sim_{\cdot[L(\alpha, F, f)]} Q'$ 
  <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 $\langle f \rangle$ is always and only used as a prefix to a predicate or an action formula. So to simplify the representation of formula

trees with effects, the effect operator is merged into the predicate or action it precedes.

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

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

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

```
lemma wf-Tree-wf: wf Tree-wf
⟨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 · (tConj tset) = tConj (map-bset (permute p) tset) — neat trick to get
    around the fact that tset is not of permutation type yet
  | p · (tNot t) = tNot (p · t)
  | p · (tPred f φ) = tPred (p · f) (p · φ)
  | p · (tAct f α t) = tAct (p · f) (p · α) (p · 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 · tConj tset = tConj (p · tset)
⟨proof⟩
```

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

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

```
lemma Tree-wf-eqvt-aux:
  assumes (t1, t2) ∈ Tree-wf shows (p · t1, p · t2) ∈ Tree-wf
```

<proof>

lemma *Tree-wf-eqvt* [*eqvt, simp*]: $p \cdot \text{Tree-wf} = \text{Tree-wf}$
<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*]: $\text{supp } (t\text{Conj } tset) = \text{supp } tset$
<proof>

lemma *supp-tNot* [*simp*]: $\text{supp } (t\text{Not } t) = \text{supp } t$
<proof>

lemma *supp-tPred* [*simp*]: $\text{supp } (t\text{Pred } f \ \varphi) = \text{supp } f \cup \text{supp } \varphi$
<proof>

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

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

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

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

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

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

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

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

$\llbracket bs=bs'; x=x'; R (p' \cdot x') y' \longleftrightarrow R' (p' \cdot x') y'; f x' = f' x'; f y' = f' y'; p=p'; cs=cs'; y=y' \rrbracket \Longrightarrow \text{alpha-set } (bs, x) R f p (cs, y) \longleftrightarrow \text{alpha-set } (bs', x') R' f' p' (cs', y')$
 $\langle \text{proof} \rangle$

quotient-type

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

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

$\langle \text{proof} \rangle$

lemma *permute-rep-abs-Tree_p*:

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

$\langle \text{proof} \rangle$

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

$\text{Tree-wf} \langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

lemma *Tree-wf_pE*:

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

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

$\langle \text{proof} \rangle$

lemma *wf-Tree-wf_p*: $\text{wf } \text{Tree-wf}_p$

$\langle \text{proof} \rangle$

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-Tree_p t1, abs-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** =_α 50) **where**

— (=_α)

alpha-tConj: tConj tset1 =_α tConj tset2 ↔ rel-bset *alpha-Tree* tset1 tset2

| *alpha-tNot*: tNot t1 =_α tNot t2 ↔ t1 =_α t2

| *alpha-tPred*: tPred f1 φ1 =_α tPred f2 φ2 ↔ f1 = f2 ∧ φ1 = φ2

— the action may have binding names

| *alpha-tAct*: tAct f1 α1 t1 =_α tAct f2 α2 t2 ↔

f1 = f2 ∧ (∃ p. (bn α1, t1) ≈_{set} *alpha-Tree* (supp-rel *alpha-Tree*) p (bn α2, t2) ∧ (bn α1, α1) ≈_{set} ((=)) supp p (bn α2, α2))

| *alpha-other*: - =_α - ↔ False

— 254 subgoals (!)

⟨*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 =_α t2

and ∧ tset1 tset2. (∧ a b. a ∈ set-bset tset1 ⇒ b ∈ set-bset tset2 ⇒ a =_α b ⇒ P a b) ⇒

rel-bset (=α) tset1 tset2 ⇒ P (tConj tset1) (tConj tset2)

and ∧ t1 t2. t1 =_α t2 ⇒ P t1 t2 ⇒ P (tNot t1) (tNot t2)

and ∧ f φ. P (tPred f φ) (tPred f φ)

and ∧ f1 α1 t1 f2 α2 t2. (∧ p. p · t1 =_α t2 ⇒ P (p · t1) t2) ⇒ f1 = f2 ⇒ (∃ p. (bn α1, t1) ≈_{set} (=α) (supp-rel (=α)) p (bn α2, t2) ∧ (bn α1, α1) ≈_{set} (=) supp p (bn α2, α2)) ⇒

P (tAct f1 α1 t1) (tAct f2 α2 t2)

shows P t1 t2

⟨*proof*⟩

α-equivalence is equivariant.

lemma *alpha-Tree-eqvt-aux*:

assumes ∧ a b. (a =_α b) · t =_α t ↔ p · (a =_α b) · t =_α p · t

shows p · supp-rel (=α) t = supp-rel (=α) (p · t)

<proof>

lemma *alpha-Tree-eqv'*: $t1 =_\alpha t2 \longleftrightarrow p \cdot t1 =_\alpha p \cdot t2$
<proof>

lemma *alpha-Tree-eqv [eqvt]*: $t1 =_\alpha t2 \implies p \cdot t1 =_\alpha p \cdot t2$
<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 $\text{supp-rel } (=_\alpha) t1 = \text{supp-rel } (=_\alpha) t2$
<proof>

tAct preserves α -equivalence.

lemma *alpha-Tree-tAct*:
 assumes $t1 =_\alpha t2$
 shows $tAct f \alpha t1 =_\alpha tAct f \alpha t2$
<proof>

The following lemmas describe the support modulo α -equivalence.

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

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

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

lemma *infinite-mono*: $\text{infinite } S \implies (\bigwedge x. x \in S \implies x \in T) \implies \text{infinite } T$
<proof>

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

assumes *finite* (*supp-rel* ($=_\alpha$) *t*)
shows *supp-rel* ($=_\alpha$) (*tAct f* α *t*) = *supp f* \cup (*supp* α \cup *supp-rel* ($=_\alpha$) *t* - *bn* α)
 \langle *proof* \rangle

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

quotient-type

(*'idx, 'pred, 'act, 'eff*) *Tree* $_\alpha$ = (*'idx, 'pred::pt, 'act::bn, 'eff::fs*) *Tree* / *alpha-Tree*
 \langle *proof* \rangle

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

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

The permutation operation is lifted from trees.

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

lift-definition *permute-Tree* $_\alpha$:: *perm* \Rightarrow (*'a, 'b, 'c, 'd*) *Tree* $_\alpha$ \Rightarrow (*'a, 'b, 'c, 'd*) *Tree* $_\alpha$
is *permute*
 \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-Tree* $_\alpha$.*abs-eq* [*eqvt, simp*]

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

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*
 \langle *proof* \rangle

lemma *map-bset-abs-rep-Tree* $_\alpha$: *map-bset* *abs-Tree* $_\alpha$ (*map-bset* *rep-Tree* $_\alpha$ *tset* $_\alpha$) = *tset* $_\alpha$
 \langle *proof* \rangle

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

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

lift-definition *Pred_α* :: $'eff \Rightarrow 'pred \Rightarrow ('idx, 'pred::pt, 'act::bn, 'eff::fs) \text{Tree}_{\alpha}$ **is**
 $tPred$
 ⟨proof⟩

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

The lifted constructors are equivariant.

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

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

lemma *Pred_α-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_α-eq-iff* [*simp*]: $Conj_{\alpha} \text{ tset1}_{\alpha} = Conj_{\alpha} \text{ tset2}_{\alpha} \iff \text{tset1}_{\alpha} = \text{tset2}_{\alpha}$
 ⟨proof⟩

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

lemma *Pred_α-eq-iff* [*simp*]: $Pred_{\alpha} f1 \varphi1 = Pred_{\alpha} f2 \varphi2 \iff f1 = f2 \wedge \varphi1 = \varphi2$
 ⟨proof⟩

lemma *Act_α-eq-iff*: $Act_{\alpha} f1 \alpha1 t1 = Act_{\alpha} f2 \alpha2 t2 \iff tAct f1 \alpha1 (\text{rep-Tree}_{\alpha} t1) =_{\alpha} tAct f2 \alpha2 (\text{rep-Tree}_{\alpha} t2)$
 ⟨proof⟩

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

lemma *Tree_α-free* [*simp*]:
shows $Conj_{\alpha} \text{ tset}_{\alpha} \neq Not_{\alpha} t_{\alpha}$
and $Conj_{\alpha} \text{ tset}_{\alpha} \neq Pred_{\alpha} f \varphi$

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*: $supp t_\alpha = supp-rel (=_\alpha) (rep-Tree_\alpha t_\alpha)$
 <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 $\bigwedge tset_\alpha. (\bigwedge x. x \in set-bset tset_\alpha \implies P x) \implies P (Conj_\alpha tset_\alpha)$
and $\bigwedge t_\alpha. P t_\alpha \implies P (Not_\alpha t_\alpha)$
and $\bigwedge f pred. P (Pred_\alpha f pred)$
and $\bigwedge f act t_\alpha. P t_\alpha \implies P (Act_\alpha f act t_\alpha)$
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 α \Rightarrow bool **where**
Conj α : finite (supp tset α) \Longrightarrow ($\bigwedge t_\alpha. t_\alpha \in \text{set-bset } tset_\alpha \Longrightarrow \text{hereditarily-fs } t_\alpha$)
 \Longrightarrow *hereditarily-fs* (Conj α tset α)
| *Not α* : *hereditarily-fs* t α \Longrightarrow *hereditarily-fs* (Not α t α)
| *Pred α* : *hereditarily-fs* (Pred α f φ)
| *Act α* : *hereditarily-fs* t α \Longrightarrow *hereditarily-fs* (Act α f α t α)

hereditarily-fs is equivariant.

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

assumes *hereditarily-fs* t α
shows *hereditarily-fs* (p \cdot t α)

<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 α \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 α ::('idx,'pred,'act,'eff) Tree α .
hereditarily-fs t α }

<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* \Rightarrow (*'a*,*'b*,*'c*,*'d*) *formula* \Rightarrow (*'a*,*'b*,*'c*,*'d*)
formula

is *permute*
 \langle *proof* \rangle

instance
 \langle *proof* \rangle

end

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

lemma *Abs-formula-eqvt* [*simp*]:
assumes *hereditarily-fs* t_α
shows $p \cdot \text{Abs-formula } t_\alpha = \text{Abs-formula } (p \cdot t_\alpha)$
 \langle *proof* \rangle

lemma *supp-Abs-formula* [*simp*]:
assumes *hereditarily-fs* t_α
shows $\text{supp } (\text{Abs-formula } t_\alpha) = \text{supp } t_\alpha$
 \langle *proof* \rangle

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

lemma *supp-Rep-formula* [*simp*]: $\text{supp } (\text{Rep-formula } x) = \text{supp } x$
 \langle *proof* \rangle

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

Formulas are in fact finitely supported.

instance *formula* :: (*type*, *fs*, *bn*, *fs*) *fs*
 \langle *proof* \rangle

14.7 Constructors for infinitary formulas

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

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

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

lemma $finite\text{-}supp\text{-}implies\text{-}hereditarily\text{-}fs\text{-}Conj_{\alpha}$ [simp]:

assumes $finite\ (supp\ xset)$

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

$\langle proof \rangle$

lemma $Conj\text{-}rep\text{-}eq$:

assumes $finite\ (supp\ xset)$

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

$\langle proof \rangle$

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

Not_{α}

$\langle proof \rangle$

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

$Pred_{\alpha}$

$\langle proof \rangle$

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

Act_{α}

$\langle proof \rangle$

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

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

assumes $finite\ (supp\ xset)$

shows $p \cdot Conj\ xset = Conj\ (p \cdot xset)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

The following lemmas describe the support of constructed formulas.

lemma $supp\text{-}Conj$ [simp]:

assumes $finite\ (supp\ xset)$

shows $supp\ (Conj\ xset) = supp\ xset$

$\langle proof \rangle$

lemma *supp-Not* [*simp*]: $\text{supp } (\text{Not } x) = \text{supp } x$
 ⟨*proof*⟩

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

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

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

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

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

lemma *Pred-eq-iff* [*simp*]: $\text{Pred } f1 \ \varphi1 = \text{Pred } f2 \ \varphi2 \iff f1 = f2 \wedge \varphi1 = \varphi2$
 ⟨*proof*⟩

lemma *Act-eq-iff*: $\text{Act } f1 \ \alpha1 \ x1 = \text{Act } f2 \ \alpha2 \ x2 \iff \text{Act}_\alpha \ f1 \ \alpha1 \ (\text{Rep-formula } x1) = \text{Act}_\alpha \ f2 \ \alpha2 \ (\text{Rep-formula } x2)$
 ⟨*proof*⟩

Helpful lemmas for dealing with equalities involving *Act*.

lemma *Act-eq-iff-perm*: $\text{Act } f1 \ \alpha1 \ x1 = \text{Act } f2 \ \alpha2 \ x2 \iff$
 $f1 = f2 \wedge (\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2)$
 (**is** *?l* \iff *?r*)
 ⟨*proof*⟩

lemma *Act-eq-iff-perm-renaming*: $\text{Act } f1 \ \alpha1 \ x1 = \text{Act } f2 \ \alpha2 \ x2 \iff$
 $f1 = f2 \wedge (\exists p. \text{supp } x1 - \text{bn } \alpha1 = \text{supp } x2 - \text{bn } \alpha2 \wedge (\text{supp } x1 - \text{bn } \alpha1) \#* p \wedge p \cdot x1 = x2 \wedge \text{supp } \alpha1 - \text{bn } \alpha1 = \text{supp } \alpha2 - \text{bn } \alpha2 \wedge (\text{supp } \alpha1 - \text{bn } \alpha1) \#* p \wedge p \cdot \alpha1 = \alpha2 \wedge \text{supp } p \subseteq \text{bn } \alpha1 \cup p \cdot \text{bn } \alpha1)$
 (**is** *?l* \iff *?r*)
 ⟨*proof*⟩

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

lemma *Tree-free* [*simp*]:
shows $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Not } x$
and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Pred } f \ \varphi$
and $\text{finite } (\text{supp } xset) \implies \text{Conj } xset \neq \text{Act } f \ \alpha \ x$
and $\text{Not } x \neq \text{Pred } f \ \varphi$
and $\text{Not } x1 \neq \text{Act } f \ \alpha \ x2$
and $\text{Pred } f1 \ \varphi \neq \text{Act } f2 \ \alpha \ x$

<proof>

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* \Rightarrow ('*idx*, '*pred*, '*act*, '*effect*) *formula* \Rightarrow *bool*
where

Conj: *finite* (*supp* *xset*) \Longrightarrow ($\bigwedge x. x \in \text{set-bset } xset \Longrightarrow \text{is-FL-formula } F x$) \Longrightarrow
is-FL-formula *F* (*Conj* *xset*)

| *Not*: *is-FL-formula* *F* *x* \Longrightarrow *is-FL-formula* *F* (*Not* *x*)

| *Pred*: $f \in_{fs} F \Longrightarrow \text{is-FL-formula } F$ (*Pred* *f* φ)

| *Act*: $f \in_{fs} F \Longrightarrow \text{bn } \alpha \#* (F, f) \Longrightarrow \text{is-FL-formula } (L (\alpha, F, f)) x \Longrightarrow \text{is-FL-formula } F$ (*Act* *f* α *x*)

abbreviation *in-A* :: ('*idx*, '*pred*, '*act*, '*effect*) *formula* \Rightarrow '*effect first* \Rightarrow *bool*

($- \in \mathcal{A}[-]$ [*51*, *0*] *50*) **where**

$x \in \mathcal{A}[F] \equiv \text{is-FL-formula } F x$

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

lemma *is-FL-formula-eqv* [*eqvt*]: $x \in \mathcal{A}[F] \Longrightarrow p \cdot x \in \mathcal{A}[p \cdot F]$

<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 $\equiv \{(x, y). x =_{\alpha} y\}$

lemma *alpha-Tree-relI* [*simp*]:

assumes $x =_\alpha y$ **shows** $(x,y) \in \text{alpha-Tree-rel}$
 $\langle \text{proof} \rangle$

lemma *alpha-Tree-relE*:
assumes $(x,y) \in \text{alpha-Tree-rel}$ **and** $x =_\alpha y \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *wf-alpha-Tree-rel-hull-rel-Tree-wf*:
 $wf (\text{alpha-Tree-rel } O \text{ hull-rel } O \text{ Tree-wf})$
 $\langle \text{proof} \rangle$

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

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

15.1 Validity for infinitely branching trees

context *effect-nominal-ts*
begin

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

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

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

FL-valid-Tree is equivariant.

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

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

assumes $FL\text{-valid-Tree } P \ t$ **shows** $FL\text{-valid-Tree } (p \cdot P) \ (p \cdot t)$
 $\langle proof \rangle$

α -equivalent trees validate the same states.

lemma $alpha\text{-Tree-FL-valid-Tree}$:
assumes $t1 =_{\alpha} t2$
shows $FL\text{-valid-Tree } P \ t1 \longleftrightarrow FL\text{-valid-Tree } P \ t2$
 $\langle proof \rangle$

15.2 Validity for trees modulo α -equivalence

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

$FL\text{-valid-Tree}$
 $\langle proof \rangle$

lemma $FL\text{-valid-Tree}_{\alpha}\text{-eqvt}$ [eqvt]:
assumes $FL\text{-valid-Tree}_{\alpha} \ P \ t$ **shows** $FL\text{-valid-Tree}_{\alpha} \ (p \cdot P) \ (p \cdot t)$
 $\langle proof \rangle$

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

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

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

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

15.3 Validity for infinitary formulas

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

lemma $FL\text{-valid-eqvt}$ [eqvt]:
assumes $P \models x$ **shows** $(p \cdot P) \models (p \cdot x)$
 $\langle proof \rangle$

lemma $FL\text{-valid-Conj}$ [simp]:
assumes $finite \ (supp \ xset)$

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

lemma *FL-valid-Not* [*simp*]: $P \models \text{Not } x \longleftrightarrow \neg P \models x$
 $\langle \text{proof} \rangle$

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

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

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

lemma *FL-valid-Act-strong*:

assumes *finite* (*supp* X)

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

lemma *FL-valid-Act-fresh*:

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

shows $P \models \text{Act } f \ \alpha \ x \longleftrightarrow (\exists P'. \langle f \rangle P \rightarrow \langle \alpha, P' \rangle \wedge P' \models x)$
 $\langle \text{proof} \rangle$

end

end

theory *FL-Logical-Equivalence*

imports

FL-Validity

begin

16 (Strong) Logical Equivalence

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

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

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

and *transition* :: $'state \Rightarrow ('act::bn, 'state) \text{residual} \Rightarrow \text{bool}$ (**infix** \rightarrow 70)

and *effect-apply* :: $'effect::fs \Rightarrow 'state \Rightarrow 'state$ ($\langle \cdot \rangle$ - [0,101] 100) +

assumes *card-idx-perm*: $|UNIV::\text{perm } \text{set}| < o \ |UNIV::'idx \ \text{set}|$

and *card-idx-state*: $|UNIV::'state \ \text{set}| < o \ |UNIV::'idx \ \text{set}|$

begin

definition *FL-logically-equivalent* :: $'effect \ \text{first} \Rightarrow 'state \Rightarrow 'state \Rightarrow \text{bool}$ **where**
FL-logically-equivalent $F \ P \ Q \equiv$

$\forall x::('idx, 'pred, 'act, 'effect) \text{ formula. } x \in \mathcal{A}[F] \longrightarrow (P \models x \longleftrightarrow Q \models x)$

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

end

end

theory *FL-Bisimilarity-Implies-Equivalence*

imports

FL-Logical-Equivalence

begin

17 F/L -Bisimilarity Implies Logical Equivalence

context *indexed-effect-nominal-ts*

begin

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

assumes $f \in_{fs} F$

and $bn \ \alpha \ \#_* (F, f)$

and $x \in \mathcal{A}[L(\alpha, F, f)]$

and $\bigwedge P Q. P \sim_{\cdot}[L(\alpha, F, f)] Q \implies P \models x \longleftrightarrow Q \models x$

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* :: $('idx, 'pred, 'act, 'effect) \text{ formula} \Rightarrow 'state$
 $\Rightarrow 'state \Rightarrow \text{bool}$

(- *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-eqt* :

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 $F P Q \longleftrightarrow \neg(\exists x. x \in \mathcal{A}[F] \wedge x \text{ distinguishes } P \text{ from } Q)$
 ⟨proof⟩

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

lemma *FL-distinguished-bounded-support*:

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

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** $P \sim.[F] Q$
 ⟨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* \times *'effect fs-set* \times *'state*
 | EF *'effect fs-set* \times *'state*

instantiation $L\text{-state} :: (pt,pt) pt$

begin

fun *permute-L-state* :: *perm* \Rightarrow ('a,'b) *L-state* \Rightarrow ('a,'b) *L-state* **where**
 $p \cdot (AC\ x) = AC\ (p \cdot x)$
 | $p \cdot (EF\ x) = EF\ (p \cdot x)$

instance
<proof>

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

instance
<proof>

end

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

lemma *supp-Act* [*simp*]: *supp* (Act α) = *supp* α
<proof>

lemma *supp-Eff* [*simp*]: $\text{supp} (\text{Eff } f) = \text{supp } f$
 $\langle \text{proof} \rangle$

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

instance
 $\langle \text{proof} \rangle$

end

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

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

instance
 $\langle \text{proof} \rangle$

end

19.3 Satisfaction

context *effect-nominal-ts*
begin

fun *L-satisfies* :: (*'state,'effect*) *L-state* \Rightarrow *'pred* \Rightarrow *bool* (**infix** \vdash_L $\gamma 0$) **where**
 $AC (-, -, P) \vdash_L \varphi \longleftrightarrow P \vdash \varphi$
 $|\ EF - \vdash_L \varphi \longleftrightarrow \text{False}$

lemma *L-satisfies-eqvt*: **assumes** $P_L \vdash_L \varphi$ **shows** $(p \cdot P_L) \vdash_L (p \cdot \varphi)$
 $\langle \text{proof} \rangle$

end

19.4 Transitions

context *effect-nominal-ts*
begin

fun *L-transition* :: (*'state,'effect*) *L-state* \Rightarrow ((*'act,'effect*) *L-action*, (*'state,'effect*) *L-state*) *residual* \Rightarrow *bool* (**infix** \rightarrow_L $\gamma 0$) **where**
 $AC (f, F, P) \rightarrow_L \alpha P' \longleftrightarrow (\exists \alpha P'. P \rightarrow \langle \alpha, P^\wedge \rangle \wedge \alpha P' = \langle \text{Act } \alpha, EF (L (\alpha, F, f), P^\wedge) \rangle \wedge \text{bn } \alpha \#_* (F, f))$ — note the freshness condition
 $|\ EF (F, P) \rightarrow_L \alpha P' \longleftrightarrow (\exists f. f \in_{fs} F \wedge \alpha P' = \langle \text{Eff } f, AC (f, F, \langle f \rangle P) \rangle)$

lemma *L-transition-eqt*: **assumes** $P_L \rightarrow_L \alpha_L P_L'$ **shows** $(p \cdot P_L) \rightarrow_L (p \cdot \alpha_L P_L')$
 $\langle proof \rangle$

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 $\exists \alpha\ P'.\ P \rightarrow \langle \alpha, P' \rangle \wedge \langle \alpha_L, P_L \rangle = \langle Act\ \alpha, EF\ (L\ (\alpha, F, f), P') \rangle \wedge bn\ \alpha$
 $\sharp^* X$
 $\langle proof \rangle$

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

end

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

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

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

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

lemma *L-transform-Tree-eqt* [*eqt*]: $p \cdot L\text{-transform-Tree}\ t = L\text{-transform-Tree}\ (p \cdot t)$
 $\langle proof \rangle$

L-transform-Tree respects α -equivalence.

lemma *alpha-Tree-L-transform-Tree*:

assumes *alpha-Tree* $t1\ t2$

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

<proof>

L-transform for trees modulo α -equivalence.

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

L-transform-Tree

<proof>

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

($p \cdot t_{\alpha}$)

<proof>

lemma *L-transform-Tree $_{\alpha}$ -Conj $_{\alpha}$ [simp]*: $L\text{-transform-Tree}_{\alpha} (Conj_{\alpha} tset_{\alpha}) = Formula.Conj_{\alpha} (map\text{-bset } L\text{-transform-Tree}_{\alpha} tset_{\alpha})$

<proof>

lemma *L-transform-Tree $_{\alpha}$ -Not $_{\alpha}$ [simp]*: $L\text{-transform-Tree}_{\alpha} (Not_{\alpha} t_{\alpha}) = Formula.Not_{\alpha} (L\text{-transform-Tree}_{\alpha} t_{\alpha})$

<proof>

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

<proof>

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

<proof>

lemma *finite-supp-map-bset-L-transform-Tree $_{\alpha}$ [simp]*:

assumes *finite* (*supp* $tset_{\alpha}$)

shows *finite* (*supp* (*map-bset* $L\text{-transform-Tree}_{\alpha} tset_{\alpha}$))

<proof>

lemma *L-transform-Tree $_{\alpha}$ -preserves-hereditarily-fs*:

assumes *hereditarily-fs* t_{α}

shows *Formula.hereditarily-fs* ($L\text{-transform-Tree}_{\alpha} t_{\alpha}$)

<proof>

L-transform for F/L -formulas.

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

L-transform-Tree $_{\alpha}$

<proof>

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

lemma *L-transform-formula-Conj* [simp]:
assumes *finite* (*supp* *xset*)
shows $L\text{-transform-formula } (Conj\ xset) = Formula.Conj_{\alpha} (map\ bset\ L\text{-transform-formula } xset)$
 ⟨proof⟩

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

lemma *L-transform-formula-Pred* [simp]: $L\text{-transform-formula } (Pred\ f\ \varphi) = Formula.Act_{\alpha} (Eff\ f) (Formula.Pred_{\alpha}\ \varphi)$
 ⟨proof⟩

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

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

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

definition *L-transform* :: $(\prime idx, \prime pred :: fs, \prime act :: bn, \prime eff :: fs)\ formula \Rightarrow (\prime idx, \prime pred, \prime act, \prime eff)\ L\text{-action})\ Formula.formula$ **where**
 $L\text{-transform } x = Formula.Abs\ formula (L\text{-transform-formula } x)$

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

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

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

lemma *L-transform-Not* [simp]: $L\text{-transform } (Not\ x) = Formula.Not (L\text{-transform } x)$
 ⟨proof⟩

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

lemma *L-transform-Act* [simp]: *L-transform* (FL-*Formula.Act* $f \alpha$) = *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 ($x::('idx, 'pred, 'act, 'effect) \text{ formula}$) $\in \mathcal{A}[F]$
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_L 100)

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

inductive *L-bisimilar* :: (*'state, 'effect*) *L-state* \Rightarrow (*'state, 'effect*) *L-state* \Rightarrow *bool*
where

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

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

definition *invL-FL-bisimilar* :: *'effect first* \Rightarrow *'state* \Rightarrow *'state* \Rightarrow *bool* **where**
invL-FL-bisimilar $F P Q \equiv$ EF (F, P) \sim_L EF(F, Q)

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

theorem $P \sim_L[F] Q \longleftrightarrow$ EF (F, P) \sim_L EF(F, Q)

<proof>

end

The following (alternative) proof of the “ \leftarrow ” direction of this equivalence, namely that bisimilarity in the L -transform implies F/L -bisimilarity, uses the fact that the L -transform preserves satisfaction of formulas, together with the fact that bisimilarity (in the L -transform) implies logical equivalence. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system with effects where, additionally, the cardinality of the state set of the L -transform is bounded. We could re-organize our formalization to remove this assumption: the proof of $\llbracket \text{indexed-nominal-ts TYPE}(?'idx) ?satisfies ?transition; \text{nominal-ts.bisimilar } ?satisfies ?transition ?P ?Q \rrbracket \implies \text{indexed-nominal-ts.logically-equivalent TYPE}(?'idx) ?satisfies ?transition ?P ?Q$ 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 $L :: ('act::bn) \times ('effect::fs) fs\text{-set} \times 'effect \Rightarrow 'effect fs\text{-set}$

and $satisfies :: 'state::fs \Rightarrow 'pred::fs \Rightarrow \text{bool}$ (**infix** \vdash 70)

and $transition :: 'state \Rightarrow ('act, 'state) residual \Rightarrow \text{bool}$ (**infix** \rightarrow 70)

and $effect\text{-apply} :: 'effect \Rightarrow 'state \Rightarrow 'state$ (($-$)- [0,101] 100) +

assumes $card\text{-idx-}L\text{-transform-state}: |UNIV::('state, 'effect) L\text{-state set}| < o |UNIV::'idx set|$

begin

interpretation L -transform: indexed-nominal-ts (\vdash_L) (\rightarrow_L)

<proof>

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

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

<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]: $bn \ \tau \ \#* \ P$
 ⟨proof⟩

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

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

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

The transition relations defined above are equivariant.

lemma *tau-transition-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 (\Rightarrow), *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 $P \Rightarrow_{\{\alpha\}} P'$
 $\langle proof \rangle$

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

lemma *observable-tau-transition*:
assumes $P \Rightarrow_{\{\tau\}} P'$
shows $P \Rightarrow P'$
 $\langle proof \rangle$

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

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

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

lemma *weak-transition-weakI*:
assumes $P \Rightarrow Q$ **and** $Q \Rightarrow_{\langle \alpha \rangle} Q'$ **and** $Q' \Rightarrow P'$
shows $P \Rightarrow_{\langle \alpha \rangle} P'$
 $\langle proof \rangle$

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

$Q' \wedge R P' Q'))$

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

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

lemma *is-weak-bisimulation-eqvt* :

assumes *is-weak-bisimulation* R **shows** *is-weak-bisimulation* $(p \cdot R)$
<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** $\text{bn } \alpha \#* Q$ **and** $P \Rightarrow \langle \alpha \rangle P'$
obtains Q' **where** $Q \Rightarrow \langle \alpha \rangle Q'$ **and** $P' \approx \cdot Q'$
<proof>

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* \longleftrightarrow *?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* \Rightarrow *'pred::fs* \Rightarrow *bool* (**infix** \vdash *70*)
and *transition* :: *'state* \Rightarrow (*'act::bn, 'state*) *residual* \Rightarrow *bool* (**infix** \rightarrow *70*) +
assumes *card-idx-perm*: $|UNIV::perm\ set| < o\ |UNIV::'idx\ set|$
and *card-idx-state*: $|UNIV::'state\ set| < o\ |UNIV::'idx\ set|$
and *card-idx-nat*: $|UNIV::nat\ set| < o\ |UNIV::'idx\ set|$
begin

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

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

primrec *tau-steps* :: (*'idx, 'pred::fs, 'act::bn*) *formula* \Rightarrow *nat* \Rightarrow (*'idx, 'pred, 'act*)
formula

where
 $tau-steps\ x\ 0 = x$
 $| tau-steps\ x\ (Suc\ n) = Act\ \tau\ (tau-steps\ x\ n)$

lemma *tau-steps-eqvt* [*simp*]:
 $p \cdot tau-steps\ x\ n = tau-steps\ (p \cdot x)\ (p \cdot n)$
<proof>

lemma *tau-steps-eqvt'* [*simp*]:
 $p \cdot tau-steps\ x = tau-steps\ (p \cdot x)$
<proof>

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

$p \cdot \text{tau-steps} = \text{tau-steps}$
 ⟨proof⟩

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

lemma *tau-steps-not-self*:
 $x = \text{tau-steps } x \ n \longleftrightarrow n = 0$
 ⟨proof⟩

definition *weak-tau-modality* :: ('idx,'pred::fs,'act::bn) formula \Rightarrow ('idx,'pred,'act) formula

where
 $\text{weak-tau-modality } x \equiv \text{Disj} (\text{map-bset} (\text{tau-steps } x) (\text{Abs-bset } \text{UNIV}))$

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

lemma *weak-tau-modality-eqvt* [simp]:
 $p \cdot \text{weak-tau-modality } x = \text{weak-tau-modality} (p \cdot x)$
 ⟨proof⟩

lemma *weak-tau-modality-eq-iff* [simp]:
 $\text{weak-tau-modality } x = \text{weak-tau-modality } y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *supp-weak-tau-modality* [simp]:
 $\text{supp} (\text{weak-tau-modality } x) = \text{supp } x$
 ⟨proof⟩

lemma *Act-weak-tau-modality-eq-iff* [simp]:
 $\text{Act } \alpha 1 (\text{weak-tau-modality } x1) = \text{Act } \alpha 2 (\text{weak-tau-modality } x2) \longleftrightarrow \text{Act } \alpha 1$
 $x1 = \text{Act } \alpha 2 x2$
 ⟨proof⟩

definition *weak-action-modality* :: 'act \Rightarrow ('idx,'pred::fs,'act::bn) formula \Rightarrow ('idx,'pred,'act) formula ($\langle\langle-\rangle\rangle$ -)

where
 $\langle\langle\alpha\rangle\rangle x \equiv \text{if } \alpha = \tau \text{ then } \text{weak-tau-modality } x \text{ else } \text{weak-tau-modality} (\text{Act } \alpha (\text{weak-tau-modality } x))$

lemma *weak-action-modality-eqvt* [simp]:
 $p \cdot \langle\langle\alpha\rangle\rangle x = \langle\langle p \cdot \alpha \rangle\rangle (p \cdot x)$
 ⟨proof⟩

lemma *weak-action-modality-tau*:
 $\langle\langle\tau\rangle\rangle x = \text{weak-tau-modality } x$

<proof>

lemma *weak-action-modality-not-tau*:

assumes $\alpha \neq \tau$

shows $\langle\langle\alpha\rangle\rangle x = \text{weak-tau-modality } (\text{Act } \alpha \ (\text{weak-tau-modality } x))$

<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 $\langle\langle\tau\rangle\rangle \text{Act } \alpha \ (\text{weak-tau-modality } x) = \langle\langle\alpha\rangle\rangle x$ by definition, but clearly not $\text{Act } \tau \ (\text{Act } \alpha \ (\text{weak-tau-modality } x)) = \text{Act } \alpha \ x$.

lemma *weak-action-modality-eq*:

assumes $\text{Act } \alpha 1 \ x1 = \text{Act } \alpha 2 \ x2$

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 \Rightarrow bool

where

wf-Conj: $\text{finite } (\text{supp } xset) \Longrightarrow (\bigwedge x. x \in \text{set-bset } xset \Longrightarrow \text{weak-formula } x) \Longrightarrow \text{weak-formula } (\text{Conj } xset)$

| *wf-Not*: $\text{weak-formula } x \Longrightarrow \text{weak-formula } (\text{Not } x)$

| *wf-Act*: $\text{weak-formula } x \Longrightarrow \text{weak-formula } (\langle\langle\alpha\rangle\rangle x)$

| *wf-Pred*: $\text{weak-formula } x \Longrightarrow \text{weak-formula } (\langle\langle\tau\rangle\rangle (\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x))))$

lemma *finite-supp-wf-Pred* [simp]: $\text{finite } (\text{supp } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x)))$

<proof>

weak-formula is equivariant.

lemma *weak-formula-eqvt* [simp]: $\text{weak-formula } x \Longrightarrow \text{weak-formula } (p \cdot x)$

<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*:

$P \models \text{weak-tau-modality } x \longleftrightarrow (\exists n. P \models \text{tau-steps } x \ n)$
 $\langle \text{proof} \rangle$

lemma *tau-steps-iff-tau-transition*:

$(\exists n. P \models \text{tau-steps } x \ n) \longleftrightarrow (\exists P'. P \Rightarrow P' \wedge P' \models x)$
 $\langle \text{proof} \rangle$

lemma *valid-weak-tau-modality*:

$P \models \text{weak-tau-modality } x \longleftrightarrow (\exists P'. P \Rightarrow P' \wedge P' \models x)$
 $\langle \text{proof} \rangle$

lemma *valid-weak-action-modality*:

$P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists \alpha' x' P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \Rightarrow \langle \alpha' \rangle P' \wedge P' \models x')$
 (is ?l \longleftrightarrow ?r)
 $\langle \text{proof} \rangle$

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

lemma *valid-weak-action-modality-strong*:

assumes *finite* (*supp* X)
shows $P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists \alpha' x' P'. \text{Act } \alpha \ x = \text{Act } \alpha' \ x' \wedge P \Rightarrow \langle \alpha' \rangle P' \wedge P' \models x' \wedge \text{bn } \alpha' \ \sharp^* X)$
 $\langle \text{proof} \rangle$

lemma *valid-weak-action-modality-fresh*:

assumes $\text{bn } \alpha \ \sharp^* P$
shows $P \models (\langle \langle \alpha \rangle \rangle x) \longleftrightarrow (\exists P'. P \Rightarrow \langle \alpha \rangle P' \wedge P' \models x)$
 $\langle \text{proof} \rangle$

end

end

theory *Weak-Logical-Equivalence*

imports

Weak-Formula

Weak-Validity

begin

23 Weak Logical Equivalence

context *indexed-weak-nominal-ts*

begin

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

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

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

lemma *weakly-logically-equivalent-eqv*:

assumes $P \equiv Q$ **shows** $p \cdot P \equiv p \cdot Q$
 <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 $\bigwedge P\ Q. P \approx Q \implies P \models x \longleftrightarrow Q \models x$

and $P \approx Q$

— not needed: and *weak-formula* x

and $P \models \langle\langle\alpha\rangle\rangle x$

shows $Q \models \langle\langle\alpha\rangle\rangle x$

<proof>

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

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

and $P \approx Q$

— not needed: and *weak-formula* x

and $P \models \langle\langle\tau\rangle\rangle(\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x)))$

shows $Q \models \langle\langle\tau\rangle\rangle(\text{Conj } (\text{binsert } (\text{Pred } \varphi) (\text{bsingleton } x)))$

<proof>

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

<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-eqt [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: ( $P \equiv Q$ )  $\longleftrightarrow \neg(\exists x. \text{weak-formula }
x \wedge x \text{ distinguishes } P \text{ from } Q)$ 
  <proof>

```

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

```

lemma distinguished-bounded-support:
  assumes weak-formula x and x distinguishes P from Q
  obtains y where weak-formula y and  $\text{supp } y \subseteq \text{supp } P$  and y distinguishes P
from Q
  <proof>

```

```

lemma weak-equivalence-is-weak-bisimulation: is-weak-bisimulation weakly-logically-equivalent
  <proof>

```

```

theorem weak-equivalence-implies-weak-bisimilarity: assumes  $P \equiv Q$  shows  $P \approx Q$ 
  <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 \Rightarrow 'state \Rightarrow ('idx, 'pred, 'act) formula **where**

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

— just an auxiliary lemma that will be useful further below

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

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

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

assumes $\neg (P \equiv \cdot Q)$
shows $\text{supp } (\text{Abs-bset } \{-p \cdot (\varepsilon x. \text{weak-formula } x \wedge \text{supp } x \subseteq \text{supp } (p \cdot P) \wedge x \text{ distinguishes } (p \cdot P) \text{ from } (p \cdot Q))\} | p. \text{True}) :: - \text{set}'\text{idx} \subseteq \text{supp } P$
 $\langle \text{proof} \rangle$

lemma *distinguishing-weak-formula-eqt [simp]*:

assumes $\neg (P \equiv \cdot Q)$
shows $p \cdot \text{distinguishing-weak-formula } P Q = \text{distinguishing-weak-formula } (p \cdot P) (p \cdot Q)$
 $\langle \text{proof} \rangle$

lemma *supp-distinguishing-weak-formula*:

assumes $\neg (P \equiv \cdot Q)$
shows $\text{supp } (\text{distinguishing-weak-formula } P Q) \subseteq \text{supp } P$
 $\langle \text{proof} \rangle$

lemma *distinguishing-weak-formula-distinguishes*:

assumes $\neg (P \equiv \cdot Q)$

shows (*distinguishing-weak-formula* $P Q$) *distinguishes* P from Q
 ⟨*proof*⟩

lemma *distinguishing-weak-formula-is-weak*:
assumes $\neg (P \equiv Q)$
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 $P \equiv \text{Conj } (\text{Abs-bset } \{ \textit{distinguishing-weak-formula } P Q | Q. \neg (P \equiv Q) \})$

— just an auxiliary lemma that will be useful further below

lemma *characteristic-weak-formula-card-aux*:
 $|\{ \textit{distinguishing-weak-formula } P Q | Q. \neg (P \equiv Q) \}| < o \text{ natLeq } + c \mid \text{UNIV} ::$
'idx set
 ⟨*proof*⟩

lemma *characteristic-weak-formula-supp-aux*:
shows $\text{supp } (\text{Abs-bset } \{ \textit{distinguishing-weak-formula } P Q | Q. \neg (P \equiv Q) \}) :: \text{set}[\textit{'idx}] \subseteq \text{supp } P$
 ⟨*proof*⟩

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

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

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

lemma *characteristic-weak-formula-is-characteristic'*:
 $Q \models \textit{characteristic-weak-formula } P \iff P \equiv Q$
 ⟨*proof*⟩

lemma *characteristic-weak-formula-is-characteristic*:
 $Q \models \textit{characteristic-weak-formula } P \iff P \approx Q$
 ⟨*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 $\bigwedge P Q. P \in S \implies P \approx \cdot Q \implies Q \in S$
shows $P \models \text{Disj} (\text{Abs-bset} (\text{characteristic-weak-formula } 'S)) \longleftrightarrow P \in S$
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* \Rightarrow (*'a','b*) *S-action* \Rightarrow (*'a','b*) *S-action* **where**
 $p \cdot (\text{Act } \alpha) = \text{Act } (p \cdot \alpha)$
 | $p \cdot (\text{Pred } \varphi) = \text{Pred } (p \cdot \varphi)$

instance

<proof>

end

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

lemma *supp-Act* [*simp*]: $\text{supp} (\text{Act } \alpha) = \text{supp } \alpha$
<proof>

lemma *supp-Pred [simp]*: $\text{supp } (\text{Pred } \varphi) = \text{supp } \varphi$
 $\langle \text{proof} \rangle$

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

instance
 $\langle \text{proof} \rangle$

end

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

fun *bn-S-action* :: $(a,b) S\text{-action} \Rightarrow \text{atom set}$ **where**
 $bn\text{-}S\text{-action } (\text{Act } \alpha) = bn \ \alpha$
 $| bn\text{-}S\text{-action } (\text{Pred } -) = \{\}$

instance
 $\langle \text{proof} \rangle$

end

27.2 Satisfaction

context *nominal-ts*
begin

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

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

lemma *S-satisfies-eqvt*: **assumes** $P \vdash_S \varphi$ **shows** $(p \cdot P) \vdash_S (p \cdot \varphi)$
 $\langle \text{proof} \rangle$

end

27.3 Transitions

context *nominal-ts*
begin

inductive *S-transition* :: $'state \Rightarrow ((a,b) S\text{-action}, 'state) \text{residual} \Rightarrow bool$
(**infix** \rightarrow_S 70) **where**
 $\text{Act}: P \rightarrow \langle \alpha, P' \rangle \implies P \rightarrow_S \langle \text{Act } \alpha, P' \rangle$

| $Pred: P \vdash \varphi \implies P \rightarrow_S \langle Pred \ \varphi, P \rangle$

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

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 $\bigwedge \alpha. \alpha_S = Act \ \alpha \implies P \rightarrow \langle \alpha, P' \rangle \implies R$
and $\bigwedge \varphi. \alpha_S = Pred \ \varphi \implies P' = P \implies P \vdash \varphi \implies R$
shows R
 $\langle proof \rangle$

lemma *S-transition-Act-iff*: $P \rightarrow_S \langle Act \ \alpha, P' \rangle \longleftrightarrow P \rightarrow \langle \alpha, P' \rangle$
 $\langle proof \rangle$

lemma *S-transition-Pred-iff*: $P \rightarrow_S \langle Pred \ \varphi, P' \rangle \longleftrightarrow P' = P \wedge P \vdash \varphi$
 $\langle proof \rangle$

end

27.4 Strong Bisimilarity in the S -transform

context *nominal-ts*

begin

interpretation *S-transform*: *nominal-ts* $(\vdash_S) (\rightarrow_S)$
 $\langle proof \rangle$

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

notation *S-transform.bisimilar* (**infix** \sim_S 100)

Bisimilarity is equivalent to bisimilarity in the S -transform.

lemma *bisimilar-is-S-transform-bisimulation*: *S-transform.is-bisimulation* *bisimilar*
 $\langle proof \rangle$

lemma *S-transform-bisimilar-is-bisimulation*: *is-bisimulation* *S-transform.bisimilar*
 $\langle proof \rangle$

theorem *S-transform-bisimilar-iff*: $P \sim_S Q \longleftrightarrow P \sim Q$
 $\langle proof \rangle$

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_S 100)

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

<proof>

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

<proof>

lemma *S-transform-weak-transition-iff*: $P \Rightarrow\langle\text{Act } \alpha\rangle_S P' \longleftrightarrow P \Rightarrow\langle\alpha\rangle P'$

<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*: $P \approx_S Q \longleftrightarrow P \approx \cdot Q$

<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 $S\text{-transform-Tree} :: ('idx, 'pred :: fs, 'act :: bn)\ Tree \Rightarrow ('idx, unit, ('act, 'pred)\ S\text{-action})\ Tree_\alpha$ **where**
 $S\text{-transform-Tree}\ (tConj\ tset) = Conj_\alpha\ (map\ bset\ S\text{-transform-Tree}\ tset)$
 $| S\text{-transform-Tree}\ (tNot\ t) = Not_\alpha\ (S\text{-transform-Tree}\ t)$
 $| S\text{-transform-Tree}\ (tPred\ \varphi) = Act_\alpha\ (S\text{-action.Pred}\ \varphi)\ (Conj_\alpha\ bempty)$
 $| S\text{-transform-Tree}\ (tAct\ \alpha\ t) = Act_\alpha\ (S\text{-action.Act}\ \alpha)\ (S\text{-transform-Tree}\ t)$

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

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

lemma $alpha\text{-Tree-}S\text{-transform-Tree}$:
assumes $t1 =_\alpha t2$
shows $S\text{-transform-Tree}\ t1 = S\text{-transform-Tree}\ t2$
 $\langle proof \rangle$

$S\text{-transform}$ for trees modulo α -equivalence.

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

lemma $S\text{-transform-Tree}_\alpha\text{-eqvt}\ [eqvt]: p \cdot S\text{-transform-Tree}_\alpha\ t_\alpha = S\text{-transform-Tree}_\alpha\ (p \cdot t_\alpha)$
 $\langle proof \rangle$

lemma $S\text{-transform-Tree}_\alpha\text{-Conj}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Conj_\alpha\ tset_\alpha) = Conj_\alpha\ (map\ bset\ S\text{-transform-Tree}_\alpha\ tset_\alpha)$
 $\langle proof \rangle$

lemma $S\text{-transform-Tree}_\alpha\text{-Not}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Not_\alpha\ t_\alpha) = Not_\alpha\ (S\text{-transform-Tree}_\alpha\ t_\alpha)$
 $\langle proof \rangle$

lemma $S\text{-transform-Tree}_\alpha\text{-Pred}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Pred_\alpha\ \varphi) = Act_\alpha\ (S\text{-action.Pred}\ \varphi)\ (Conj_\alpha\ bempty)$
 $\langle proof \rangle$

lemma $S\text{-transform-Tree}_\alpha\text{-Act}_\alpha\ [simp]: S\text{-transform-Tree}_\alpha\ (Act_\alpha\ \alpha\ t_\alpha) = Act_\alpha\ (S\text{-action.Act}\ \alpha)\ (S\text{-transform-Tree}_\alpha\ t_\alpha)$

<proof>

lemma *finite-supp-map-bset-S-transform-Tree $_{\alpha}$* [simp]:
 assumes *finite (supp tset $_{\alpha}$)*
 shows *finite (supp (map-bset S-transform-Tree $_{\alpha}$ tset $_{\alpha}$))*
<proof>

lemma *S-transform-Tree $_{\alpha}$ -preserves-hereditarily-fs*:
 assumes *hereditarily-fs t $_{\alpha}$*
 shows *hereditarily-fs (S-transform-Tree $_{\alpha}$ t $_{\alpha}$)*
<proof>

S-transform for (strong) formulas.

lift-definition *S-transform-formula* :: (*'idx, 'pred::fs, 'act::bn*) *formula* \Rightarrow (*'idx, unit, ('act, 'pred) S-action*) *Tree $_{\alpha}$* **is**
 S-transform-Tree $_{\alpha}$
<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* \Rightarrow (*'idx, unit, ('act, 'pred) S-action*) *formula* **where**
 S-transform *x* = *Abs-formula (S-transform-formula x)*

lemma *S-transform-eqv* [eqvt]: $p \cdot S\text{-transform } x = S\text{-transform } (p \cdot x)$
 ⟨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]: $P \models \text{Conj } \text{bempty}$
 ⟨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** $P \models x \longleftrightarrow P \models_S S\text{-transform } x$
 ⟨proof⟩

end

context *indexed-nominal-ts*
begin

The following (alternative) proof of the “ \rightarrow ” direction of theorem *nominal-ts.bisimilar* (\vdash_S) (\rightarrow_S) $?P ?Q = ?P \sim ?Q$, namely that bisimilarity in the *S-transform*

implies bisimilarity in the original transition system, uses the fact that the S -transform(ation) preserves satisfaction of formulas, together with the fact that bisimilarity (in the S -transform) implies logical equivalence, and equivalence (in the original transition system) implies bisimilarity. However, since we proved the latter in the context of indexed nominal transition systems, this proof requires an indexed nominal transition system.

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

notation S -transform.bisimilar (**infix** \sim_S 100)

theorem $P \sim_S Q \longrightarrow P \sim \cdot Q$
 ⟨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_S 100)

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

corollary $P \approx_S Q \iff (\forall x. \text{weak-formula } x \longrightarrow P \models_S S\text{-transform } x \iff Q \models_S S\text{-transform } x)$
 ⟨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** $\forall P. P \models x \iff P \models_S y$

⟨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.