A General Theory of Syntax with Bindings

Lorenzo Gheri and Andrei Popescu

March 17, 2025

Abstract

We formalize a theory of syntax with bindings that has been developed and refined over the last decade to support several large formalization efforts. Terms are defined for an arbitrary number of constructors of varying numbers of inputs, quotiented to alpha-equivalence and sorted according to a binding signature. The theory includes many properties of the standard operators on terms: substitution, swapping and freshness. It also includes bindings-aware induction and recursion principles and support for semantic interpretation. This work has been presented in the ITP 2017 paper "A Formalized General Theory of Syntax with Bindings".

Contents

1 Quasi-Terms with Swapping and Freshness

theory *QuasiTerms-Swap-Fresh* **imports** *Preliminaries*
begin

This section defines and studies the (totally free) datatype of quasi-terms and the notions of freshness and swapping variables for them. “Quasi" refers to the fact that these items are not (yet) factored to alpha-equivalence. We shall later call “terms" those alpha-equivalence classes.

1.1 The datatype of quasi-terms with bindings

datatype
('index,'bindex,'varSort,'var,'opSym)qTerm =
  qVar 'varSort 'var
  |*qOp 'opSym ('index, (('index,'bindex,'varSort,'var,'opSym)qTerm))input*
      ('bindex, (('index,'bindex,'varSort,'var,'opSym)qAbs)) input
and
('index,'bindex,'varSort,'var,'opSym)qAbs =
  qAbs 'varSort 'var ('index,'bindex,'varSort,'var,'opSym)qTerm

Above:

- “Var" stands for “variable injection"
- “Op" stands for “operation"
- “opSym" stands for “operation symbol"
- “q" stands for “quasi"
- “Abs" stands for “abstraction"

Thus, a quasi-term is either (an injection of) a variable, or an operation symbol applied to a term-input and an abstraction-input (where, for any type T, T-inputs are partial maps from indexes to T. A quasi-abstraction is essentially a pair (variable,quasi-term).

type-synonym *('index,'bindex,'varSort,'var,'opSym)qTermItem* =
('index,'bindex,'varSort,'var,'opSym)qTerm +

$('index,'bindex,'varSort,'var,'opSym)qAbs$

abbreviation *termIn* ::
$('index,'bindex,'varSort,'var,'opSym)qTerm \Rightarrow ('index,'bindex,'varSort,'var,'opSym)qTermItem$
where *termIn X == Inl X*

abbreviation *absIn* ::
$('index,'bindex,'varSort,'var,'opSym)qAbs \Rightarrow ('index,'bindex,'varSort,'var,'opSym)qTermItem$
where *absIn A == Inr A*

1.2 Induction principles

definition *qTermLess* :: $('index,'bindex,'varSort,'var,'opSym)qTermItem\ rel$
where
qTermLess == $\{(termIn\ X,\ termIn(qOp\ delta\ inp\ binp))|\ X\ delta\ inp\ binp\ i.\ inp\ i = Some\ X\} \cup$
$\{(absIn\ A,\ termIn(qOp\ delta\ inp\ binp))|\ A\ delta\ inp\ binp\ i.\ binp\ i = Some\ A\} \cup$
$\{(termIn\ X,\ absIn\ (qAbs\ xs\ x\ X))|\ X\ xs\ x.\ True\}$

This induction will be used only temporarily, until we get a better one, involving swapping:

lemma *qTerm-rawInduct*[*case-names Var Op Abs*]:
fixes *X* :: $('index,'bindex,'varSort,'var,'opSym)qTerm$ **and**
A :: $('index,'bindex,'varSort,'var,'opSym)qAbs$ **and** *phi phiAbs*
assumes
Var: $\bigwedge$ *xs x. phi (qVar xs x)* **and**
Op: $\bigwedge$ *delta inp binp.* $[\![$*liftAll phi inp*; *liftAll phiAbs binp*$]\!] \Longrightarrow$ *phi (qOp delta inp binp)* **and**
Abs: $\bigwedge$ *xs x X. phi X* $\Longrightarrow$ *phiAbs (qAbs xs x X)*
shows *phi X* $\wedge$ *phiAbs A*
by (*induct rule*: *qTerm-qAbs.induct*)
(*fastforce intro*!: *Var Op Abs rangeI simp*: *liftAll-def*)+

lemma *qTermLess-wf*: *wf qTermLess*
unfolding *wf-def* **proof** *safe*
fix *chi item*
assume ∗: $\forall$ *item*. ($\forall$ *item'*. (*item'*, *item*) $\in$ *qTermLess* $\longrightarrow$ *chi item'*) $\longrightarrow$ *chi item*
show *chi item*
proof−
{**fix** *X A*
have *chi (termIn X)* $\wedge$ *chi (absIn A)*
apply(*induct rule*: *qTerm-rawInduct*)
using ∗ **unfolding** *qTermLess-def liftAll-def* **by** *blast*+
}
thus *?thesis* **by**(*cases item*) *auto*
qed
qed

lemma *qTermLessPlus-wf*: *wf* (*qTermLess* ^+)
using *qTermLess-wf wf-trancl* **by** *auto*

The skeleton of a quasi-term item – this is the generalization of the size function from the case of finitary syntax. We use the skeleton later for proving correct various recursive function definitions, notably that of "alpha".

function
qSkel :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* ⇒ (*'index*,*'bindex*)*tree*
and
qSkelAbs :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* ⇒ (*'index*,*'bindex*)*tree*
where
qSkel (*qVar xs x*) = *Branch* (λ*i*. *None*) (λ*i*. *None*)
|
qSkel (*qOp delta inp binp*) = *Branch* (*lift qSkel inp*) (*lift qSkelAbs binp*)
|
qSkelAbs (*qAbs xs x X*) = *Branch* (λ*i*. *Some*(*qSkel X*)) (λ*i*. *None*)
by(*pat-completeness*, *auto*)
termination by(*relation qTermLess*, *simp add*: *qTermLess-wf*, *auto simp add*: *qTermLess-def*)

Next is a template for generating induction principles whenever we come up with relation on terms included in the kernel of the skeleton operator.

lemma *qTerm-templateInduct*[*case-names Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*
and *phi phiAbs* **and** *rel*
assumes
REL: ⋀ *X Y*. (*X*,*Y*) ∈ *rel* ⟹ *qSkel Y* = *qSkel X* **and**
Var: ⋀ *xs x*. *phi* (*qVar xs x*) **and**
Op: ⋀ *delta inp binp*. ⟦*liftAll phi inp*; *liftAll phiAbs binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: ⋀ *xs x X*. (⋀ *Y*. (*X*,*Y*) ∈ *rel* ⟹ *phi Y*) ⟹ *phiAbs* (*qAbs xs x X*)
shows *phi X* ∧ *phiAbs A*
proof−
{**fix** *T*
have ∀ *X A*. (*T* = *qSkel X* ⟶ *phi X*) ∧ (*T* = *qSkelAbs A* ⟶ *phiAbs A*)
proof(*induct rule*: *treeLess-induct*)
case (*1 T'*)
show *?case* **apply** *safe*
subgoal for *X* -
using *assms 1* **unfolding** *treeLess-def liftAll-def*
by (*cases X*) (*auto simp add*: *lift-def*, *metis option.simps*(*5*))
subgoal for - *A* **apply** (*cases A*)
using *assms 1* **unfolding** *treeLess-def* **by** *simp* .
qed
}
thus *?thesis* **by** *blast*
qed

A modification of the canonical immediate-subterm relation on quasi-terms, that takes into account a relation assumed included in the skeleton kernel.

definition *qTermLess-modulo* ::
$('index,'bindex,'varSort,'var,'opSym)qTerm\ rel \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qTermItem\ rel$
where
qTermLess-modulo rel ==
$\{(termIn\ X,\ termIn(qOp\ delta\ inp\ binp)) |\ X\ delta\ inp\ binp\ i.\ inp\ i = Some\ X\} \cup$
$\{(absIn\ A,\ termIn(qOp\ delta\ inp\ binp)) |\ A\ delta\ inp\ binp\ j.\ binp\ j = Some\ A\} \cup$
$\{(termIn\ Y,\ absIn\ (qAbs\ xs\ x\ X)) |\ X\ Y\ xs\ x.\ (X,Y) \in rel\}$

lemma *qTermLess-modulo-wf*:
fixes *rel*::$('index,'bindex,'varSort,'var,'opSym)qTerm\ rel$
assumes $\bigwedge X\ Y.\ (X,Y) \in rel \Longrightarrow qSkel\ Y = qSkel\ X$
shows *wf* (*qTermLess-modulo rel*)
proof(*unfold wf-def*, *auto*)
  fix *chi item*
  assume $*$:
  $\forall item.\ (\forall item'.\ (item',\ item) \in qTermLess\text{-}modulo\ rel \longrightarrow chi\ item')$
      $\longrightarrow chi\ item$
  show *chi item*
  proof−
    obtain *phi* **where** *phi-def*: $phi = (\lambda\ X.\ chi\ (termIn\ X))$ **by** *blast*
    obtain *phiAbs* **where** *phiAbs-def*: $phiAbs = (\lambda\ A.\ chi\ (absIn\ A))$ **by** *blast*
    {**fix** *X A*
     have $chi\ (termIn\ X) \wedge chi\ (absIn\ A)$
     apply(*induct rule*: *qTerm-templateInduct*[*of rel*])
     using $*$ *assms* **unfolding** *qTermLess-modulo-def liftAll-def* **by** *blast+*
    }
    thus *?thesis* **unfolding** *phi-def phiAbs-def*
    by(*cases item*, *auto*)
  qed
qed

1.3 Swap and substitution on variables

definition *sw* :: $'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'varSort \Rightarrow 'var \Rightarrow 'var$
where
sw ys y1 y2 xs x ==
 if ys = xs then if x = y1 then y2
        else if x = y2 then y1
            else x
 else x

abbreviation *sw-abbrev* :: $'var \Rightarrow 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'varSort \Rightarrow 'var$
(‹- @-[- ∧ -]′--› *200*)
where (*x* @*xs*[*y1* ∧ *y2*]-*ys*) == *sw ys y1 y2 xs x*

definition *sb* :: $'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'varSort \Rightarrow 'var \Rightarrow 'var$

where
sb ys y1 y2 xs x ==
if ys = *xs then if x* = *y2 then y1*
else x
else x

abbreviation *sb-abbrev* :: *'var* ⇒ *'varSort* ⇒ *'var* ⇒ *'var* ⇒ *'varSort* ⇒ *'var*
(‹- @-[- '/ -]'--› *200*)
where (*x* @*xs*[*y1* / *y2*]-*ys*) == *sb ys y1 y2 xs x*

theorem *sw-simps1*[*simp*]: (*x* @*xs*[*x* ∧ *y*]-*xs*) = *y*
unfolding *sw-def* **by** *simp*

theorem *sw-simps2*[*simp*]: (*x* @*xs*[*y* ∧ *x*]-*xs*) = *y*
unfolding *sw-def* **by** *simp*

theorem *sw-simps3*[*simp*]:
(*zs* ≠ *xs* ∨ *x* ∉ {*z1*,*z2*}) ⟹ (*x* @*xs*[*z1* ∧ *z2*]-*zs*) = *x*
unfolding *sw-def* **by** *simp*

lemmas *sw-simps* = *sw-simps1 sw-simps2 sw-simps3*

theorem *sw-ident*[*simp*]: (*x* @*xs*[*y* ∧ *y*]-*ys*) = *x*
unfolding *sw-def* **by** *auto*

theorem *sw-compose*:
((*z* @*zs*[*x* ∧ *y*]-*xs*) @*zs*[*x'* ∧ *y'*]-*xs'*) =
((*z* @*zs*[*x'* ∧ *y'*]-*xs'*) @*zs*[(*x* @*xs*[*x'* ∧ *y'*]-*xs'*) ∧ (*y* @*xs*[*x'* ∧ *y'*]-*xs'*)]-*xs*)
by(*unfold sw-def*, *auto*)

theorem *sw-commute*:
assumes *zs* ≠ *zs'* ∨ {*x*,*y*} *Int* {*x'*,*y'*} = {}
shows ((*u* @*us*[*x* ∧ *y*]-*zs*) @*us*[*x'* ∧ *y'*]-*zs'*) = ((*u* @*us*[*x'* ∧ *y'*]-*zs'*) @*us*[*x* ∧ *y*]-*zs*)
using *assms* **by**(*unfold sw-def*, *auto*)

theorem *sw-involutive*[*simp*]:
((*z* @*zs*[*x* ∧ *y*]-*xs*) @*zs*[*x* ∧ *y*]-*xs*) = *z*
by(*unfold sw-def*, *auto*)

theorem *sw-inj*[*simp*]:
((*z* @*zs*[*x* ∧ *y*]-*xs*) = (*z'* @*zs*[*x* ∧ *y*]-*xs*)) = (*z* = *z'*)
by (*simp add*: *sw-def*)

lemma *sw-preserves-mship*[*simp*]:
assumes {*y1*,*y2*} ⊆ *Var ys*
shows ((*x* @*xs*[*y1* ∧ *y2*]-*ys*) ∈ *Var xs*) = (*x* ∈ *Var xs*)
using *assms* **unfolding** *sw-def* **by** *auto*

theorem *sw-sym*:

$(z\ @zs[x \wedge y]\text{-}xs) = (z\ @zs[y \wedge x]\text{-}xs)$
by (*unfold sw-def*) *auto*

theorem *sw-involutive2*[*simp*]:
$((z\ @zs[x \wedge y]\text{-}xs)\ @zs[y \wedge x]\text{-}xs) = z$
by (*unfold sw-def*) *auto*

theorem *sw-trans*:
$us \neq zs \vee u \notin \{y,z\} \Longrightarrow$
$((u\ @us[y \wedge x]\text{-}zs)\ @us[z \wedge y]\text{-}zs) = (u\ @us[z \wedge x]\text{-}zs)$
by (*unfold sw-def*) *auto*

lemmas *sw-otherSimps* =
sw-ident sw-involutive sw-inj sw-preserves-mship sw-involutive2

theorem *sb-simps1*[*simp*]: $(x\ @xs[y\ /\ x]\text{-}xs) = y$
unfolding *sb-def* **by** *simp*

theorem *sb-simps2*[*simp*]:
$(zs \neq xs \vee z2 \neq x) \Longrightarrow (x\ @xs[z1\ /\ z2]\text{-}zs) = x$
unfolding *sb-def* **by** *auto*

lemmas *sb-simps* = *sb-simps1 sb-simps2*

theorem *sb-ident*[*simp*]: $(x\ @xs[y\ /\ y]\text{-}ys) = x$
unfolding *sb-def* **by** *auto*

theorem *sb-compose1*:
$((z\ @zs[y1\ /\ x]\text{-}xs)\ @zs[y2\ /\ x]\text{-}xs) = (z\ @zs[(y1\ @xs[y2\ /\ x]\text{-}xs)\ /\ x]\text{-}xs)$
by(*unfold sb-def*, *auto*)

theorem *sb-compose2*:
$ys \neq xs \vee (x2 \notin \{y1,y2\}) \Longrightarrow$
$((z\ @zs[x1\ /\ x2]\text{-}xs)\ @zs[y1\ /\ y2]\text{-}ys) =$
$((z\ @zs[y1\ /\ y2]\text{-}ys)\ @zs[(x1\ @xs[y1\ /\ y2]\text{-}ys)\ /\ x2]\text{-}xs)$
by (*unfold sb-def*) *auto*

theorem *sb-commute*:
assumes $zs \neq zs' \vee \{x,y\}\ Int\ \{x',y'\} = \{\}$
shows $((u\ @us[x\ /\ y]\text{-}zs)\ @us[x'\ /\ y']\text{-}zs') = ((u\ @us[x'\ /\ y']\text{-}zs')\ @us[x\ /\ y]\text{-}zs)$
using *assms* **by** (*unfold sb-def*) *auto*

theorem *sb-idem*[*simp*]:
$((z\ @zs[x\ /\ y]\text{-}xs)\ @zs[x\ /\ y]\text{-}xs) = (z\ @zs[x\ /\ y]\text{-}xs)$
by (*unfold sb-def*) *auto*

lemma *sb-preserves-mship*[*simp*]:
assumes $\{y1,y2\} \subseteq Var\ ys$
shows $((x\ @xs[y1\ /\ y2]\text{-}ys) \in Var\ xs) = (x \in Var\ xs)$

using *assms* **by** (*unfold sb-def*) *auto*

theorem *sb-trans*:
$us \neq zs \vee u \neq y \Longrightarrow$
$((u \ @us[y\ /\ x]\text{-}zs)\ @us[z\ /\ y]\text{-}zs) = (u\ @us[z\ /\ x]\text{-}zs)$
by (*unfold sb-def*) *auto*

lemmas *sb-otherSimps* =
sb-ident sb-idem sb-preserves-mship

1.4 The swapping and freshness operators

For establishing the preliminary results quickly, we use both the notion of binding-sensitive freshness (operator "qFresh") and that of "absolute" freshness, ignoring bindings (operator "qAFresh"). Later, for alpha-equivalence classes, "qAFresh" will not make sense.

definition
aux-qSwap-ignoreFirst3 ::
$'varSort * 'var * 'var * ('index,'bindex,'varSort,'var,'opSym)qTerm +$
$'varSort * 'var * 'var * ('index,'bindex,'varSort,'var,'opSym)qAbs \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qTermItem$
where
aux-qSwap-ignoreFirst3 K =
(*case K of Inl*(*zs*,*x*,*y*,*X*) ⇒ *termIn X*
|*Inr*(*zs*,*x*,*y*,*A*) ⇒ *absIn A*)

lemma *qTermLess-ingoreFirst3-wf*:
wf(*inv-image qTermLess aux-qSwap-ignoreFirst3*)
using *qTermLess-wf wf-inv-image* **by** *auto*

function
$qSwap :: 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow ('index,'bindex,'varSort,'var,'opSym)qTerm$
$\Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qTerm$
and
$qSwapAbs :: 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow ('index,'bindex,'varSort,'var,'opSym)qAbs$
$\Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qAbs$
where
$qSwap\ zs\ x\ y\ (qVar\ zs'\ z) = qVar\ zs'\ (z\ @zs'[x \wedge y]\text{-}zs)$
|
qSwap zs x y (*qOp delta inp binp*) =
qOp delta (*lift* (*qSwap zs x y*) *inp*) (*lift* (*qSwapAbs zs x y*) *binp*)
|
$qSwapAbs\ zs\ x\ y\ (qAbs\ zs'\ z\ X) = qAbs\ zs'\ (z\ @zs'[x \wedge y]\text{-}zs)\ (qSwap\ zs\ x\ y\ X)$
by(*pat-completeness*, *auto*)
termination
by(*relation inv-image qTermLess aux-qSwap-ignoreFirst3*,

simp add: *qTermLess-ingoreFirst3-wf*,
auto simp add: *qTermLess-def aux-qSwap-ignoreFirst3-def*)

lemmas *qSwapAll-simps* = *qSwap.simps qSwapAbs.simps*

abbreviation *qSwap-abbrev* ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* ⇒ '*var* ⇒ '*var* ⇒ '*varSort* ⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* (‹- #[[- ∧ -]]'-› *200*)
where (*X* #[[*z1* ∧ *z2*]]-*zs*) == *qSwap zs z1 z2 X*

abbreviation *qSwapAbs-abbrev* ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* ⇒ '*var* ⇒ '*var* ⇒ '*varSort* ⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* (‹- $[[- ∧ -]]'-› *200*)
where (*A* $[[*z1* ∧ *z2*]]-*zs*) == *qSwapAbs zs z1 z2 A*

definition
aux-qFresh-ignoreFirst2 ::
'*varSort* ∗ '*var* ∗ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* +
'*varSort* ∗ '*var* ∗ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* ⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTermItem*
where
aux-qFresh-ignoreFirst2 K =
(*case K of Inl*(*zs*,*x*,*X*) ⇒ *termIn X*
|*Inr* (*zs*,*x*,*A*) ⇒ *absIn A*)

lemma *qTermLess-ingoreFirst2-wf*: *wf*(*inv-image qTermLess aux-qFresh-ignoreFirst2*)
using *qTermLess-wf wf-inv-image* **by** *auto*

The quasi absolutely-fresh predicate: (note that this is not an oxymoron: “quasi" refers to being an operator on quasi-terms, and not on terms, i.e., on alpha-equivalence classes; “absolutely” refers to not ignoring bindings in the notion of freshness, and thus counting absolutely all the variables.

function
qAFresh :: '*varSort* ⇒ '*var* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* ⇒ *bool*
and
qAFreshAbs :: '*varSort* ⇒ '*var* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* ⇒ *bool*
where
qAFresh xs x (*qVar ys y*) = (*xs* ≠ *ys* ∨ *x* ≠ *y*)
|
qAFresh xs x (*qOp delta inp binp*) =
(*liftAll* (*qAFresh xs x*) *inp* ∧ *liftAll* (*qAFreshAbs xs x*) *binp*)
|
qAFreshAbs xs x (*qAbs ys y X*) = ((*xs* ≠ *ys* ∨ *x* ≠ *y*) ∧ *qAFresh xs x X*)
by(*pat-completeness*, *auto*)
termination
by(*relation inv-image qTermLess aux-qFresh-ignoreFirst2*,
simp add: *qTermLess-ingoreFirst2-wf*,
auto simp add: *qTermLess-def aux-qFresh-ignoreFirst2-def*)

lemmas *qAFreshAll-simps* = *qAFresh.simps qAFreshAbs.simps*

The next is standard freshness – note that its definition differs from that of absolute freshness only at the clause for abstractions.

function
qFresh :: *'varSort* ⇒ *'var* ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* ⇒ *bool*
and
qFreshAbs :: *'varSort* ⇒ *'var* ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* ⇒ *bool*
where
qFresh xs x (*qVar ys y*) = (*xs* ≠ *ys* ∨ *x* ≠ *y*)
|
qFresh xs x (*qOp delta inp binp*) =
(*liftAll* (*qFresh xs x*) *inp* ∧ *liftAll* (*qFreshAbs xs x*) *binp*)
|
qFreshAbs xs x (*qAbs ys y X*) = ((*xs* = *ys* ∧ *x* = *y*) ∨ *qFresh xs x X*)
by(*pat-completeness*, *auto*)
termination
by(*relation inv-image qTermLess aux-qFresh-ignoreFirst2*,
simp add: *qTermLess-ingoreFirst2-wf*,
auto simp add: *qTermLess-def aux-qFresh-ignoreFirst2-def*)

lemmas *qFreshAll-simps* = *qFresh.simps qFreshAbs.simps*

1.5 Compositional properties of swapping

lemma *qSwapAll-ident*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*
shows (*X* #[[*x* ∧ *x*]]-*zs*) = *X* ∧ (*A* $[[*x* ∧ *x*]]-*zs*) = *A*
by (*induct rule*: *qTerm-rawInduct*)
(*auto simp add*: *liftAll-def lift-cong lift-ident*)

corollary *qSwap-ident*[*simp*]: (*X* #[[*x* ∧ *x*]]-*zs*) = *X*
by(*simp add*: *qSwapAll-ident*)

lemma *qSwapAll-compose*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *zs x y x' y'*
shows
((*X* #[[*x* ∧ *y*]]-*zs*) #[[*x'* ∧ *y'*]]-*zs'*) =
((*X* #[[*x'* ∧ *y'*]]-*zs'*) #[[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]]-*zs*)
∧
((*A* $[[*x* ∧ *y*]]-*zs*) $[[*x'* ∧ *y'*]]-*zs'*) =
((*A* $[[*x'* ∧ *y'*]]-*zs'*) $[[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]]-*zs*)
proof(*induct rule*: *qTerm-rawInduct*[*of - - X A*])
case (*Op delta inp binp*)
then show *?case* **by** (*auto intro*!: *lift-cong simp*: *liftAll-def lift-comp*)
qed (*auto simp add*: *sw-def sw-compose*)

corollary *qSwap-compose*:
$((X \#[[x \wedge y]]\text{-}zs)\ \#[[x' \wedge y']]\text{-}zs') =$
$((X \#[[x' \wedge y']]\text{-}zs')\ \#[[(x\ @zs[x' \wedge y']\text{-}zs') \wedge (y\ @zs[x' \wedge y']\text{-}zs')]]\text{-}zs)$
by (*meson qSwapAll-compose*)

lemma *qSwap-commute*:
assumes $zs \neq zs' \vee \{x,y\}\ Int\ \{x',y'\} = \{\}$
shows $((X \#[[x \wedge y]]\text{-}zs)\ \#[[x' \wedge y']]\text{-}zs') = ((X \#[[x' \wedge y']]\text{-}zs')\ \#[[x \wedge y]]\text{-}zs)$
by (*metis assms disjoint-insert*(*1*) *qSwapAll-compose sw-simps3*)

lemma *qSwapAll-involutive*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *zs x y*
shows $((X \#[[x \wedge y]]\text{-}zs)\ \#[[x \wedge y]]\text{-}zs) = X \wedge$
$((A\ \$[[x \wedge y]]\text{-}zs)\ \$[[x \wedge y]]\text{-}zs) = A$
proof(*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
  case (*Op delta inp binp*)
  then show *?case*
    unfolding *qSwapAll-simps*(*2*) *liftAll-lift-ext*
    lift-comp o-def
    by (*simp add*: *lift-ident*)
qed(*auto*)

corollary *qSwap-involutive*[*simp*]:
$((X \#[[x \wedge y]]\text{-}zs)\ \#[[x \wedge y]]\text{-}zs) = X$
by(*simp add*: *qSwapAll-involutive*)

lemma *qSwapAll-sym*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *zs x y*
shows $(X \#[[x \wedge y]]\text{-}zs) = (X \#[[y \wedge x]]\text{-}zs) \wedge$
$(A\ \$[[x \wedge y]]\text{-}zs) = (A\ \$[[y \wedge x]]\text{-}zs)$
by (*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
  (*auto simp*: *sw-sym lift-comp liftAll-lift-ext*)

corollary *qSwap-sym*:
$(X \#[[x \wedge y]]\text{-}zs) = (X \#[[y \wedge x]]\text{-}zs)$
by(*simp add*: *qSwapAll-sym*)

lemma *qAFreshAll-qSwapAll-id*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *zs z1 z2*
shows $(qAFresh\ zs\ z1\ X \wedge qAFresh\ zs\ z2\ X \longrightarrow (X \#[[z1 \wedge z2]]\text{-}zs) = X) \wedge$
$(qAFreshAbs\ zs\ z1\ A \wedge qAFreshAbs\ zs\ z2\ A \longrightarrow (A\ \$[[z1 \wedge z2]]\text{-}zs) = A)$
by (*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
  (*auto intro*!: *ext simp*: *liftAll-def lift-def option.case-eq-if*)

corollary *qAFresh-qSwap-id*[*simp*]:
$[\![$*qAFresh zs z1 X*; *qAFresh zs z2 X*$]\!]$ $\Longrightarrow$ (*X* #[[*z1* ∧ *z2*]]-*zs*) = *X*
by(*simp add*: *qAFreshAll-qSwapAll-id*)

lemma *qAFreshAll-qSwapAll-compose*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs***and** *zs x y z*
shows (*qAFresh zs y X* ∧ *qAFresh zs z X* ⟶
((*X* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*) = (*X* #[[*z* ∧ *x*]]-*zs*)) ∧
(*qAFreshAbs zs y A* ∧ *qAFreshAbs zs z A* ⟶
((*A* $[[*y* ∧ *x*]]-*zs*) $[[*z* ∧ *y*]]-*zs*) = (*A* $[[*z* ∧ *x*]]-*zs*))
by (*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
(*auto intro*!: *ext simp*: *sw-trans lift-comp lift-def liftAll-def option.case-eq-if*)

corollary *qAFresh-qSwap-compose*:
$[\![$*qAFresh zs y X*; *qAFresh zs z X*$]\!]$ $\Longrightarrow$
((*X* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*) = (*X* #[[*z* ∧ *x*]]-*zs*)
by(*simp add*: *qAFreshAll-qSwapAll-compose*)

1.6 Induction and well-foundedness modulo swapping

lemma *qSkel-qSwapAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *x y zs*
shows *qSkel*(*X* #[[*x* ∧ *y*]]-*zs*) = *qSkel X* ∧
qSkelAbs(*A* $[[*x* ∧ *y*]]-*zs*) = *qSkelAbs A*
proof(*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
case (*Op delta inp binp*)
then show *?case*
unfolding *qSwapAll-simps*(*2*) *liftAll-lift-ext qSkel.simps*(*2*)
lift-comp comp-apply **by** *simp*
qed *auto*

corollary *qSkel-qSwap*: *qSkel*(*X* #[[*x* ∧ *y*]]-*zs*) = *qSkel X*
by(*simp add*: *qSkel-qSwapAll*)

For induction modulo swapping, one may wish to swap not just once, but several times at the induction hypothesis (an example of this will be the proof of compatibility of "qSwap" with alpha) – for this, we introduce the following relation (the suffix "Raw" signifies the fact that the involved variables are not required to be well-sorted):

inductive-set *qSwapped* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm rel*
where
Refl: (*X*,*X*) ∈ *qSwapped*
|
Trans: $[\![$(*X*,*Y*) ∈ *qSwapped*; (*Y*,*Z*) ∈ *qSwapped*$]\!]$ $\Longrightarrow$ (*X*,*Z*) ∈ *qSwapped*
|
Swap: (*X*,*Y*) ∈ *qSwapped* $\Longrightarrow$ (*X*, *Y* #[[*x* ∧ *y*]]-*zs*) ∈ *qSwapped*

lemmas *qSwapped-Clauses* = *qSwapped.Refl qSwapped.Trans qSwapped.Swap*

lemma *qSwap-qSwapped*: $(X,\ X\ \#[[x \wedge y]]\text{-}zs)$: *qSwapped*
by (*auto simp add*: *qSwapped-Clauses*)

lemma *qSwapped-qSkel*:
$(X,Y) \in qSwapped \implies qSkel\ Y = qSkel\ X$
by(*erule qSwapped.induct*, *auto simp add*: *qSkel-qSwap*)

The following is henceforth our main induction principle for quasi-terms. At the clause for abstractions, the user may choose among the two induction hypotheses (IHs):
-(1) IH for all swapped terms
-(2) IH for all terms with the same skeleton.

The user may choose only one of the above, and ignore the others, but may of course also assume both. (2) is stronger than (1), but we offer both of them for convenience in proofs. Most of the times, (1) will be the most convenient.

lemma *qTerm-induct*[*case-names Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *phi phiAbs*
assumes
  Var: $\bigwedge$ *xs x. phi* (*qVar xs x*) **and**
  Op: $\bigwedge$ *delta inp binp.* ⟦*liftAll phi inp*; *liftAll phiAbs binp*⟧
    $\implies$ *phi* (*qOp delta inp binp*) **and**
  Abs: $\bigwedge$ *xs x X.* ⟦$\bigwedge$ *Y*. $(X,Y) \in qSwapped \implies phi\ Y$;
    $\bigwedge$ *Y*. $qSkel\ Y = qSkel\ X \implies phi\ Y$⟧
    $\implies$ *phiAbs* (*qAbs xs x X*)
shows *phi X* $\wedge$ *phiAbs A*
  by (*induct rule*: *qTerm-templateInduct*[*of qSwapped* $\cup$ $\{(X,Y).\ qSkel\ Y = qSkel\ X\}$],
    auto simp add: *qSwapped-qSkel assms*)

The following relation will be needed for proving alpha-equivalence well-defined:

definition *qTermQSwappedLess* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTermItem rel*
where *qTermQSwappedLess* = *qTermLess-modulo qSwapped*

lemma *qTermQSwappedLess-wf*: *wf qTermQSwappedLess*
unfolding *qTermQSwappedLess-def*
using *qSwapped-qSkel qTermLess-modulo-wf*[*of qSwapped*] **by** *blast*

1.7 More properties connecting swapping and freshness

lemma *qSwap-3commute*:
assumes ∗: *qAFresh ys y X* **and** ∗∗: *qAFresh ys y0 X*
and ∗∗∗: $ys \neq zs \vee y0 \notin \{z1,z2\}$

shows ((*X* #[[*z1* ∧ *z2*]]-*zs*) #[[*y0* ∧ *x* @*ys*[*z1* ∧ *z2*]-*zs*]]-*ys*) =
(((*X* #[[*y* ∧ *x*]]-*ys*) #[[*y0* ∧ *y*]]-*ys*) #[[*z1* ∧ *z2*]]-*zs*)
proof−
have *y0* = (*y0* @*ys*[*z1* ∧ *z2*]-*zs*) **using** ∗∗∗ **unfolding** *sw-def* **by** *auto*
hence ((*X* #[[*z1* ∧ *z2*]]-*zs*) #[[*y0* ∧ *x* @*ys*[*z1* ∧ *z2*]-*zs*]]-*ys*) =
((*X* #[[*y0* ∧ *x*]]-*ys*) #[[*z1* ∧ *z2*]]-*zs*)
by(*simp add*: *qSwap-compose*[*of* - *z1*])
also have ((*X* #[[*y0* ∧ *x*]]-*ys*) #[[*z1* ∧ *z2*]]-*zs*) =
(((*X* #[[*y* ∧ *x*]]-*ys*) #[[*y0* ∧ *y*]]-*ys*) #[[*z1* ∧ *z2*]]-*zs*)
using ∗ ∗∗ **by** (*simp add*: *qAFresh-qSwap-compose*)
finally show *?thesis* **.**
qed

lemma *qAFreshAll-imp-qFreshAll*:
fixes *X*::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*qTerm* **and**
A::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*qAbs* **and** *xs x*
shows (*qAFresh xs x X* ⟶ *qFresh xs x X*) ∧
(*qAFreshAbs xs x A* ⟶ *qFreshAbs xs x A*)
by(*induct rule*: *qTerm-rawInduct*, *auto simp add*: *liftAll-def*)

corollary *qAFresh-imp-qFresh*:
qAFresh xs x X ⟹ *qFresh xs x X*
by(*simp add*: *qAFreshAll-imp-qFreshAll*)

lemma *qSwapAll-preserves-qAFreshAll*:
fixes *X*::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*qTerm* **and**
A::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*qAbs* **and** *ys y zs z1 z2*
shows
(*qAFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh ys y X*) ∧
(*qAFreshAbs ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*A* $[[*z1* ∧ *z2*]]-*zs*) = *qAFreshAbs ys y A*)
proof(*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
case (*Op delta inp binp*)
then show *?case*
unfolding *qAFreshAll-simps*(*2*) *qSwapAll-simps*(*2*) *liftAll-lift-comp o-def*
unfolding *liftAll-def* **by** *presburger*
qed(*auto simp add*: *sw-def*)

corollary *qSwap-preserves-qAFresh*[*simp*]:
(*qAFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh ys y X*)
by(*simp add*: *qSwapAll-preserves-qAFreshAll*)

lemma *qSwap-preserves-qAFresh-distinct*:
assumes *ys* ≠ *zs* ∨ *y* ∉ {*z1*,*z2*}
shows *qAFresh ys y* (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh ys y X*
proof−
have *y* = (*y* @*ys*[*z1* ∧ *z2*]-*zs*) **using** *assms* **unfolding** *sw-def* **by** *auto*
thus *?thesis* **using** *qSwap-preserves-qAFresh*[*of ys zs z1 z2 y*] **by** *auto*
qed

lemma *qAFresh-qSwap-exchange1*:
qAFresh zs z2 (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh zs z1 X*
proof−
  have *z2* = (*z1* @*zs*[*z1* ∧ *z2*]-*zs*) **unfolding** *sw-def* **by** *auto*
  thus *?thesis* **using** *qSwap-preserves-qAFresh*[*of zs zs z1 z2 z1 X*] **by** *auto*
qed

lemma *qAFresh-qSwap-exchange2*:
qAFresh zs z2 (*X* #[[*z2* ∧ *z1*]]-*zs*) = *qAFresh zs z1 X*
by(*auto simp add*: *qAFresh-qSwap-exchange1 qSwap-sym*)

lemma *qSwapAll-preserves-qFreshAll*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
    A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and** *ys y zs z1 z2*
shows
(*qFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh ys y X*) ∧
(*qFreshAbs ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*A* $[[*z1* ∧ *z2*]]-*zs*) = *qFreshAbs ys y A*)
proof(*induct rule*: *qTerm-rawInduct*[*of* - - *X A*])
  case (*Op delta inp binp*)
  then show *?case*
   unfolding *qFreshAll-simps*(*2*) *qSwapAll-simps*(*2*) *liftAll-lift-comp o-def*
   unfolding *liftAll-def* **by** *presburger*
qed (*auto simp add*: *sw-def*)

corollary *qSwap-preserves-qFresh*:
(*qFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh ys y X*)
by(*simp add*: *qSwapAll-preserves-qFreshAll*)

lemma *qSwap-preserves-qFresh-distinct*:
assumes *ys* ≠ *zs* ∨ *y* ∉ {*z1*,*z2*}
shows *qFresh ys y* (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh ys y X*
proof−
  have *y* = (*y* @*ys*[*z1* ∧ *z2*]-*zs*) **using** *assms* **unfolding** *sw-def* **by** *auto*
  thus *?thesis* **using** *qSwap-preserves-qFresh*[*of ys zs z1 z2 y*] **by** *auto*
qed

lemma *qFresh-qSwap-exchange1*:
qFresh zs z2 (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh zs z1 X*
proof−
  have *z2* = (*z1* @*zs*[*z1* ∧ *z2*]-*zs*) **unfolding** *sw-def* **by** *auto*
  thus *?thesis* **using** *qSwap-preserves-qFresh*[*of zs zs z1 z2 z1 X*] **by** *auto*
qed

lemma *qFresh-qSwap-exchange2*:
qFresh zs z1 X = *qFresh zs z2* (*X* #[[*z2* ∧ *z1*]]-*zs*)
by (*auto simp add*: *qFresh-qSwap-exchange1 qSwap-sym*)

lemmas *qSwap-qAFresh-otherSimps* =
qSwap-ident qSwap-involutive qAFresh-qSwap-id qSwap-preserves-qAFresh

end

2 Availability of Fresh Variables and Alpha-Equivalence

theory *QuasiTerms-PickFresh-Alpha*
imports *QuasiTerms-Swap-Fresh*

begin

Here we define good quasi-terms and alpha-equivalence on quasi-terms, and prove relevant properties such as the ability to pick fresh variables for good quasi-terms and the fact that alpha is indeed an equivalence and is compatible with all the operators.

We do most of the work on freshness and alpha-equivalence unsortedly, for raw quasi-terms. (And we do it in such a way that it then applies immediately to sorted quasi-terms.) We do need sortedness of variables (as well as a cardinality assumption), however, for alpha-equivalence to have the desired properties. Therefore we work in a locale.

2.1 The FixVars locale

definition *var-infinite* **where**
var-infinite (- :: *'var*) ==
 infinite (*UNIV* :: *'var set*)

definition *var-regular* **where**
var-regular (- :: *'var*) ==
 regular |*UNIV* :: *'var set*|

definition *varSort-lt-var* **where**
varSort-lt-var (- :: *'varSort*) (- :: *'var*) ==
 |*UNIV* :: *'varSort set*| <*o* |*UNIV* :: *'var set*|

locale *FixVars* =
  fixes *dummyV* :: *'var* **and** *dummyVS* :: *'varSort*
  assumes *var-infinite*: *var-infinite* (*undefined* :: *'var*)
  and *var-regular*: *var-regular* (*undefined* :: *'var*)
  and *varSort-lt-var*: *varSort-lt-var* (*undefined* :: *'varSort*) (*undefined* :: *'var*)

context *FixVars*
begin

lemma *varSort-lt-var-INNER*:
|*UNIV* :: *'varSort set*| <*o* |*UNIV* :: *'var set*|
using *varSort-lt-var*

unfolding *varSort-lt-var-def* **by** *simp*

lemma *varSort-le-Var*:
$|UNIV :: 'varSort\ set| \leq o\ |UNIV :: 'var\ set|$
using *varSort-lt-var-INNER ordLess-imp-ordLeq* **by** *auto*

theorem *var-infinite-INNER*: *infinite* (*UNIV* :: *'var set*)
using *var-infinite* **unfolding** *var-infinite-def* **by** *simp*

theorem *var-regular-INNER*: *regular* $|UNIV :: 'var\ set|$
using *var-regular* **unfolding** *var-regular-def* **by** *simp*

theorem *infinite-var-regular-INNER*:
infinite (*UNIV* :: *'var set*) $\wedge$ *regular* $|UNIV :: 'var\ set|$
by (*simp add*: *var-infinite-INNER var-regular-INNER*)

theorem *finite-ordLess-var*:
$(|S| <o\ |UNIV :: 'var\ set| \vee finite\ S) = (|S| <o\ |UNIV :: 'var\ set|)$
by (*auto simp add*: *var-infinite-INNER finite-ordLess-infinite2*)

2.2 Good quasi-terms

Essentially, good quasi-term items will be those with meaningful binders and not too many variables. Good quasi-terms are a concept intermediate between (raw) quasi-terms and sorted quasi-terms. This concept was chosen to be strong enough to facilitate proofs of most of the desired properties of alpha-equivalence, avoiding, *for most of the hard part of the work*, the overhead of sortedness. Since we later prove that quasi-terms are good, all the results are then immediately transported to a sorted setting.

function
qGood :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* $\Rightarrow$ *bool*
and
qGoodAbs :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* $\Rightarrow$ *bool*
where
qGood (*qVar xs x*) = *True*
|
qGood (*qOp delta inp binp*) =
(*liftAll qGood inp* $\wedge$ *liftAll qGoodAbs binp* $\wedge$
$|\{i.\ inp\ i \neq None\}| <o\ |UNIV :: 'var\ set| \wedge$
$|\{i.\ binp\ i \neq None\}| <o\ |UNIV :: 'var\ set|$)
|
qGoodAbs (*qAbs xs x X*) = *qGood X*
by (*pat-completeness*, *auto*)
termination
apply(*relation qTermLess*)
apply(*simp-all add*: *qTermLess-wf*)

by(*auto simp add*: *qTermLess-def*)

fun *qGoodItem* :: (*'index,'bindex,'varSort,'var,'opSym*)*qTermItem* ⇒ *bool* **where**
qGoodItem (*Inl qX*) = *qGood qX*
|
qGoodItem (*Inr qA*) = *qGoodAbs qA*

lemma *qSwapAll-preserves-qGoodAll1*:
fixes *X*::(*'index,'bindex,'varSort,'var,'opSym*)*qTerm* **and**
A::(*'index,'bindex,'varSort,'var,'opSym*)*qAbs* **and** *zs x y*
shows
(*qGood X* ⟶ *qGood* (*X* #[[*x* ∧ *y*]]-*zs*)) ∧
(*qGoodAbs A* ⟶ *qGoodAbs* (*A* $[[*x* ∧ *y*]]-*zs*))
apply(*rule qTerm-induct*[*of* - - *X A*])
apply(*simp-all add*: *sw-def*)
unfolding *lift-def liftAll-def* **apply** *auto*
apply(*case-tac inp i*, *auto*)
apply(*case-tac binp i*, *auto*)
proof−
fix *inp*::(*'index*,(*'index,'bindex,'varSort,'var,'opSym*)*qTerm*)*input* **and** *zs xs x y*
let *?K1* = {*i*. ∃ *X*. *inp i* = *Some X*}
let *?K2* = {*i*. ∃ *X*. (*case inp i of None* ⇒ *None* | *Some X* ⇒ *Some* (*X* #[[*x* ∧ *y*]]-*zs*))
= *Some X*}
assume |*?K1*| <*o* |*UNIV* :: *'var set*|
moreover have *?K1* = *?K2* **by**(*auto*, *case-tac inp x*, *auto*)
ultimately show |*?K2*| <*o* |*UNIV* :: *'var set*| **by** *simp*
next
fix *binp*::(*'bindex*,(*'index,'bindex,'varSort,'var,'opSym*)*qAbs*)*input* **and** *zs xs x y*
let *?K1* = {*i*. ∃ *A*. *binp i* = *Some A*}
let *?K2* = {*i*. ∃ *A*. (*case binp i of None* ⇒ *None* | *Some A* ⇒ *Some* (*A* $[[*x* ∧ *y*]]-*zs*))
= *Some A*}
assume |*?K1*| <*o* |*UNIV* :: *'var set*|
moreover have *?K1* = *?K2* **by**(*auto*, *case-tac binp x*, *auto*)
ultimately show |*?K2*| <*o* |*UNIV* :: *'var set*| **by** *simp*
qed

corollary *qSwap-preserves-qGood1*:
qGood X ⟹ *qGood* (*X* #[[*x* ∧ *y*]]-*zs*)
by(*simp add*: *qSwapAll-preserves-qGoodAll1*)

corollary *qSwapAbs-preserves-qGoodAbs1*:
qGoodAbs A ⟹ *qGoodAbs* (*A* $[[*x* ∧ *y*]]-*zs*)
by(*simp add*: *qSwapAll-preserves-qGoodAll1*)

lemma *qSwap-preserves-qGood2*:
assumes *qGood*(*X* #[[*x* ∧ *y*]]-*zs*)
shows *qGood X*

by (*metis assms qSwap-involutive qSwap-preserves-qGood1*)

lemma *qSwapAbs-preserves-qGoodAbs2*:
assumes *qGoodAbs*(*A* $[[*x* ∧ *y*]]-*zs*)
shows *qGoodAbs A*
by (*metis assms qSwapAbs-preserves-qGoodAbs1 qSwapAll-involutive*)

lemma *qSwap-preserves-qGood*: (*qGood* (*X* #[[*x* ∧ *y*]]-*zs*)) = (*qGood X*)
using *qSwap-preserves-qGood1 qSwap-preserves-qGood2* **by** *blast*

lemma *qSwapAbs-preserves-qGoodAbs*:
(*qGoodAbs* (*A* $[[*x* ∧ *y*]]-*zs*)) = (*qGoodAbs A*)
using *qSwapAbs-preserves-qGoodAbs1 qSwapAbs-preserves-qGoodAbs2* **by** *blast*

lemma *qSwap-twice-preserves-qGood*:
(*qGood* ((*X* #[[*x* ∧ *y*]]-*zs*) #[[*x'* ∧ *y'*]]-*zs'*)) = (*qGood X*)
by (*simp add*: *qSwap-preserves-qGood*)

lemma *qSwapped-preserves-qGood*:
$(X,Y) \in$ *qSwapped* ⟹ *qGood Y* = *qGood X*
apply (*induct rule*: *qSwapped.induct*)
using *qSwap-preserves-qGood* **by** *auto*

lemma *qGood-qTerm-templateInduct*[*case-names Rel Var Op Abs*]:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *phi phiAbs rel*
assumes
REL: ⋀ *X Y*. ⟦*qGood X*; $(X,Y) \in$ *rel*⟧ ⟹ *qGood Y* ∧ *qSkel Y* = *qSkel X* **and**
Var: ⋀ *xs x*. *phi* (*qVar xs x*) **and**
Op: ⋀ *delta inp binp*. ⟦|{*i*. *inp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
|{*i*. *binp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
liftAll (λ*X*. *qGood X* ∧ *phi X*) *inp*;
liftAll (λ*A*. *qGoodAbs A* ∧ *phiAbs A*) *binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: ⋀ *xs x X*. ⟦*qGood X*; ⋀ *Y*. $(X,Y) \in$ *rel* ⟹ *phi Y*⟧
⟹ *phiAbs* (*qAbs xs x X*)
shows
(*qGood X* ⟶ *phi X*) ∧ (*qGoodAbs A* ⟶ *phiAbs A*)
apply(*induct rule*: *qTerm-templateInduct*[*of* {(X,Y). *qGood X* ∧ $(X,Y) \in$ *rel*}])
using *assms* **by** (*simp-all add*: *liftAll-def*)

lemma *qGood-qTerm-rawInduct*[*case-names Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *phi phiAbs*
assumes
Var: ⋀ *xs x*. *phi* (*qVar xs x*) **and**
Op: ⋀ *delta inp binp*. ⟦|{*i*. *inp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
|{*i*. *binp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
liftAll (λ *X*. *qGood X* ∧ *phi X*) *inp*;

liftAll (λ *A*. *qGoodAbs A* ∧ *phiAbs A*) *binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: ⋀ *xs x X*. ⟦*qGood X*; *phi X*⟧ ⟹ *phiAbs* (*qAbs xs x X*)
shows (*qGood X* ⟶ *phi X*) ∧ (*qGoodAbs A* ⟶ *phiAbs A*)
apply(*induct rule*: *qGood-qTerm-templateInduct* [*of Id*])
by(*simp-all add*: *assms*)

lemma *qGood-qTerm-induct*[*case-names Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *phi phiAbs*
assumes
Var: ⋀ *xs x*. *phi* (*qVar xs x*) **and**
Op: ⋀ *delta inp binp*. ⟦|{*i*. *inp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
|{*i*. *binp i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|;
liftAll (λ *X*. *qGood X* ∧ *phi X*) *inp*;
liftAll (λ *A*. *qGoodAbs A* ∧ *phiAbs A*) *binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: ⋀ *xs x X*. ⟦*qGood X*;
⋀ *Y*. *qGood Y* ∧ *qSkel Y* = *qSkel X* ⟹ *phi Y*;
⋀ *Y*. (*X*,*Y*) ∈ *qSwapped* ⟹ *phi Y*⟧
⟹ *phiAbs* (*qAbs xs x X*)
shows
(*qGood X* ⟶ *phi X*) ∧ (*qGoodAbs A* ⟶ *phiAbs A*)
apply(*induct rule*: *qGood-qTerm-templateInduct*
[*of qSwapped* ∪ {(*X*,*Y*). *qGood Y* ∧ *qSkel Y* = *qSkel X*}])
using *qSwapped-qSkel qSwapped-preserves-qGood*
by(*auto simp add*: *assms*)

A form specialized for mutual induction (this time, without the cardinality hypotheses):

lemma *qGood-qTerm-induct-mutual*[*case-names Var1 Var2 Op1 Op2 Abs1 Abs2*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
and *A*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *phi1 phi2 phiAbs1 phiAbs2*
assumes
Var1: ⋀ *xs x*. *phi1* (*qVar xs x*) **and**
Var2: ⋀ *xs x*. *phi2* (*qVar xs x*) **and**
Op1: ⋀ *delta inp binp*. ⟦*liftAll* (λ *X*. *qGood X* ∧ *phi1 X*) *inp*;
liftAll (λ *A*. *qGoodAbs A* ∧ *phiAbs1 A*) *binp*⟧
⟹ *phi1* (*qOp delta inp binp*) **and**
Op2: ⋀ *delta inp binp*. ⟦*liftAll* (λ *X*. *qGood X* ∧ *phi2 X*) *inp*;
liftAll (λ *A*. *qGoodAbs A* ∧ *phiAbs2 A*) *binp*⟧
⟹ *phi2* (*qOp delta inp binp*) **and**
Abs1: ⋀ *xs x X*. ⟦*qGood X*;
⋀ *Y*. *qGood Y* ∧ *qSkel Y* = *qSkel X* ⟹ *phi1 Y*;
⋀ *Y*. *qGood Y* ∧ *qSkel Y* = *qSkel X* ⟹ *phi2 Y*;
⋀ *Y*. (*X*,*Y*) ∈ *qSwapped* ⟹ *phi1 Y*;
⋀ *Y*. (*X*,*Y*) ∈ *qSwapped* ⟹ *phi2 Y*⟧
⟹ *phiAbs1* (*qAbs xs x X*) **and**
Abs2: ⋀ *xs x X*. ⟦*qGood X*;

$\bigwedge$ *Y*. *qGood Y* ∧ *qSkel Y* = *qSkel X* ⟹ *phi1 Y*;
$\bigwedge$ *Y*. *qGood Y* ∧ *qSkel Y* = *qSkel X* ⟹ *phi2 Y*;
$\bigwedge$ *Y*. (*X*,*Y*) ∈ *qSwapped* ⟹ *phi1 Y*;
$\bigwedge$ *Y*. (*X*,*Y*) ∈ *qSwapped* ⟹ *phi2 Y*;
phiAbs1 (*qAbs xs x X*)⟧
⟹ *phiAbs2* (*qAbs xs x X*)
shows
(*qGood X* ⟶ (*phi1 X* ∧ *phi2 X*)) ∧
(*qGoodAbs A* ⟶ (*phiAbs1 A* ∧ *phiAbs2 A*))
apply(*induct rule*: *qGood-qTerm-induct*[*of* - - *X A*])
by(*auto simp add*: *assms liftAll-and*)

2.3 The ability to pick fresh variables

lemma *single-non-qAFreshAll-ordLess-var*:
fixes *X* :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
and *A*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
shows
(*qGood X* ⟶ |{*x*. ¬ *qAFresh xs x X*}| <*o* |*UNIV* :: '*var set*|) ∧
(*qGoodAbs A* ⟶ |{*x*. ¬ *qAFreshAbs xs x A*}| <*o* |*UNIV* :: '*var set*|)
proof(*induct rule*: *qGood-qTerm-rawInduct*)
case (*Var xs x*)
then show *?case* **using** *infinite-var-regular-INNER* **by** *simp*
next
case (*Op delta inp binp*)
let *?Left* = {*x*. ¬ *qAFresh xs x* (*qOp delta inp binp*)}
obtain *J* **where** *J-def*: *J* = {*i*. ∃ *X*. *inp i* = *Some X*} **by** *blast*
let *?S* = ⋃ *i* ∈ *J*. {*x*. ∃ *X*. *inp i* = *Some X* ∧ ¬ *qAFresh xs x X*}
{**fix** *i*
obtain *K* **where** *K-def*: *K* = {*X*. *inp i* = *Some X*} **by** *blast*
have *finite K* **unfolding** *K-def* **by** (*cases inp i*, *auto*)
hence |*K*| <*o* |*UNIV* :: '*var set*| **using** *var-infinite-INNER finite-ordLess-infinite2*
by *auto*
moreover have ∀ *X* ∈ *K*. |{*x*. ¬ *qAFresh xs x X*}| <*o* |*UNIV* :: '*var set*|
unfolding *K-def* **using** *Op* **unfolding** *liftAll-def* **by** *simp*
ultimately have |⋃ *X* ∈ *K*. {*x*. ¬ *qAFresh xs x X*}| <*o* |*UNIV* :: '*var set*|
using *var-regular-INNER* **by** (*simp add*: *regular-UNION*)
moreover
have {*x*. ∃ *X*. *inp i* = *Some X* ∧ ¬ *qAFresh xs x X*} =
(⋃ *X* ∈ *K*. {*x*. ¬ *qAFresh xs x X*}) **unfolding** *K-def* **by** *blast*
ultimately
have |{*x*. ∃ *X*. *inp i* = *Some X* ∧ ¬ *qAFresh xs x X*}| <*o* |*UNIV* :: '*var set*|
by *simp*
}
moreover have |*J*| <*o* |*UNIV* :: '*var set*| **unfolding** *J-def*
using *Op* **unfolding** *liftAll-def* **by** *simp*
ultimately
have *1*: |*?S*| <*o* |*UNIV* :: '*var set*|
using *var-regular-INNER* **by** (*simp add*: *regular-UNION*)

obtain *Ja* **where** *Ja-def*: $Ja = \{i.\ \exists\ A.\ binp\ i = Some\ A\}$ **by** *blast*
let $?Sa = \bigcup\ i \in Ja.\ \{x.\ \exists\ A.\ binp\ i = Some\ A \wedge \neg\ qAFreshAbs\ xs\ x\ A\}$
{fix *i*
obtain *K* **where** *K-def*: $K = \{A.\ binp\ i = Some\ A\}$ **by** *blast*
have *finite K* **unfolding** *K-def* **by** (*cases binp i*, *auto*)
hence $|K| <o |UNIV :: 'var\ set|$ **using** *var-infinite-INNER finite-ordLess-infinite2*
by *auto*
moreover have $\forall\ A \in K.\ |\{x.\ \neg\ qAFreshAbs\ xs\ x\ A\}| <o |UNIV :: 'var\ set|$
unfolding *K-def* **using** *Op* **unfolding** *liftAll-def* **by** *simp*
ultimately have $|\bigcup\ A \in K.\ \{x.\ \neg\ qAFreshAbs\ xs\ x\ A\}| <o |UNIV :: 'var\ set|$
using *var-regular-INNER* **by** (*simp add*: *regular-UNION*)
moreover
have $\{x.\ \exists A.\ binp\ i = Some\ A \wedge \neg\ qAFreshAbs\ xs\ x\ A\} =$
$(\bigcup\ A \in K.\ \{x.\ \neg\ qAFreshAbs\ xs\ x\ A\})$ **unfolding** *K-def* **by** *blast*
ultimately
have $|\{x.\ \exists A.\ binp\ i = Some\ A \wedge \neg\ qAFreshAbs\ xs\ x\ A\}| <o |UNIV :: 'var\ set|$
by *simp*
}
moreover have $|Ja| <o |UNIV :: 'var\ set|$
unfolding *Ja-def* **using** *Op* **unfolding** *liftAll-def* **by** *simp*
ultimately have $|?Sa| <o |UNIV :: 'var\ set|$
using *var-regular-INNER* **by** (*simp add*: *regular-UNION*)
with *1* **have** $|?S\ Un\ ?Sa| <o |UNIV :: 'var\ set|$
using *var-infinite-INNER card-of-Un-ordLess-infinite* **by** *auto*
moreover have $?Left = ?S\ Un\ ?Sa$
by (*auto simp*: *J-def Ja-def liftAll-def*)
ultimately show *?case* **by** *simp*
next
case (*Abs xsa x X*)
let $?Left = \{xa.\ xs = xsa \wedge xa = x \vee \neg\ qAFresh\ xs\ xa\ X\}$
have $|\{x\}| <o |UNIV :: 'var\ set|$ **by** (*auto simp add*: *var-infinite-INNER*)
hence $|\{x\} \cup \{x.\ \neg\ qAFresh\ xs\ x\ X\}| <o |UNIV :: 'var\ set|$
using *Abs var-infinite-INNER card-of-Un-ordLess-infinite* **by** *blast*
moreover
{have $?Left \subseteq \{x\} \cup \{x.\ \neg\ qAFresh\ xs\ x\ X\}$ **by** *blast*
hence $|?Left| \leq o\ |\{x\} \cup \{x.\ \neg\ qAFresh\ xs\ x\ X\}|$ **using** *card-of-mono1* **by** *auto*
}
ultimately show *?case* **using** *ordLeq-ordLess-trans* **by** *auto*
qed

corollary *single-non-qAFresh-ordLess-var*:
$qGood\ X \Longrightarrow |\{x.\ \neg\ qAFresh\ xs\ x\ X\}| <o |UNIV :: 'var\ set|$
by(*simp add*: *single-non-qAFreshAll-ordLess-var*)

corollary *single-non-qAFreshAbs-ordLess-var*:
$qGoodAbs\ A \Longrightarrow |\{x.\ \neg\ qAFreshAbs\ xs\ x\ A\}| <o |UNIV :: 'var\ set|$
by(*simp add*: *single-non-qAFreshAll-ordLess-var*)

lemma *single-non-qFresh-ordLess-var*:
assumes *qGood X*
shows $|\{x.\ \neg\ qFresh\ xs\ x\ X\}|$ $<o$ $|UNIV :: 'var\ set|$
using *qAFresh-imp-qFresh card-of-mono1 single-non-qAFresh-ordLess-var ordLeq-ordLess-trans* **by** (*metis Collect-mono assms*)

lemma *single-non-qFreshAbs-ordLess-var*:
assumes *qGoodAbs A*
shows $|\{x.\ \neg\ qFreshAbs\ xs\ x\ A\}|$ $<o$ $|UNIV :: 'var\ set|$
using *qAFreshAll-imp-qFreshAll card-of-mono1 single-non-qAFreshAbs-ordLess-var ordLeq-ordLess-trans* **by** (*metis Collect-mono assms*)

lemma *non-qAFresh-ordLess-var*:
assumes *GOOD*: $\forall\ X \in XS.\ qGood\ X$ **and** *Var*: $|XS|$ $<o$ $|UNIV :: 'var\ set|$
shows $|\{x|\ x\ X.\ X \in XS \wedge \neg\ qAFresh\ xs\ x\ X\}|$ $<o$ $|UNIV :: 'var\ set|$
proof−
  define *K* **and** *F* **where** $K \equiv \{x|\ x\ X.\ X \in XS \wedge \neg\ qAFresh\ xs\ x\ X\}$
  and $F \equiv (\lambda\ X.\ \{x.\ X \in XS \wedge \neg\ qAFresh\ xs\ x\ X\})$
  have $K = (\bigcup\ X \in XS.\ F\ X)$ **unfolding** *K-def F-def* **by** *auto*
  moreover have $\forall\ X \in XS.\ |F\ X|$ $<o$ $|UNIV :: 'var\ set|$
  unfolding *F-def* **using** *GOOD single-non-qAFresh-ordLess-var* **by** *auto*
  ultimately have $|K|$ $<o$ $|UNIV :: 'var\ set|$ **using** *var-regular-INNER Var*
  by(*auto simp add*: *regular-UNION*)
  thus *?thesis* **unfolding** *K-def* **.**
qed

lemma *non-qAFresh-or-in-ordLess-var*:
assumes *Var*: $|V|$ $<o$ $|UNIV :: 'var\ set|$ **and** $|XS|$ $<o$ $|UNIV :: 'var\ set|$ **and** $\forall$ $X \in XS.\ qGood\ X$
shows $|\{x|\ x\ X.\ (x \in V \vee (X \in XS \wedge \neg\ qAFresh\ xs\ x\ X))\}|$ $<o$ $|UNIV :: 'var\ set|$
proof−
  define *J* **and** *K* **where** $J \equiv \{x|\ x\ X.\ (x \in V \vee (X \in XS \wedge \neg\ qAFresh\ xs\ x$ $X))\}$
  and $K \equiv \{x|\ x\ X.\ X \in XS \wedge \neg\ qAFresh\ xs\ x\ X\}$
  have $J \subseteq K \cup V$ **unfolding** *J-def K-def* **by** *auto*
  hence $|J|$ $\leq o$ $|K \cup V|$ **using** *card-of-mono1* **by** *auto*
  moreover
  {**have** $|K|$ $<o$ $|UNIV :: 'var\ set|$ **unfolding** *K-def* **using** *assms non-qAFresh-ordLess-var* **by** *auto*
  hence $|K \cup V|$ $<o$ $|UNIV :: 'var\ set|$ **using** *Var var-infinite-INNER card-of-Un-ordLess-infinite* **by** *auto*
  }
  ultimately have $|J|$ $<o$ $|UNIV :: 'var\ set|$ **using** *ordLeq-ordLess-trans* **by** *blast*
  thus *?thesis* **unfolding** *J-def* **.**
qed

lemma *obtain-set-qFresh-card-of*:
assumes $|V|$ $<o$ $|UNIV :: 'var\ set|$ **and** $|XS|$ $<o$ $|UNIV :: 'var\ set|$ **and** $\forall$ $X \in XS.\ qGood\ X$

shows ∃ *W*. *infinite W* ∧ *W Int V* = {} ∧
(∀ *x* ∈ *W*. ∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
proof−
define *J* **where** *J* ≡ {*x*| *x X*. (*x* ∈ *V* ∨ (*X* ∈ *XS* ∧ ¬ *qAFresh xs x X*))}
let *?W* = *UNIV* − *J*
have |*J*| <*o* |*UNIV* :: ′*var set*|
unfolding *J-def* **using** *assms non-qAFresh-or-in-ordLess-var* **by** *auto*
hence *infinite ?W* **using** *var-infinite-INNER subset-ordLeq-diff-infinite*[*of* - *J*]
by *auto*
moreover
have *?W* ∩ *V* = {} ∧ (∀ *x* ∈ *?W*. ∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
unfolding *J-def* **using** *qAFresh-imp-qFresh* **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

lemma *obtain-set-qFresh*:
assumes *finite V* ∨ |*V*| <*o* |*UNIV* :: ′*var set*| **and** *finite XS* ∨ |*XS*| <*o* |*UNIV*
:: ′*var set*| **and**
∀ *X* ∈ *XS*. *qGood X*
shows ∃ *W*. *infinite W* ∧ *W Int V* = {} ∧
(∀ *x* ∈ *W*. ∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
using *assms*
by(*fastforce simp add*: *var-infinite-INNER obtain-set-qFresh-card-of*)

lemma *obtain-qFresh-card-of*:
assumes |*V*| <*o* |*UNIV* :: ′*var set*| **and** |*XS*| <*o* |*UNIV* :: ′*var set*| **and** ∀ *X* ∈
XS. *qGood X*
shows ∃ *x*. *x* ∉ *V* ∧ (∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
proof−
obtain *W* **where** *infinite W* **and**
∗: *W* ∩ *V* = {} ∧ (∀ *x* ∈ *W*. ∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
using *assms obtain-set-qFresh-card-of* **by** *blast*
then obtain *x* **where** *x* ∈ *W* **using** *infinite-imp-nonempty* **by** *fastforce*
thus *?thesis* **using** ∗ **by** *auto*
qed

lemma *obtain-qFresh*:
assumes *finite V* ∨ |*V*| <*o* |*UNIV* :: ′*var set*| **and** *finite XS* ∨ |*XS*| <*o* |*UNIV*
:: ′*var set*| **and**
∀ *X* ∈ *XS*. *qGood X*
shows ∃ *x*. *x* ∉ *V* ∧ (∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)
using *assms*
by(*fastforce simp add*: *var-infinite-INNER obtain-qFresh-card-of*)

definition *pickQFresh* **where**
pickQFresh xs V XS ==
SOME x. *x* ∉ *V* ∧ (∀ *X* ∈ *XS*. *qAFresh xs x X* ∧ *qFresh xs x X*)

lemma *pickQFresh-card-of*:

assumes $|V| <o\ |UNIV :: 'var\ set|$ **and** $|XS| <o\ |UNIV :: 'var\ set|$ **and** $\forall\ X \in XS.\ qGood\ X$
shows $pickQFresh\ xs\ V\ XS \notin V \wedge$
$(\forall\ X \in XS.\ qAFresh\ xs\ (pickQFresh\ xs\ V\ XS)\ X \wedge qFresh\ xs\ (pickQFresh\ xs\ V\ XS)\ X)$
unfolding *pickQFresh-def* **apply**(*rule someI-ex*)
using *assms obtain-qFresh-card-of* **by** *blast*

lemma *pickQFresh*:
assumes $finite\ V \vee |V| <o\ |UNIV :: 'var\ set|$ **and** $finite\ XS \vee |XS| <o\ |UNIV :: 'var\ set|$ **and**
$\forall\ X \in XS.\ qGood\ X$
shows $pickQFresh\ xs\ V\ XS \notin V \wedge$
$(\forall\ X \in XS.\ qAFresh\ xs\ (pickQFresh\ xs\ V\ XS)\ X \wedge qFresh\ xs\ (pickQFresh\ xs\ V\ XS)\ X)$
unfolding *pickQFresh-def* **apply**(*rule someI-ex*)
using *assms* **by**(*auto simp add*: *obtain-qFresh*)

end

2.4 Alpha-equivalence

2.4.1 Definition

definition *aux-alpha-ignoreSecond* ::
$('index,'bindex,'varSort,'var,'opSym)qTerm * ('index,'bindex,'varSort,'var,'opSym)qTerm$
$+$
$('index,'bindex,'varSort,'var,'opSym)qAbs * ('index,'bindex,'varSort,'var,'opSym)qAbs$
$\Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qTermItem$
where
$aux\text{-}alpha\text{-}ignoreSecond\ K ==$
$case\ K\ of\ Inl(X,Y) \Rightarrow termIn\ X$
$|Inr(A,B) \Rightarrow absIn\ A$

lemma *aux-alpha-ignoreSecond-qTermLessQSwapped-wf*:
wf(*inv-image qTermQSwappedLess aux-alpha-ignoreSecond*)
using *qTermQSwappedLess-wf wf-inv-image* **by** *auto*

function
alpha **and** *alphaAbs*
where
$alpha\ (qVar\ xs\ x)\ (qVar\ xs'\ x') \longleftrightarrow xs = xs' \wedge x = x'$
|
$alpha\ (qOp\ delta\ inp\ binp)\ (qOp\ delta'\ inp'\ binp') \longleftrightarrow$
$delta = delta' \wedge sameDom\ inp\ inp' \wedge sameDom\ binp\ binp' \wedge$
$liftAll2\ alpha\ inp\ inp' \wedge$
$liftAll2\ alphaAbs\ binp\ binp'$
|

alpha (*qVar xs x*) (*qOp delta′ inp′ binp′*) ⟷ *False*
|
alpha (*qOp delta inp binp*) (*qVar xs′ x′*) ⟷ *False*
|
alphaAbs (*qAbs xs x X*) (*qAbs xs′ x′ X′*) ⟷
xs = *xs′* ∧
(∃ *y*. *y* ∉ {*x*,*x′*} ∧ *qAFresh xs y X* ∧ *qAFresh xs′ y X′* ∧
alpha (*X* #[[*y* ∧ *x*]]-*xs*) (*X′* #[[*y* ∧ *x′*]]-*xs′*))
by(*pat-completeness*, *auto*)
termination
apply(*relation inv-image qTermQSwappedLess aux-alpha-ignoreSecond*)
apply(*simp add*: *aux-alpha-ignoreSecond-qTermLessQSwapped-wf*)
by(*auto simp add*: *qTermQSwappedLess-def qTermLess-modulo-def*
aux-alpha-ignoreSecond-def qSwap-qSwapped)

abbreviation *alpha-abbrev* (**infix** ‹#=› *50*) **where** *X* #= *Y* ≡ *alpha X Y*
abbreviation *alphaAbs-abbrev* (**infix** ‹$=› *50*) **where** *A* $= *B* ≡ *alphaAbs A B*

context *FixVars*
begin

2.4.2 Simplification and elimination rules

lemma *alpha-inp-None*:
qOp delta inp binp #= *qOp delta′ inp′ binp′* ⟹
(*inp i* = *None*) = (*inp′ i* = *None*)
by(*auto simp add*: *sameDom-def*)

lemma *alpha-binp-None*:
qOp delta inp binp #= *qOp delta′ inp′ binp′* ⟹
(*binp i* = *None*) = (*binp′ i* = *None*)
by(*auto simp add*: *sameDom-def*)

lemma *qVar-alpha-iff*:
qVar xs x #= *X′* ⟷ *X′* = *qVar xs x*
by(*cases X′*, *auto*)

lemma *alpha-qVar-iff*:
X #= *qVar xs′ x′* ⟷ *X* = *qVar xs′ x′*
by(*cases X*, *auto*)

lemma *qOp-alpha-iff*:
qOp delta inp binp #= *X′* ⟷
(∃ *inp′ binp′*.
X′ = *qOp delta inp′ binp′* ∧ *sameDom inp inp′* ∧ *sameDom binp binp′* ∧
liftAll2 (λ*Y Y′*. *Y* #= *Y′*) *inp inp′* ∧
liftAll2 (λ*A A′*. *A* $= *A′*) *binp binp′*)
by(*cases X′*) *auto*

lemma *alpha-qOp-iff*:
$X \ \#= \ qOp \ delta' \ inp' \ binp' \longleftrightarrow$
$(\exists \ inp \ binp. \ X = qOp \ delta' \ inp \ binp \wedge sameDom \ inp \ inp' \wedge sameDom \ binp \ binp'$
$\wedge$
$liftAll2 \ (\lambda Y \ Y'. \ Y \ \#= \ Y') \ inp \ inp' \wedge$
$liftAll2 \ (\lambda A \ A'. \ A \ \$= \ A') \ binp \ binp')$
by(*cases X*) *auto*

lemma *qAbs-alphaAbs-iff*:
$qAbs \ xs \ x \ X \ \$= \ A' \longleftrightarrow$
$(\exists \ x' \ y \ X'. \ A' = qAbs \ xs \ x' \ X' \wedge$
$y \notin \{x,x'\} \wedge qAFresh \ xs \ y \ X \wedge qAFresh \ xs \ y \ X' \wedge$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= \ (X' \ \#[[y \wedge x']]\text{-}xs))$
by(*cases A'*) *auto*

lemma *alphaAbs-qAbs-iff*:
$A \ \$= \ qAbs \ xs' \ x' \ X' \longleftrightarrow$
$(\exists \ x \ y \ X. \ A = qAbs \ xs' \ x \ X \wedge$
$y \notin \{x,x'\} \wedge qAFresh \ xs' \ y \ X \wedge qAFresh \ xs' \ y \ X' \wedge$
$(X \ \#[[y \wedge x]]\text{-}xs') \ \#= \ (X' \ \#[[y \wedge x']]\text{-}xs'))$
by(*cases A*) *auto*

2.4.3 Basic properties

In a nutshell: “alpha" is included in the kernel of “qSkel", is an equivalence on good quasi-terms, preserves goodness, and all operators and relations (except “qAFresh") preserve alpha.

lemma *alphaAll-qSkelAll*:
fixes X::('index,'bindex,'varSort,'var,'opSym)qTerm **and**
A::('index,'bindex,'varSort,'var,'opSym)qAbs
shows
$(\forall \ X'. \ X \ \#= \ X' \longrightarrow qSkel \ X = qSkel \ X') \wedge$
$(\forall \ A'. \ A \ \$= \ A' \longrightarrow qSkelAbs \ A = qSkelAbs \ A')$
proof(*induction rule*: *qTerm-induct*)
  case (*Var xs x*)
  then show *?case* **unfolding** *qVar-alpha-iff* **by** *simp*
next
  case (*Op delta inp binp*)
  show *?case* **proof** *safe*
    fix X'
    assume $qOp \ delta \ inp \ binp \ \#= \ X'$
    then obtain $inp' \ binp'$ **where** $X'eq$: $X' = qOp \ delta \ inp' \ binp'$ **and**
      1: $sameDom \ inp \ inp' \wedge sameDom \ binp \ binp'$ **and**
      2: $liftAll2 \ (\lambda \ Y \ Y'. \ Y \ \#= \ Y') \ inp \ inp' \wedge$
        $liftAll2 \ (\lambda \ A \ A'. \ A \ \$= \ A') \ binp \ binp'$
    unfolding *qOp-alpha-iff* **by** *auto*
    from *Op.IH 1 2*
    show $qSkel \ (qOp \ delta \ inp \ binp) = qSkel \ X'$

by (*simp add*: *X′eq fun-eq-iff option.case-eq-if*
lift-def liftAll-def sameDom-def liftAll2-def)
qed
next
case (*Abs xs x X*)
show *?case*
proof *safe*
fix *A′* **assume** *qAbs xs x X* $= *A′*
then obtain *X′ x′ y* **where** *A′eq*: *A′* = *qAbs xs x′ X′* **and**
∗: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*) **unfolding** *qAbs-alphaAbs-iff* **by** *auto*
moreover have (*X*, *X* #[[*y* ∧ *x*]]-*xs*) ∈ *qSwapped* **using** *qSwap-qSwapped* **by** *fastforce*
ultimately have *qSkel*(*X* #[[*y* ∧ *x*]]-*xs*) = *qSkel*(*X′* #[[*y* ∧ *x′*]]-*xs*)
using *Abs.IH* **by** *blast*
hence *qSkel X* = *qSkel X′* **by**(*auto simp add*: *qSkel-qSwap*)
thus *qSkelAbs* (*qAbs xs x X*) = *qSkelAbs A′* **unfolding** *A′eq* **by** *simp*
qed
qed

corollary *alpha-qSkel*:
fixes *X X′* :: (*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qTerm*
shows *X* #= *X′* ⟹ *qSkel X* = *qSkel X′*
by(*simp add*: *alphaAll-qSkelAll*)

Symmetry of alpha is a property that holds for arbitrary (not necessarily good) quasi-terms.

lemma *alphaAll-sym*:
fixes *X*::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qTerm* **and**
A::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qAbs*
shows
(∀ *X′*. *X* #= *X′* ⟶ *X′* #= *X*) ∧ (∀ *A′*. *A* $= *A′* ⟶ *A′* $= *A*)
proof(*induction rule*: *qTerm-induct*)
case (*Var xs x*)
then show *?case* **unfolding** *qVar-alpha-iff* **by** *simp*
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
fix *X′* **assume** *qOp delta inp binp* #= *X′*
then obtain *inp′ binp′* **where** *X′*: *X′* = *qOp delta inp′ binp′* **and**
1: *sameDom inp inp′* ∧ *sameDom binp binp′*
and *2*: *liftAll2* (λ*Y Y′*. *Y* #= *Y′*) *inp inp′* ∧
liftAll2 (λ*A A′*. *A* $= *A′*) *binp binp′*
unfolding *qOp-alpha-iff* **by** *auto*
thus *X′* #= *qOp delta inp binp*
unfolding *X′* **using** *Op.IH 1 2*
by (*auto simp add*: *fun-eq-iff option.case-eq-if*
lift-def liftAll-def sameDom-def liftAll2-def)
qed

next
case (*Abs xs x X*)
show *?case* **proof** *safe*
fix A' **assume** $qAbs\ xs\ x\ X\ \$=\ A'$
then obtain $x'\ y\ X'$ **where**
1: $A' = qAbs\ xs\ x'\ X' \wedge y \notin \{x,\ x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X'$ **and**
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)$
unfolding *qAbs-alphaAbs-iff* **by** *auto*
moreover have $(X,\ X\ \#[[y \wedge x]]\text{-}xs) \in qSwapped$ **by** (*simp add*: *qSwap-qSwapped*)
ultimately have $(X'\ \#[[y \wedge x']]\text{-}xs)\ \#=\ (X\ \#[[y \wedge x]]\text{-}xs)$ **using** *Abs.IH* **by** *simp*
thus $A'\ \$=\ qAbs\ xs\ x\ X$ **using** *1* **by** *auto*
qed
qed

corollary *alpha-sym*:
fixes $X\ X'$ :: (*'index,'bindex,'varSort,'var,'opSym*)*qTerm*
shows $X\ \#=\ X' \Longrightarrow X'\ \#=\ X$
by(*simp add*: *alphaAll-sym*)

corollary *alphaAbs-sym*:
fixes $A\ A'$ ::(*'index,'bindex,'varSort,'var,'opSym*)*qAbs*
shows $A\ \$=\ A' \Longrightarrow A'\ \$=\ A$
by(*simp add*: *alphaAll-sym*)

Reflexivity does not hold for arbitrary quasi-terms, but onl;y for good ones. Indeed, the proof requires picking a fresh variable, guaranteed to be possible only if the quasi-term is good.

lemma *alphaAll-refl*:
fixes *X*::(*'index,'bindex,'varSort,'var,'opSym*)*qTerm* **and**
A::(*'index,'bindex,'varSort,'var,'opSym*)*qAbs*
shows
$(qGood\ X \longrightarrow X\ \#=\ X) \wedge (qGoodAbs\ A \longrightarrow A\ \$=\ A)$
apply(*rule qGood-qTerm-induct*, *simp-all*)
unfolding *liftAll-def sameDom-def liftAll2-def* **apply** *auto*
proof−
fix *xs x X*
assume *qGood X* **and**
IH: $\bigwedge Y.\ (X,Y) \in qSwapped \Longrightarrow Y\ \#=\ Y$
then obtain *y* **where** *1*: $y \neq x \wedge qAFresh\ xs\ y\ X$
using *obtain-qFresh*[*of* $\{x\}$ $\{X\}$] **by** *auto*
hence $(X,\ X\ \#[[y \wedge x]]\text{-}xs) \in qSwapped$ **using** *qSwap-qSwapped* **by** *auto*
hence $(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X\ \#[[y \wedge x]]\text{-}xs)$ **using** *IH* **by** *simp*
thus $\exists y.\ y \neq x \wedge qAFresh\ xs\ y\ X \wedge (X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X\ \#[[y \wedge x]]\text{-}xs)$
using *1* **by** *blast*
qed

corollary *alpha-refl*:
fixes *X* :: (*'index,'bindex,'varSort,'var,'opSym*)*qTerm*

shows *qGood X* $\Longrightarrow$ *X* #= *X*
by(*simp add*: *alphaAll-refl*)

corollary *alphaAbs-refl*:
fixes *A* ::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
shows *qGoodAbs A* $\Longrightarrow$ *A* \$= *A*
by(*simp add*: *alphaAll-refl*)

lemma *alphaAll-preserves-qGoodAll1*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
shows
(*qGood X* $\longrightarrow$ ($\forall$ *X'*. *X* #= *X'* $\longrightarrow$ *qGood X'*)) $\wedge$
(*qGoodAbs A* $\longrightarrow$ ($\forall$ *A'*. *A* \$= *A'* $\longrightarrow$ *qGoodAbs A'*))
apply(*rule qTerm-induct*, *auto*)
unfolding *qVar-alpha-iff* **apply**(*auto*)
proof−
fix *delta inp binp X'*
assume
IH1: *liftAll* (λ*Y*. *qGood Y* $\longrightarrow$ ($\forall$ *Y'*. *Y* #= *Y'* $\longrightarrow$ *qGood Y'*)) *inp*
and *IH2*: *liftAll* (λ*A*. *qGoodAbs A* $\longrightarrow$ ($\forall$ *A'*. *A* \$= *A'* $\longrightarrow$ *qGoodAbs A'*)) *binp*
and ∗: *liftAll qGood inp liftAll qGoodAbs binp*
and ∗∗: |{*i*. $\exists$ *Y*. *inp i* = *Some Y*}| <*o* |*UNIV* :: '*var set*|
|{*i*. $\exists$ *A*. *binp i* = *Some A*}| <*o* |*UNIV* :: '*var set*|
and *qOp delta inp binp* #= *X'*
then obtain *inp' binp'* **where**
X'eq: *X'* = *qOp delta inp' binp'* **and**
2: *sameDom inp inp'* $\wedge$ *sameDom binp binp'* **and**
3: *liftAll2* (λ*Y Y'*. *Y* #= *Y'*) *inp inp'* $\wedge$
liftAll2 (λ*A A'*. *A* \$= *A'*) *binp binp'*
unfolding *qOp-alpha-iff* **by** *auto*
show *qGood X'*
unfolding *X'eq* **apply** *simp* **unfolding** *liftAll-def* **apply** *auto*
proof−
fix *i Y'* **assume** *inp'*: *inp' i* = *Some Y'*
then obtain *Y* **where** *inp*: *inp i* = *Some Y*
using *2* **unfolding** *sameDom-def* **by** *fastforce*
hence *Y* #= *Y'* **using** *inp' 3* **unfolding** *liftAll2-def* **by** *blast*
moreover have *qGood Y* **using** ∗ *inp* **unfolding** *liftAll-def* **by** *simp*
ultimately show *qGood Y'* **using** *IH1 inp* **unfolding** *liftAll-def* **by** *blast*
next
fix *i A'* **assume** *binp'*: *binp' i* = *Some A'*
then obtain *A* **where** *binp*: *binp i* = *Some A*
using *2* **unfolding** *sameDom-def* **by** *fastforce*
hence *A* \$= *A'* **using** *binp' 3* **unfolding** *liftAll2-def* **by** *blast*
moreover have *qGoodAbs A* **using** ∗ *binp* **unfolding** *liftAll-def* **by** *simp*
ultimately show *qGoodAbs A'* **using** *IH2 binp* **unfolding** *liftAll-def* **by** *blast*
next
have {*i*. $\exists$ *Y'*. *inp' i* = *Some Y'*} = {*i*. $\exists$ *Y*. *inp i* = *Some Y*}

using *2* **unfolding** *sameDom-def* **by** *force*
thus $|\{i.\ \exists\, Y'.\ inp'\ i = Some\ Y'\}|$ $<o$ $|UNIV :: 'var\ set|$ **using** $**$ **by** *simp*
next
have $\{i.\ \exists\, A'.\ binp'\ i = Some\ A'\} = \{i.\ \exists\, A.\ binp\ i = Some\ A\}$
using *2* **unfolding** *sameDom-def* **by** *force*
thus $|\{i.\ \exists\, A'.\ binp'\ i = Some\ A'\}|$ $<o$ $|UNIV :: 'var\ set|$ **using** $**$ **by** *simp*
qed
next
fix *xs x X A'*
assume *IH*: $\bigwedge Y.\ (X,Y) \in qSwapped \Longrightarrow qGood\ Y \longrightarrow (\forall X'.\ Y \#= X' \longrightarrow qGood\ X')$
and $*$: $qGood\ X$ **and** $qAbs\ xs\ x\ X\ \$= A'$
then obtain $x'\ y\ X'$ **where** $A' = qAbs\ xs\ x'\ X'$ **and**
1: $(X\ \#[[y \wedge x]]\text{-}xs) \#= (X'\ \#[[y \wedge x']]\text{-}xs)$
unfolding *qAbs-alphaAbs-iff* **by** *auto*
thus *qGoodAbs A'*
proof(*auto*)
have $(X,\ X\ \#[[y \wedge x]]\text{-}xs) \in qSwapped$ **by**(*auto simp add*: *qSwap-qSwapped*)
moreover have $qGood(X\ \#[[y \wedge x]]\text{-}xs)$ **using** $*$ *qSwap-preserves-qGood* **by** *auto*
ultimately have $qGood(X'\ \#[[y \wedge x']]\text{-}xs)$ **using** *1 IH* **by** *auto*
thus *qGood X'* **using** $*$ *qSwap-preserves-qGood* **by** *auto*
qed
qed

corollary *alpha-preserves-qGood1*:
$[\![X \#= X';\ qGood\ X]\!] \Longrightarrow qGood\ X'$
using *alphaAll-preserves-qGoodAll1* **by** *blast*

corollary *alphaAbs-preserves-qGoodAbs1*:
$[\![A\ \$= A';\ qGoodAbs\ A]\!] \Longrightarrow qGoodAbs\ A'$
using *alphaAll-preserves-qGoodAll1* **by** *blast*

lemma *alpha-preserves-qGood2*:
$[\![X \#= X';\ qGood\ X']\!] \Longrightarrow qGood\ X$
using *alpha-sym alpha-preserves-qGood1* **by** *blast*

lemma *alphaAbs-preserves-qGoodAbs2*:
$[\![A\ \$= A';\ qGoodAbs\ A']\!] \Longrightarrow qGoodAbs\ A$
using *alphaAbs-sym alphaAbs-preserves-qGoodAbs1* **by** *blast*

lemma *alpha-preserves-qGood*:
$X \#= X' \Longrightarrow qGood\ X = qGood\ X'$
using *alpha-preserves-qGood1 alpha-preserves-qGood2* **by** *blast*

lemma *alphaAbs-preserves-qGoodAbs*:
$A\ \$= A' \Longrightarrow qGoodAbs\ A = qGoodAbs\ A'$
using *alphaAbs-preserves-qGoodAbs1 alphaAbs-preserves-qGoodAbs2* **by** *blast*

lemma *alpha-qSwap-preserves-qGood1*:
assumes *ALPHA*: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$ **and**
GOOD: *qGood X*
shows *qGood X'*
proof −
have *qGood*$(X \ \#[[y \wedge x]]\text{-}zs)$ **using** *GOOD qSwap-preserves-qGood* **by** *auto*
hence *qGood* $(X' \ \#[[y' \wedge x']]\text{-}zs')$ **using** *ALPHA alpha-preserves-qGood* **by** *auto*
thus *qGood X'* **using** *qSwap-preserves-qGood* **by** *auto*
qed

lemma *alpha-qSwap-preserves-qGood2*:
assumes *ALPHA*: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$ **and**
GOOD': *qGood X'*
shows *qGood X*
proof −
have *qGood*$(X' \ \#[[y' \wedge x']]\text{-}zs')$ **using** *GOOD' qSwap-preserves-qGood* **by** *auto*
hence *qGood* $(X \ \#[[y \wedge x]]\text{-}zs)$ **using** *ALPHA alpha-preserves-qGood* **by** *auto*
thus *qGood X* **using** *qSwap-preserves-qGood* **by** *auto*
qed

lemma *alphaAbs-qSwapAbs-preserves-qGoodAbs2*:
assumes *ALPHA*: $(A \ \$[[y \wedge x]]\text{-}zs) \ \$= (A' \ \$[[y' \wedge x']]\text{-}zs')$ **and**
GOOD': *qGoodAbs A'*
shows *qGoodAbs A*
proof −
have *qGoodAbs*$(A' \ \$[[y' \wedge x']]\text{-}zs')$ **using** *GOOD' qSwapAbs-preserves-qGoodAbs*
by *auto*
hence *qGoodAbs* $(A \ \$[[y \wedge x]]\text{-}zs)$ **using** *ALPHA alphaAbs-preserves-qGoodAbs*
by *auto*
thus *qGoodAbs A* **using** *qSwapAbs-preserves-qGoodAbs* **by** *auto*
qed

lemma *alpha-qSwap-preserves-qGood*:
assumes *ALPHA*: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$
shows *qGood X* = *qGood X'*
using *assms alpha-qSwap-preserves-qGood1*
alpha-qSwap-preserves-qGood2 **by** *auto*

lemma *qSwapAll-preserves-alphaAll*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and** *z1 z2 zs*
shows
$(qGood \ X \longrightarrow (\forall \ X' \ zs \ z1 \ z2. \ X \ \#= X' \longrightarrow$
$(X \ \#[[z1 \wedge z2]]\text{-}zs) \ \#= (X' \ \#[[z1 \wedge z2]]\text{-}zs))) \ \wedge$
$(qGoodAbs \ A \longrightarrow (\forall \ A' \ zs \ z1 \ z2. \ A \ \$= A' \longrightarrow$
$(A \ \$[[z1 \wedge z2]]\text{-}zs) \ \$= (A' \ \$[[z1 \wedge z2]]\text{-}zs)))$
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
then show *?case* **unfolding** *qVar-alpha-iff* **by** *simp*

next
  case (*Op delta inp binp*)
  show *?case* **proof** *safe*
    fix X' *zs z1 z2*
    assume *qOp delta inp binp* #= X' **term** X' **term** *binp*
    then obtain *inp' binp'* **where** *X'eq*: X' = *qOp delta inp' binp'* **and**
    1: *sameDom inp inp'* ∧ *sameDom binp binp'*
    and *2*: *liftAll2* (λ *Y Y'*. *Y* #= *Y'*) *inp inp'* ∧
        liftAll2 (λ *A A'*. *A* $= *A'*) *binp binp'*
    unfolding *qOp-alpha-iff* **by** *auto*
    thus ((*qOp delta inp binp*) #[[*z1* ∧ *z2*]]-*zs*) #= (*X'* #[[*z1* ∧ *z2*]]-*zs*)
    unfolding *X'eq* **using** *Op.IH*
    by (*auto simp add*: *fun-eq-iff option.case-eq-if*
      lift-def liftAll-def sameDom-def liftAll2-def)
  qed
next
  case (*Abs xs x X*)
  show *?case* **proof** *safe*
    fix *A' zs z1 z2* **assume** *qAbs xs x X* $= *A'*
    then obtain *x' y X'* **where** *A'*: *A'* = *qAbs xs x' X'* **and**
    y-not: *y* ∉ {*x*, *x'*} **and** *y-fresh*: *qAFresh xs y X* ∧ *qAFresh xs y X'* **and**
    alpha: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)
    unfolding *qAbs-alphaAbs-iff* **by** *auto*
     hence *goodX'*: *qGood X'* **using** ‹*qGood X*› *alpha-qSwap-preserves-qGood* **by** *fastforce*

    obtain *u* **where** *u-notin*: *u* ∉ {*x*,*x'*,*z1*,*z2*,*y*} **and**
            u-freshXX': *qAFresh xs u X* ∧ *qAFresh xs u X'*
    using ‹*qGood X*› *goodX' obtain-qFresh*[*of* {*x*,*x'*,*z1*,*z2*,*y*} {*X*,*X'*}] **by** *auto*
    hence *u-not*: *u* ≠ (*x* @*xs*[*z1* ∧ *z2*]-*zs*) ∧ *u* ≠ (*x'* @*xs*[*z1* ∧ *z2*]-*zs*)
    unfolding *sw-def* **using** *u-notin* **by** *auto*
     have *u-fresh*: *qAFresh xs u* (*X* #[[*z1* ∧ *z2*]]-*zs*) ∧ *qAFresh xs u* (*X'* #[[*z1* ∧ *z2*]]-*zs*)
   using *u-freshXX' u-notin* **by**(*auto simp add*: *qSwap-preserves-qAFresh-distinct*)

    have ((*X* #[[*z1* ∧ *z2*]]-*zs*) #[[*u* ∧ (*x* @*xs*[*z1* ∧ *z2*]-*zs*)]]-*xs*) =
       (((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #[[*z1* ∧ *z2*]]-*zs*)
    using *y-fresh u-freshXX' u-notin* **by** (*simp add*: *qSwap-3commute*)
    moreover
    {**have** *1*: (*X*, *X* #[[*y* ∧ *x*]]-*xs*) ∈ *qSwapped* **by**(*simp add*: *qSwap-qSwapped*)
    hence ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #= ((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*u* ∧ *y*]]-*xs*)
     using *alpha Abs.IH* **by** *auto*
     moreover have (*X*, (*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) ∈ *qSwapped*
     using *1* **by**(*auto simp add*: *qSwapped.Swap*)
     ultimately have (((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #[[*z1* ∧ *z2*]]-*zs*) #=
           (((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #[[*z1* ∧ *z2*]]-*zs*)
     using *Abs.IH* **by** *auto*
    }
    moreover

have $(((X' \#[[y \wedge x']]\text{-}xs) \#[[u \wedge y]]\text{-}xs) \#[[z1 \wedge z2]]\text{-}zs) =$
$((X' \#[[z1 \wedge z2]]\text{-}zs) \#[[u \wedge (x' @xs[z1 \wedge z2]\text{-}zs)]]\text{-}xs)$
using *y-fresh u-freshXX' u-notin* **by** (*auto simp add*: *qSwap-3commute*)
ultimately have $((X \#[[z1 \wedge z2]]\text{-}zs) \#[[u \wedge (x @xs[z1 \wedge z2]\text{-}zs)]]\text{-}xs) \#=$
$((X' \#[[z1 \wedge z2]]\text{-}zs) \#[[u \wedge (x' @xs[z1 \wedge z2]\text{-}zs)]]\text{-}xs)$ **by** *simp*
thus $((qAbs\ xs\ x\ X)\ \$[[z1 \wedge z2]]\text{-}zs)\ \$= (A'\ \$[[z1 \wedge z2]]\text{-}zs)$
unfolding A' **using** *u-not u-fresh* **by** *auto*
qed
qed

corollary *qSwap-preserves-alpha*:
assumes $qGood\ X \vee qGood\ X'$ **and** $X \#= X'$
shows $(X \#[[z1 \wedge z2]]\text{-}zs) \#= (X' \#[[z1 \wedge z2]]\text{-}zs)$
using *assms alpha-preserves-qGood qSwapAll-preserves-alphaAll* **by** *blast*

corollary *qSwapAbs-preserves-alphaAbs*:
assumes $qGoodAbs\ A \vee qGoodAbs\ A'$ **and** $A\ \$= A'$
shows $(A\ \$[[z1 \wedge z2]]\text{-}zs)\ \$= (A'\ \$[[z1 \wedge z2]]\text{-}zs)$
using *assms alphaAbs-preserves-qGoodAbs qSwapAll-preserves-alphaAll* **by** *blast*

lemma *qSwap-twice-preserves-alpha*:
assumes $qGood\ X \vee qGood\ X'$ **and** $X \#= X'$
shows $((X \#[[z1 \wedge z2]]\text{-}zs) \#[[u1 \wedge u2]]\text{-}us) \#= ((X' \#[[z1 \wedge z2]]\text{-}zs) \#[[u1 \wedge u2]]\text{-}us)$
by (*simp add*: *assms qSwap-preserves-alpha qSwap-preserves-qGood*)

lemma *alphaAll-trans*:
fixes X::(*'index,'bindex,'varSort,'var,'opSym*)*qTerm* **and**
A::(*'index,'bindex,'varSort,'var,'opSym*)*qAbs*
shows
$(qGood\ X \longrightarrow (\forall\ X'\ X''.\ X \#= X' \wedge X' \#= X'' \longrightarrow X \#= X'')) \wedge$
$(qGoodAbs\ A \longrightarrow (\forall\ A'\ A''.\ A\ \$= A' \wedge A'\ \$= A'' \longrightarrow A\ \$= A''))$
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
then show *?case* **by** (*simp add*: *qVar-alpha-iff*)
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
fix $X'\ X''$ **assume** $qOp\ delta\ inp\ binp \#= X'$ **and** $*$: $X' \#= X''$
then obtain $inp'\ binp'$ **where**
1: $X' = qOp\ delta\ inp'\ binp'$ **and**
2: $sameDom\ inp\ inp' \wedge sameDom\ binp\ binp'$ **and**
3: $liftAll2\ (\lambda Y\ Y'.\ Y \#= Y')\ inp\ inp' \wedge$
$liftAll2\ (\lambda A\ A'.\ A\ \$= A')\ binp\ binp'$
unfolding *qOp-alpha-iff* **by** *auto*
obtain $inp''\ binp''$ **where**
11: $X'' = qOp\ delta\ inp''\ binp''$ **and**
22: $sameDom\ inp'\ inp'' \wedge sameDom\ binp'\ binp''$ **and**
33: $liftAll2\ (\lambda Y'\ Y''.\ Y' \#= Y'')\ inp'\ inp'' \wedge$

liftAll2 ($\lambda A'$ A''. A' \$= A'') *binp' binp''*
using ∗ **unfolding** *1* **unfolding** *qOp-alpha-iff* **by** *auto*
have *liftAll2* (#=) *inp inp''* **unfolding** *liftAll2-def* **proof** *safe*
fix *i Y Y''*
assume *inp*: *inp i* = *Some Y* **and** *inp''*: *inp'' i* = *Some Y''*
then obtain *Y'* **where** *inp'*: *inp' i* = *Some Y'*
using *2* **unfolding** *sameDom-def* **by** *force*
hence *Y* #= *Y'* **using** *inp 3* **unfolding** *liftAll2-def* **by** *blast*
moreover have *Y'* #= *Y''* **using** *inp' inp'' 33* **unfolding** *liftAll2-def* **by** *blast*
ultimately show *Y* #= *Y''* **using** *inp Op.IH* **unfolding** *liftAll-def* **by** *blast*
qed
moreover have *liftAll2* (\$=) *binp binp''* **unfolding** *liftAll2-def* **proof** *safe*
fix *i A A''*
assume *binp*: *binp i* = *Some A* **and** *binp''*: *binp'' i* = *Some A''*
then obtain *A'* **where** *binp'*: *binp' i* = *Some A'*
using *2* **unfolding** *sameDom-def* **by** *force*
hence *A* \$= *A'* **using** *binp 3* **unfolding** *liftAll2-def* **by** *blast*
moreover have *A'* \$= *A''* **using** *binp' binp'' 33* **unfolding** *liftAll2-def* **by** *blast*
ultimately show *A* \$= *A''* **using** *binp Op.IH* **unfolding** *liftAll-def* **by** *blast*
qed
ultimately show *qOp delta inp binp* #= *X''*
by (*simp add*: *11 2 22 sameDom-trans*[*of inp inp'*] *sameDom-trans*[*of binp binp'*])
qed
next
case (*Abs xs x X*)
show *?case* **proof** *safe*
fix *A' A''*
assume *qAbs xs x X* \$= *A'* **and** ∗: *A'* \$= *A''*
then obtain *x' y X'* **where** *A'*: *A'* = *qAbs xs x' X'* **and** *y-not*: $y \notin \{x, x'\}$ **and**
y-fresh: *qAFresh xs y X* ∧ *qAFresh xs y X'* **and**
alpha: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)
unfolding *qAbs-alphaAbs-iff* **by** *auto*
obtain *x'' z X''* **where** *A''*: *A''* = *qAbs xs x'' X''* **and** *z-not*: $z \notin \{x', x''\}$ **and**
z-fresh: *qAFresh xs z X'* ∧ *qAFresh xs z X''* **and**
alpha': (*X'* #[[*z* ∧ *x'*]]-*xs*) #= (*X''* #[[*z* ∧ *x''*]]-*xs*)
using ∗ **unfolding** *A' qAbs-alphaAbs-iff* **by** *auto*
have *goodX'*: *qGood X'*
using *alpha* ‹*qGood X*› *alpha-qSwap-preserves-qGood* **by** *fastforce*
hence *goodX''*: *qGood X''*
using *alpha' alpha-qSwap-preserves-qGood* **by** *fastforce*
have *good*: *qGood*((*X* #[[*y* ∧ *x*]]-*xs*)) ∧ *qGood*((*X'* #[[*z* ∧ *x'*]]-*xs*))
using ‹*qGood X*› *goodX' qSwap-preserves-qGood* **by** *auto*

obtain *u* **where** *u-not*: $u \notin \{x,x',x'',y,z\}$ **and**
u-fresh: *qAFresh xs u X* ∧ *qAFresh xs u X'* ∧ *qAFresh xs u X''*

using ‹*qGood X*› *goodX′ goodX″*
using *obtain-qFresh*[*of* {*x*,*x′*,*x″*,*y*,*z*} {*X*, *X′*, *X″*}] **by** *auto*

{**have** (*X* #[[*u* ∧ *x*]]-*xs*) = ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*)
using *u-fresh y-fresh* **by** (*auto simp add*: *qAFresh-qSwap-compose*)
moreover
have ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #= ((*X′* #[[*y* ∧ *x′*]]-*xs*) #[[*u* ∧ *y*]]-*xs*)
using *good alpha qSwap-preserves-alpha* **by** *fastforce*
moreover have ((*X′* #[[*y* ∧ *x′*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) = (*X′* #[[*u* ∧ *x′*]]-*xs*)
using *u-fresh y-fresh* **by** (*auto simp add*: *qAFresh-qSwap-compose*)
ultimately have (*X* #[[*u* ∧ *x*]]-*xs*) #= (*X′* #[[*u* ∧ *x′*]]-*xs*) **by** *simp*
}
moreover
{**have** (*X′* #[[*u* ∧ *x′*]]-*xs*) = ((*X′* #[[*z* ∧ *x′*]]-*xs*) #[[*u* ∧ *z*]]-*xs*)
using *u-fresh z-fresh* **by** (*auto simp add*: *qAFresh-qSwap-compose*)
moreover
have ((*X′* #[[*z* ∧ *x′*]]-*xs*) #[[*u* ∧ *z*]]-*xs*) #= ((*X″* #[[*z* ∧ *x″*]]-*xs*) #[[*u* ∧ *z*]]-*xs*)
using *good alpha′ qSwap-preserves-alpha* **by** *fastforce*
moreover have ((*X″* #[[*z* ∧ *x″*]]-*xs*) #[[*u* ∧ *z*]]-*xs*) = (*X″* #[[*u* ∧ *x″*]]-*xs*)
using *u-fresh z-fresh* **by** (*auto simp add*: *qAFresh-qSwap-compose*)
ultimately have (*X′* #[[*u* ∧ *x′*]]-*xs*) #= (*X″* #[[*u* ∧ *x″*]]-*xs*) **by** *simp*
}
moreover have (*X*, *X* #[[*u* ∧ *x*]]-*xs*) ∈ *qSwapped* **by** (*simp add*: *qSwap-qSwapped*)
ultimately have (*X* #[[*u* ∧ *x*]]-*xs*) #= (*X″* #[[*u* ∧ *x″*]]-*xs*)
using *Abs.IH* **by** *blast*
thus *qAbs xs x X* $= *A″*
unfolding *A″* **using** *u-not u-fresh* **by** *auto*
qed
qed

corollary *alpha-trans*:
assumes *qGood X* ∨ *qGood X′* ∨ *qGood X″ X* #= *X′ X′* #= *X″*
shows *X* #= *X″*
by (*meson alphaAll-trans alpha-preserves-qGood assms*)

corollary *alphaAbs-trans*:
assumes *qGoodAbs A* ∨ *qGoodAbs A′* ∨ *qGoodAbs A″*
and *A* $= *A′ A′* $= *A″*
shows *A* $= *A″*
using *assms alphaAbs-preserves-qGoodAbs alphaAll-trans* **by** *blast*

lemma *alpha-trans-twice*:
⟦*qGood X* ∨ *qGood X′* ∨ *qGood X″* ∨ *qGood X‴*;
X #= *X′*; *X′* #= *X″*; *X″* #= *X‴*⟧ ⟹ *X* #= *X‴*
using *alpha-trans* **by** *blast*

lemma *alphaAbs-trans-twice*:
⟦*qGoodAbs A* ∨ *qGoodAbs A′* ∨ *qGoodAbs A″* ∨ *qGoodAbs A‴*;

$A \mathbin{\$}= A'$; $A' \mathbin{\$}= A''$; $A'' \mathbin{\$}= A'''$⟧ ⟹ $A \mathbin{\$}= A'''$
using *alphaAbs-trans* **by** *blast*

lemma *qAbs-preserves-alpha*:
assumes *ALPHA*: X #= X' **and** *GOOD*: *qGood* X ∨ *qGood* X'
shows *qAbs xs x X* \$= *qAbs xs x X'*
proof−
have *qGood X* ∧ *qGood X'* **using** *GOOD ALPHA* **by**(*auto simp add*: *alpha-preserves-qGood*)
then obtain *y* **where** *y-not*: $y \neq x$ **and**
y-fresh: *qAFresh xs y X* ∧ *qAFresh xs y X'*
using *GOOD obtain-qFresh*[*of* {*x*} {*X*,*X'*}] **by** *auto*
hence (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x*]]-*xs*)
using *ALPHA GOOD* **by**(*simp add*: *qSwap-preserves-alpha*)
thus *?thesis* **using** *y-not y-fresh* **by** *auto*
qed

corollary *qAbs-preserves-alpha2*:
assumes *ALPHA*: *X* #= *X'* **and** *GOOD*: *qGoodAbs*(*qAbs xs x X*) ∨ *qGoodAbs* (*qAbs xs x X'*)
shows *qAbs xs x X* \$= *qAbs xs x X'*
using *assms* **by** (*intro qAbs-preserves-alpha*) *auto*

2.4.4 Picking fresh representatives

lemma *qAbs-alphaAbs-qSwap-qAFresh*:
assumes *GOOD*: *qGood X* **and** *FRESH*: *qAFresh ys x' X*
shows *qAbs ys x X* \$= *qAbs ys x'* (*X* #[[*x'* ∧ *x*]]-*ys*)
proof−
obtain *y* **where** *1*: $y \notin \{x,x'\}$ **and** *2*: *qAFresh ys y X*
using *GOOD obtain-qFresh*[*of* {*x*,*x'*} {*X*}] **by** *auto*
hence *3*: *qAFresh ys y* (*X* #[[*x'* ∧ *x*]]-*ys*)
by (*auto simp add*: *qSwap-preserves-qAFresh-distinct*)

have (*X* #[[*y* ∧ *x*]]-*ys*) = ((*X* #[[*x'* ∧ *x*]]-*ys*) #[[*y* ∧ *x'*]]-*ys*)
using *FRESH 2* **by** (*auto simp add*: *qAFresh-qSwap-compose*)
moreover have *qGood* (*X* #[[*y* ∧ *x*]]-*ys*)
using *1 GOOD qSwap-preserves-qGood* **by** *auto*
ultimately have (*X* #[[*y* ∧ *x*]]-*ys*) #= ((*X* #[[*x'* ∧ *x*]]-*ys*) #[[*y* ∧ *x'*]]-*ys*)
using *alpha-refl* **by** *simp*

thus *?thesis* **using** *1 2 3 assms* **by** *auto*
qed

lemma *qAbs-ex-qAFresh-rep*:
assumes *GOOD*: *qGood X* **and** *FRESH*: *qAFresh xs x' X*
shows ∃ *X'*. *qGood X'* ∧ *qAbs xs x X* \$= *qAbs xs x' X'*
proof−
have *1*: *qGood* (*X* #[[*x'* ∧ *x*]]-*xs*) **using** *assms qSwap-preserves-qGood* **by** *auto*
show *?thesis*

apply(*rule exI*[*of* - X #[[x' ∧ x]]-*xs*])
using *assms 1 qAbs-alphaAbs-qSwap-qAFresh* **by** *fastforce*
qed

2.5 Properties of swapping and freshness modulo alpha

lemma *qFreshAll-imp-ex-qAFreshAll*:
fixes X::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and** *zs fZs*
assumes *FIN*: *finite V*
shows
(*qGood X* ⟶
((∀ z ∈ V. ∀ zs ∈ fZs z. qFresh zs z X) ⟶
(∃ X'. X #= X' ∧ (∀ z ∈ V. ∀ zs ∈ fZs z. qAFresh zs z X')))) ∧
(*qGoodAbs A* ⟶
((∀ z ∈ V. ∀ zs ∈ fZs z. qFreshAbs zs z A) ⟶
(∃ A'. A $= A' ∧ (∀ z ∈ V. ∀ zs ∈ fZs z. qAFreshAbs zs z A'))))
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
show *?case*
by (*metis alpha-qVar-iff qAFreshAll-simps*(*1*) *qFreshAll-simps*(*1*))
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
assume ∗: ∀ z∈V. ∀ zs∈fZs z. qFresh zs z (qOp delta inp binp)
define *phi* **and** *phiAbs* **where**
phi ≡ (λ(Y::('index,'bindex,'varSort,'var,'opSym)qTerm) Y'.
Y #= Y' ∧ (∀ z∈V. ∀ zs∈fZs z. qAFresh zs z Y')) **and**
phiAbs ≡ (λ(A::('index,'bindex,'varSort,'var,'opSym)qAbs) A'.
A $= A' ∧ (∀ z∈V. ∀ zs∈fZs z. qAFreshAbs zs z A'))
have *ex-phi*: ⋀ i Y. inp i = Some Y ⟹ ∃ Y'. phi Y Y'
unfolding *phi-def* **using** *Op.IH* ∗ **by** (*auto simp add*: *liftAll-def*)
have *ex-phiAbs*: ⋀ i A. binp i = Some A ⟹ ∃ A'. phiAbs A A'
unfolding *phiAbs-def* **using** *Op.IH* ∗ **by** (*auto simp add*: *liftAll-def*)
define *inp'* **and** *binp'* **where**
inp' ≡ λ i. case inp i of Some Y ⇒ Some (SOME Y'. phi Y Y') |None ⇒
None **and**
binp' ≡ λ i. case binp i of Some A ⇒ Some (SOME A'. phiAbs A A') |None ⇒
None
show ∃ X'. qOp delta inp binp #= X' ∧ (∀ z∈V. ∀ zs∈fZs z. qAFresh zs z X')
by (*intro exI*[*of* - *qOp delta inp' binp'*])
(*auto simp add*: *inp'-def binp'-def option.case-eq-if sameDom-def liftAll-def*
liftAll2-def,
(*meson ex-phi phi-def ex-phiAbs phiAbs-def some-eq-ex*)+)
qed
next
case (*Abs xs x X*)
show *?case* **proof** *safe*
assume ∗: ∀ z∈V. ∀ zs∈fZs z. qFreshAbs zs z (qAbs xs x X)

obtain *y* **where** *y-not-x*: $y \neq x$ **and** *y-not-V*: $y \notin V$
and *y-afresh*: *qAFresh xs y X*
using *FIN* ‹*qGood X*› *obtain-qFresh*[*of* $V \cup \{x\}$ $\{X\}$] **by** *auto*
hence *y-fresh*: *qFresh xs y X* **using** *qAFresh-imp-qFresh* **by** *fastforce*
obtain *Y* **where** *Y-def*: $Y = (X \ \#[[y \wedge x\ ]]\text{-}xs)$ **by** *blast*
have *alphaXY*: *qAbs xs x X* \$= *qAbs xs y Y*
using ‹*qGood X*› *y-afresh qAbs-alphaAbs-qSwap-qAFresh* **unfolding** *Y-def* **by** *fastforce*
have $\forall z \in V.\ \forall zs \in fZs\ z.$ *qFresh zs z Y*
unfolding *Y-def*
by (*metis* ∗ *not-equals-and-not-equals-not-in qAFresh-imp-qFresh qAFresh-qSwap-exchange1*

qFreshAbs.simps qSwap-preserves-qFresh-distinct y-afresh y-not-V)
moreover have $(X, Y) \in$ *qSwapped* **unfolding** *Y-def* **by**(*simp add*: *qSwap-qSwapped*)
ultimately obtain Y' **where** Y #= Y' **and** ∗∗: $\forall z \in V.\ \forall zs \in fZs\ z.$ *qAFresh zs z* Y'
using *Abs.IH* **by** *blast*
moreover have *qGood Y* **unfolding** *Y-def* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
ultimately have *qAbs xs y Y* \$= *qAbs xs y* Y' **using** *qAbs-preserves-alpha* **by** *blast*
moreover have *qGoodAbs*(*qAbs xs x X*) **using** ‹*qGood X*› **by** *simp*
ultimately have *qAbs xs x X* \$= *qAbs xs y* Y' **using** *alphaXY alphaAbs-trans* **by** *blast*
moreover have $\forall z \in V.\ \forall zs \in fZs\ z.$ *qAFreshAbs zs z* (*qAbs xs y* Y') **using** ∗∗ *y-not-V* **by** *auto*
ultimately show $\exists A'.$ *qAbs xs x X* \$= $A' \wedge (\forall z \in V.\ \forall zs \in fZs\ z.$ *qAFreshAbs zs z* A')
by *blast*
qed
qed

corollary *qFresh-imp-ex-qAFresh*:
assumes *finite V* **and** *qGood X* **and** $\forall\ z \in V.\ \forall zs \in fZs\ z.$ *qFresh zs z X*
shows $\exists\ X'.$ *qGood* $X' \wedge X$ #= $X' \wedge (\forall\ z \in V.\ \forall zs \in fZs\ z.$ *qAFresh zs z* X')
by (*metis alphaAll-preserves-qGoodAll1 assms qFreshAll-imp-ex-qAFreshAll*)

corollary *qFreshAbs-imp-ex-qAFreshAbs*:
assumes *finite V* **and** *qGoodAbs A* **and** $\forall\ z \in V.\ \forall zs \in fZs\ z.$ *qFreshAbs zs z A*
shows $\exists\ A'.$ *qGoodAbs* $A' \wedge A$ \$= $A' \wedge (\forall\ z \in V.\ \forall zs \in fZs\ z.$ *qAFreshAbs zs z* A')
by (*metis alphaAll-preserves-qGoodAll1 assms qFreshAll-imp-ex-qAFreshAll*)

lemma *qFresh-imp-ex-qAFresh1*:
assumes *qGood X* **and** *qFresh zs z X*
shows $\exists\ X'.$ *qGood* $X' \wedge X$ #= $X' \wedge$ *qAFresh zs z* X'
using *assms qFresh-imp-ex-qAFresh*[*of* $\{z\}$ - *undefined*$(z := \{zs\})$] **by** *fastforce*

lemma *qFreshAbs-imp-ex-qAFreshAbs1*:

assumes *finite V* **and** *qGoodAbs A* **and** *qFreshAbs zs z A*
shows ∃ *A'*. *qGoodAbs A'* ∧ *A* $= *A'* ∧ *qAFreshAbs zs z A'*
using *assms qFreshAbs-imp-ex-qAFreshAbs*[*of* {*z*} - *undefined*(*z* := {*zs*})] **by** *fast-force*

lemma *qFresh-imp-ex-qAFresh2*:
assumes *qGood X* **and** *qFresh xs x X* **and** *qFresh ys y X*
shows ∃ *X'*. *qGood X'* ∧ *X* #= *X'* ∧ *qAFresh xs x X'* ∧ *qAFresh ys y X'*
using *assms*
qFresh-imp-ex-qAFresh[*of* {*x*} - *undefined*(*x* := {*xs*,*ys*})]
qFresh-imp-ex-qAFresh[*of* {*x*,*y*} - (*undefined*(*x* := {*xs*}))(*y* := {*ys*})]
by (*cases x* = *y*) *auto*

lemma *qFreshAbs-imp-ex-qAFreshAbs2*:
assumes *finite V* **and** *qGoodAbs A* **and** *qFreshAbs xs x A* **and** *qFreshAbs ys y A*
shows ∃ *A'*. *qGoodAbs A'* ∧ *A* $= *A'* ∧ *qAFreshAbs xs x A'* ∧ *qAFreshAbs ys y A'*
using *assms*
qFreshAbs-imp-ex-qAFreshAbs[*of* {*x*} - *undefined*(*x* := {*xs*,*ys*})]
qFreshAbs-imp-ex-qAFreshAbs[*of* {*x*,*y*} - (*undefined*(*x* := {*xs*}))(*y* := {*ys*})]
by (*cases x* = *y*) *auto*

lemma *qAFreshAll-qFreshAll-preserves-alphaAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
    A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *zs z*
shows
(*qGood X* ⟶
  (*qAFresh zs z X* ⟶ (∀ *X'*. *X* #= *X'* ⟶ *qFresh zs z X'*))) ∧
 (*qGoodAbs A* ⟶
  (*qAFreshAbs zs z A* ⟶ (∀ *A'*. *A* $= *A'* ⟶ *qFreshAbs zs z A'*)))
proof(*induction rule*: *qGood-qTerm-induct*)
  case (*Var xs x*)
  thus *?case* **unfolding** *qVar-alpha-iff* **by** *simp*
next
  case (*Op delta inp binp*)
  show *?case* **proof** *safe*
    fix *X'*
    assume *afresh*: *qAFresh zs z* (*qOp delta inp binp*)
    and *qOp delta inp binp* #= *X'*
    then obtain *inp'* **and** *binp'* **where** *X'eq*: *X'* = *qOp delta inp' binp'* **and**
    ∗: (∀ *i*. (*inp i* = *None*) = (*inp' i* = *None*)) ∧
      (∀ *i*. (*binp i* = *None*) = (*binp' i* = *None*)) **and**
    ∗∗: (∀ *i Y Y'*. *inp i* = *Some Y* ∧ *inp' i* = *Some Y'* ⟶ *Y* #= *Y'*) ∧
      (∀ *i A A'*. *binp i* = *Some A* ∧ *binp' i* = *Some A'* ⟶ *A* $= *A'*)
    unfolding *qOp-alpha-iff sameDom-def liftAll2-def* **by** *auto*
    {**fix** *i Y'* **assume** *inp'*: *inp' i* = *Some Y'*
     then obtain *Y* **where** *inp*: *inp i* = *Some Y* **using** ∗ **by** *fastforce*
     hence *Y* #= *Y'* **using** *inp'* ∗∗ **by** *blast*
     hence *qFresh zs z Y'* **using** *inp Op.IH afresh* **by** (*auto simp*: *liftAll-def*)
    }

moreover
{**fix** *i A'* **assume** *binp'*: *binp' i = Some A'*
then obtain *A* **where** *binp*: *binp i = Some A* **using** ∗ **by** *fastforce*
hence *A* $= *A'* **using** *binp'* ∗∗ **by** *blast*
hence *qFreshAbs zs z A'* **using** *binp Op.IH afresh* **by** (*auto simp*: *liftAll-def*)
}
ultimately show *qFresh zs z X'*
unfolding *X'eq* **apply** *simp* **unfolding** *liftAll-def* **by** *simp*
qed
next
case (*Abs xs x X*)
show *?case* **proof** *safe*
fix *A'*
assume *qAbs xs x X* $= *A'* **and** *afresh*: *qAFreshAbs zs z* (*qAbs xs x X*)
then obtain *x' y X'* **where** *A'eq*: *A' = qAbs xs x' X'* **and**
ynot: *y* ∉ {*x*, *x'*} **and** *y-afresh*: *qAFresh xs y X* ∧ *qAFresh xs y X'* **and**
alpha: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)
unfolding *qAbs-alphaAbs-iff* **by** *auto*

have *goodXxy*: *qGood*(*X* #[[*y* ∧ *x*]]-*xs*) **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodX'yx'*: *qGood*(*X'* #[[*y* ∧ *x'*]]-*xs*) **using** *alpha alpha-preserves-qGood* **by** *auto*
hence *qGood X'* **using** *qSwap-preserves-qGood* **by** *auto*
then obtain *u* **where** *u-afresh*: *qAFresh xs u X* ∧ *qAFresh xs u X'*
and *unot*: *u* ∉ {*x*,*x'*,*z*} **using** ‹*qGood X*› *obtain-qFresh*[*of* {*x*,*x'*,*z*} {*X*,*X'*}] **by** *auto*

have (*X* #[[*u* ∧ *x*]]-*xs*) = ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) ∧
(*X'* #[[*u* ∧ *x'*]]-*xs*) = ((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*u* ∧ *y*]]-*xs*)
using *u-afresh y-afresh qAFresh-qSwap-compose* **by** *fastforce*
moreover have ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*u* ∧ *y*]]-*xs*) #= ((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*u* ∧ *y*]]-*xs*)
using *goodXxy goodX'yx' alpha qSwap-preserves-alpha* **by** *fastforce*
ultimately have *alpha*: (*X* #[[*u* ∧ *x*]]-*xs*) #= (*X'* #[[*u* ∧ *x'*]]-*xs*) **by** *simp*

moreover have (*X*, *X* #[[*u* ∧ *x*]]-*xs*) ∈ *qSwapped* **by** (*simp add*: *qSwap-qSwapped*)
moreover have *qAFresh zs z* (*X* #[[*u* ∧ *x*]]-*xs*)
using *unot afresh* **by**(*auto simp add*: *qSwap-preserves-qAFresh-distinct*)
ultimately have *qFresh zs z* (*X'* #[[*u* ∧ *x'*]]-*xs*) **using** *afresh Abs.IH* **by** *simp*
hence *zs = xs* ∧ *z = x'* ∨ *qFresh zs z X'*
using *unot afresh qSwap-preserves-qFresh-distinct*[*of zs xs z*] **by** *fastforce*
thus *qFreshAbs zs z A'* **unfolding** *A'eq* **by** *simp*
qed
qed

corollary *qAFresh-qFresh-preserves-alpha*:
⟦*qGood X*; *qAFresh zs z X*; *X* #= *X'*⟧ ⟹ *qFresh zs z X'*
by(*simp add*: *qAFreshAll-qFreshAll-preserves-alphaAll*)

corollary *qAFreshAbs-imp-qFreshAbs-preserves-alphaAbs*:
⟦*qGoodAbs A*; *qAFreshAbs zs z A*; *A* $= *A'*⟧ ⟹ *qFreshAbs zs z A'*
by(*simp add*: *qAFreshAll-qFreshAll-preserves-alphaAll*)

lemma *qFresh-preserves-alpha1*:
assumes *qGood X* **and** *qFresh zs z X* **and** *X* #= *X'*
shows *qFresh zs z X'*
by (*meson alpha-sym alpha-trans assms qAFresh-qFresh-preserves-alpha qFresh-imp-ex-qAFresh1*)

lemma *qFreshAbs-preserves-alphaAbs1*:
assumes *qGoodAbs A* **and** *qFreshAbs zs z A* **and** *A* $= *A'*
shows *qFreshAbs zs z A'*
by (*meson alphaAbs-sym alphaAbs-trans assms finite.emptyI*
qAFreshAbs-imp-qFreshAbs-preserves-alphaAbs qFreshAbs-imp-ex-qAFreshAbs1)

lemma *qFresh-preserves-alpha*:
assumes *qGood X* ∨ *qGood X'* **and** *X* #= *X'*
shows *qFresh zs z X* ⟷ *qFresh zs z X'*
using *alpha-preserves-qGood alpha-sym assms qFresh-preserves-alpha1* **by** *blast*

lemma *qFreshAbs-preserves-alphaAbs*:
assumes *qGoodAbs A* ∨ *qGoodAbs A'* **and** *A* $= *A'*
shows *qFreshAbs zs z A* = *qFreshAbs zs z A'*
using *assms alphaAbs-preserves-qGoodAbs alphaAbs-sym qFreshAbs-preserves-alphaAbs1*
by *blast*

lemma *alpha-qFresh-qSwap-id*:
assumes *qGood X* **and** *qFresh zs z1 X* **and** *qFresh zs z2 X*
shows (*X* #[[*z1* ∧ *z2*]]-*zs*) #= *X*
proof−
  obtain *X'* **where** *1*: *X* #= *X'* **and** *qAFresh zs z1 X'* ∧ *qAFresh zs z2 X'*
  using *assms qFresh-imp-ex-qAFresh2* **by** *force*
  hence (*X'* #[[*z1* ∧ *z2*]]-*zs*) = *X'* **using** *qAFresh-qSwap-id* **by** *auto*
  moreover have (*X* #[[*z1* ∧ *z2*]]-*zs*) #= (*X'* #[[*z1* ∧ *z2*]]-*zs*)
  using *assms 1* **by** (*auto simp add*: *qSwap-preserves-alpha*)
  moreover have *X'* #= *X* **using** *1 alpha-sym* **by** *auto*
  moreover have *qGood*(*X* #[[*z1* ∧ *z2*]]-*zs*) **using** *assms qSwap-preserves-qGood*
by *auto*
  ultimately show *?thesis* **using** *alpha-trans* **by** *auto*
qed

lemma *alphaAbs-qFreshAbs-qSwapAbs-id*:
assumes *qGoodAbs A* **and** *qFreshAbs zs z1 A* **and** *qFreshAbs zs z2 A*
shows (*A* $[[*z1* ∧ *z2*]]-*zs*) $= *A*
proof−
  obtain *A'* **where** *1*: *A* $= *A'* **and** *qAFreshAbs zs z1 A'* ∧ *qAFreshAbs zs z2 A'*
  using *assms qFreshAbs-imp-ex-qAFreshAbs2* **by** *force*
  hence (*A'* $[[*z1* ∧ *z2*]]-*zs*) = *A'* **using** *qAFreshAll-qSwapAll-id* **by** *fastforce*

moreover have (*A* $[[*z1* ∧ *z2*]]-*zs*) $= (*A′* $[[*z1* ∧ *z2*]]-*zs*)
using *assms 1* **by** (*auto simp add*: *qSwapAbs-preserves-alphaAbs*)
moreover have *A′* $= *A* **using** *1 alphaAbs-sym* **by** *auto*
moreover have *qGoodAbs* (*A* $[[*z1* ∧ *z2*]]-*zs*) **using** *assms qSwapAbs-preserves-qGoodAbs* **by** *auto*
ultimately show *?thesis* **using** *alphaAbs-trans* **by** *auto*
qed

lemma *alpha-qFresh-qSwap-compose*:
assumes *GOOD*: *qGood X* **and** *qFresh zs y X* **and** *qFresh zs z X*
shows ((*X* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*) #= (*X* #[[*z* ∧ *x*]]-*zs*)
proof−
obtain *X′* **where** *1*: *X* #= *X′* **and** *qAFresh zs y X′* ∧ *qAFresh zs z X′*
using *assms qFresh-imp-ex-qAFresh2* **by** *force*
hence ((*X′* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*) = (*X′* #[[*z* ∧ *x*]]-*zs*)
using *qAFresh-qSwap-compose* **by** *auto*
moreover have ((*X* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*) #= ((*X′* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*)
using *GOOD 1* **by** (*auto simp add*: *qSwap-twice-preserves-alpha*)
moreover have (*X′* #[[*z* ∧ *x*]]-*zs*) #= (*X* #[[*z* ∧ *x*]]-*zs*)
using *GOOD 1* **by** (*auto simp add*: *qSwap-preserves-alpha alpha-sym*)
moreover have *qGood* ((*X* #[[*y* ∧ *x*]]-*zs*) #[[*z* ∧ *y*]]-*zs*)
using *GOOD* **by** (*auto simp add*: *qSwap-twice-preserves-qGood*)
ultimately show *?thesis* **using** *alpha-trans* **by** *auto*
qed

lemma *qAbs-alphaAbs-qSwap-qFresh*:
assumes *GOOD*: *qGood X* **and** *FRESH*: *qFresh xs x′ X*
shows *qAbs xs x X* $= *qAbs xs x′* (*X* #[[*x′* ∧ *x*]]-*xs*)
proof−
obtain *Y* **where** *good-Y*: *qGood Y* **and** *alpha*: *X* #= *Y* **and** *fresh-Y*: *qAFresh xs x′ Y*
using *assms qFresh-imp-ex-qAFresh1* **by** *fastforce*
hence *qAbs xs x Y* $= *qAbs xs x′* (*Y* #[[*x′* ∧ *x*]]-*xs*)
using *qAbs-alphaAbs-qSwap-qAFresh* **by** *blast*
moreover have *qAbs xs x X* $= *qAbs xs x Y*
using *GOOD alpha qAbs-preserves-alpha* **by** *fastforce*
moreover
{have *Y* #[[*x′* ∧ *x*]]-*xs* #= *X* #[[*x′* ∧ *x*]]-*xs*
using *GOOD alpha* **by** (*auto simp add*: *qSwap-preserves-alpha alpha-sym*)
moreover have *qGood* (*Y* #[[*x′* ∧ *x*]]-*xs*) **using** *good-Y qSwap-preserves-qGood* **by** *auto*
ultimately have *qAbs xs x′* (*Y* #[[*x′* ∧ *x*]]-*xs*) $= *qAbs xs x′* (*X* #[[*x′* ∧ *x*]]-*xs*)
using *qAbs-preserves-alpha* **by** *blast*
}
moreover have *qGoodAbs* (*qAbs xs x X*) **using** *GOOD* **by** *simp*
ultimately show *?thesis* **using** *alphaAbs-trans* **by** *blast*
qed

lemma *alphaAbs-qAbs-ex-qFresh-rep*:
assumes *GOOD*: *qGood X* **and** *FRESH*: *qFresh xs x′ X*
shows ∃ *X′*. (*X*,*X′*) ∈ *qSwapped* ∧ *qGood X′* ∧ *qAbs xs x X* $= *qAbs xs x′ X′*
proof−
 have *1*: *qGood* (*X* #[[*x′* ∧ *x*]]-*xs*) **using** *assms qSwap-preserves-qGood* **by** *auto*
 have *2*: (*X*,*X* #[[*x′* ∧ *x*]]-*xs*) ∈ *qSwapped* **by**(*simp add*: *qSwap-qSwapped*)
 show *?thesis*
 apply(*rule exI*[*of* - *X* #[[*x′* ∧ *x*]]-*xs*])
 using *assms 1 2 qAbs-alphaAbs-qSwap-qFresh* **by** *fastforce*
qed

2.6 Alternative statements of the alpha-clause for bound arguments

These alternatives are essentially variations with forall/exists and and qFresh/qAFresh.

2.6.1 First for “qAFresh"

definition *alphaAbs-ex-equal-or-qAFresh*
where
alphaAbs-ex-equal-or-qAFresh xs x X xs′ x′ X′ ==
 (*xs* = *xs′* ∧
 (∃ *y*. (*y* = *x* ∨ *qAFresh xs y X*) ∧ (*y* = *x′* ∨ *qAFresh xs y X′*) ∧
  (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))

definition *alphaAbs-ex-qAFresh*
where
alphaAbs-ex-qAFresh xs x X xs′ x′ X′ ==
 (*xs* = *xs′* ∧
 (∃ *y*. *qAFresh xs y X* ∧ *qAFresh xs y X′* ∧
  (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))

definition *alphaAbs-ex-distinct-qAFresh*
where
alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′ ==
 (*xs* = *xs′* ∧
 (∃ *y*. *y* ∉ {*x*,*x′*} ∧ *qAFresh xs y X* ∧ *qAFresh xs y X′* ∧
  (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))

definition *alphaAbs-all-equal-or-qAFresh*
where
alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′ ==
 (*xs* = *xs′* ∧
 (∀ *y*. (*y* = *x* ∨ *qAFresh xs y X*) ∧ (*y* = *x′* ∨ *qAFresh xs y X′*) ⟶
  (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))

definition *alphaAbs-all-qAFresh*
where
alphaAbs-all-qAFresh xs x X xs′ x′ X′ ==

$(xs = xs' \wedge$
$(\forall\ y.\ qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-all-distinct-qAFresh*
where
alphaAbs-all-distinct-qAFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\forall\ y.\ y \notin \{x,x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$

lemma *alphaAbs-weakestEx-imp-strongestAll*:
assumes *GOOD-X*: *qGood X* **and** *alphaAbs-ex-equal-or-qAFresh* $xs\ x\ X\ xs'\ x'\ X'$
shows *alphaAbs-all-equal-or-qAFresh* $xs\ x\ X\ xs'\ x'\ X'$
proof−
  obtain *y* **where** *xs*: $xs = xs'$ **and**
  yEqFresh: $(y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X')$ **and**
  alpha: $(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)$
  using *assms* **by** (*auto simp add*: *alphaAbs-ex-equal-or-qAFresh-def*)
  show *?thesis*
  using *xs* **unfolding** *alphaAbs-all-equal-or-qAFresh-def*
  proof(*intro conjI allI impI*, *simp*)
    fix *z* **assume** *zFresh*: $(z = x \vee qAFresh\ xs\ z\ X) \wedge (z = x' \vee qAFresh\ xs\ z\ X')$
    have $(X\ \#[[z \wedge x]]\text{-}xs) = ((X\ \#[[y \wedge x]]\text{-}xs)\ \#[[z \wedge y]]\text{-}xs)$
    proof(*cases* $z = x$)
      assume *Case1*: $z = x$
      thus *?thesis* **by**(*auto simp add*: *qSwap-sym*)
    next
      assume *Case2*: $z \neq x$
      hence *z-fresh*: *qAFresh xs z X* **using** *zFresh* **by** *auto*
      show *?thesis*
      proof(*cases* $y = x$)
        assume *Case21*: $y = x$
        show *?thesis* **unfolding** *Case21* **by** *simp*
      next
        assume *Case22*: $y \neq x$
        hence *qAFresh xs y X* **using** *yEqFresh* **by** *auto*
        thus *?thesis* **using** *z-fresh qAFresh-qSwap-compose* **by** *fastforce*
      qed
    qed
    moreover
    have $(X'\ \#[[z \wedge x']]\text{-}xs) = ((X'\ \#[[y \wedge x']]\text{-}xs)\ \#[[z \wedge y]]\text{-}xs)$
    proof(*cases* $z = x'$)
      assume *Case1*: $z = x'$
      thus *?thesis* **by**(*auto simp add*: *qSwap-sym*)
    next
      assume *Case2*: $z \neq x'$
      hence *z-fresh*: $qAFresh\ xs\ z\ X'$ **using** *zFresh* **by** *auto*
      show *?thesis*

proof(*cases* $y = x'$)
assume *Case21*: $y = x'$
show *?thesis* **unfolding** *Case21* **by** *simp*
next
assume *Case22*: $y \neq x'$
hence *qAFresh xs y X′* **using** *yEqFresh* **by** *auto*
thus *?thesis* **using** *z-fresh qAFresh-qSwap-compose* **by** *fastforce*
qed
qed
moreover
{have *qGood* (*X* #[[*y* ∧ *x*]]-*xs*) **using** *GOOD-X qSwap-preserves-qGood* **by** *auto*
hence ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*z* ∧ *y*]]-*xs*) #= ((*X′* #[[*y* ∧ *x′*]]-*xs*) #[[*z* ∧ *y*]]-*xs*)
using *alpha qSwap-preserves-alpha* **by** *fastforce*
}
ultimately show (*X* #[[*z* ∧ *x*]]-*xs*) #= (*X′* #[[*z* ∧ *x′*]]-*xs*) **by** *simp*
qed
qed

lemma *alphaAbs-weakestAll-imp-strongestEx*:
assumes *GOOD*: *qGood X* *qGood X′*
and *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
shows *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
proof−
have *xs*: *xs* = *xs′*
using *assms* **unfolding** *alphaAbs-all-distinct-qAFresh-def* **by** *auto*
obtain *y* **where** *y-not*: $y \notin \{x,x'\}$ **and**
yFresh: *qAFresh xs y X* ∧ *qAFresh xs y X′*
using *GOOD obtain-qFresh*[*of* {*x*,*x′*} {*X*,*X′*}] **by** *auto*
hence (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)
using *assms* **unfolding** *alphaAbs-all-distinct-qAFresh-def* **by** *auto*
thus *?thesis* **unfolding** *alphaAbs-ex-distinct-qAFresh-def* **using** *xs y-not yFresh* **by** *auto*
qed

lemma *alphaAbs-weakestEx-imp-strongestEx*:
assumes *GOOD*: *qGood X*
and *alphaAbs-ex-equal-or-qAFresh xs x X xs′ x′ X′*
shows *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
proof−
obtain *y* **where** *xs*: *xs* = *xs′* **and**
yEqFresh: ($y = x$ ∨ *qAFresh xs y X*) ∧ ($y = x'$ ∨ *qAFresh xs y X′*) **and**
alpha: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)
using *assms* **unfolding** *alphaAbs-ex-equal-or-qAFresh-def* **by** *blast*
hence *goodX′*: *qGood X′*
using *GOOD alpha-qSwap-preserves-qGood* **by** *fastforce*
then obtain *z* **where** *zNot*: $z \notin \{x,x',y\}$ **and**

zFresh: *qAFresh xs z X* ∧ *qAFresh xs z X'*
using *GOOD obtain-qFresh*[*of* {*x*,*x'*,*y*} {*X*,*X'*}] **by** *auto*
have (*X* #[[*z* ∧ *x*]]-*xs*) = ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*z* ∧ *y*]]-*xs*)
proof(*cases y* = *x*, *simp*)
assume *y* ≠ *x* **hence** *qAFresh xs y X* **using** *yEqFresh* **by** *auto*
thus *?thesis* **using** *zFresh qAFresh-qSwap-compose* **by** *fastforce*
qed
moreover have (*X'* #[[*z* ∧ *x'*]]-*xs*) = ((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*z* ∧ *y*]]-*xs*)
proof(*cases y* = *x'*, *simp add*: *qSwap-ident*)
assume *y* ≠ *x'* **hence** *qAFresh xs y X'* **using** *yEqFresh* **by** *auto*
thus *?thesis* **using** *zFresh qAFresh-qSwap-compose* **by** *fastforce*
qed
moreover
{have *qGood* (*X* #[[*y* ∧ *x*]]-*xs*) **using** *GOOD qSwap-preserves-qGood* **by** *auto*
hence ((*X* #[[*y* ∧ *x*]]-*xs*) #[[*z* ∧ *y*]]-*xs*) #= ((*X'* #[[*y* ∧ *x'*]]-*xs*) #[[*z* ∧ *y*]]-*xs*)
using *alpha* **by** (*auto simp add*: *qSwap-preserves-alpha*)
}
ultimately have (*X* #[[*z* ∧ *x*]]-*xs*) #= (*X'* #[[*z* ∧ *x'*]]-*xs*) **by** *simp*
thus *?thesis* **unfolding** *alphaAbs-ex-distinct-qAFresh-def* **using** *xs zNot zFresh*
by *auto*
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*:
(*qAbs xs x X* $= *qAbs xs' x' X'*) = *alphaAbs-ex-distinct-qAFresh xs x X xs' x' X'*
unfolding *alphaAbs-ex-distinct-qAFresh-def* **by** *auto*

corollary *alphaAbs-qAbs-iff-ex-distinct-qAFresh*:
(*qAbs xs x X* $= *qAbs xs' x' X'*) =
(*xs* = *xs'* ∧
(∃ *y*. *y* ∉ {*x*,*x'*} ∧ *qAFresh xs y X* ∧ *qAFresh xs y X'* ∧
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))
unfolding *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*
alphaAbs-ex-distinct-qAFresh-def **by** *fastforce*

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh*:
assumes *qGood X*
shows (*qAbs xs x X* $= *qAbs xs' x' X'*) =
alphaAbs-ex-equal-or-qAFresh xs x X xs' x' X'
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs' x' X'*
let *?Right* = *alphaAbs-ex-equal-or-qAFresh xs x X xs' x' X'*
have *?Left* ⟹ *?Right* **unfolding** *alphaAbs-ex-equal-or-qAFresh-def* **by** *auto*
moreover have *?Right* ⟹ *?Left*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*[*of* - - *X*]
alphaAbs-weakestEx-imp-strongestEx **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

corollary *alphaAbs-qAbs-iff-ex-equal-or-qAFresh*:

assumes *qGood X*
shows
(*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
(*xs* = *xs′* ∧
(∃ *y*. (*y* = *x* ∨ *qAFresh xs y X*) ∧ (*y* = *x′* ∨ *qAFresh xs y X′*) ∧
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))
proof−
have (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
alphaAbs-ex-equal-or-qAFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh* **by** *fastforce*
thus *?thesis* **unfolding** *alphaAbs-ex-equal-or-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-qAFresh*:
assumes *qGood X*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) = *alphaAbs-ex-qAFresh xs x X xs′ x′ X′*
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs′ x′ X′*
let *?Middle* = *alphaAbs-ex-equal-or-qAFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-ex-qAFresh xs x X xs′ x′ X′*
have *?Left* ⟹ *?Right* **unfolding** *alphaAbs-ex-qAFresh-def* **by** *auto*
moreover have *?Right* ⟹ *?Middle*
unfolding *alphaAbs-ex-qAFresh-def alphaAbs-ex-equal-or-qAFresh-def* **by** *auto*
moreover have *?Middle* = *?Left*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh*[*of X*] **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-ex-qAFresh*:
assumes *qGood X*
shows
(*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
(*xs* = *xs′* ∧
(∃ *y*. *qAFresh xs y X* ∧ *qAFresh xs y X′* ∧
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))
proof−
have (*qAbs xs x X* $= *qAbs xs′ x′ X′*) = *alphaAbs-ex-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-qAFresh* **by** *fastforce*
thus *?thesis* **unfolding** *alphaAbs-ex-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh*:
assumes *qGood X* **and** *qAbs xs x X* $= *qAbs xs′ x′ X′*
shows *alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh*
alphaAbs-weakestEx-imp-strongestAll **by** *fastforce*

corollary *alphaAbs-qAbs-imp-all-equal-or-qAFresh*:
assumes *qGood X* **and** (*qAbs xs x X* $= *qAbs xs′ x′ X′*)

shows
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#= (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have *alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-equal-or-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows $(qAbs\ xs\ x\ X\ \$= qAbs\ xs'\ x'\ X') =$
alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′
proof−
let *?Left* = *qAbs xs x X* \$= *qAbs xs′ x′ X′*
let *?MiddleEx* = *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
let *?MiddleAll* = *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′*
have *?Left* ⟹ *?Right*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh* **by** *blast*
moreover have *?Right* ⟹ *?MiddleAll*
unfolding *alphaAbs-all-equal-or-qAFresh-def alphaAbs-all-distinct-qAFresh-def* **by** *auto*
moreover have *?MiddleAll* ⟹ *?MiddleEx*
using *assms alphaAbs-weakestAll-imp-strongestEx* **by** *fastforce*
moreover have *?MiddleEx* ⟹ *?Left*
using *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*[*of - - X*] **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-all-equal-or-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows $(qAbs\ xs\ x\ X\ \$= qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#= (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have $(qAbs\ xs\ x\ X\ \$= qAbs\ xs'\ x'\ X') =$
alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-equal-or-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-imp-alphaAbs-all-qAFresh*:
assumes *qGood X* **and** *qAbs xs x X* \$= *qAbs xs′ x′ X′*
shows *alphaAbs-all-qAFresh xs x X xs′ x′ X′*
proof−
have *alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′*

using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-qAFresh-def alphaAbs-all-equal-or-qAFresh-def*
by *auto*
qed

corollary *alphaAbs-qAbs-imp-all-qAFresh*:
assumes *qGood X* **and** $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X')$
shows
$(xs = xs' \wedge$
$(\forall\ y.\ qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have *alphaAbs-all-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-all-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') = alphaAbs\text{-}all\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
proof−
let *?Left* = *qAbs xs x X $= qAbs xs′ x′ X′*
let *?MiddleEx* = *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
let *?MiddleAll* = *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-all-qAFresh xs x X xs′ x′ X′*
have *?Left* ⟹ *?Right* **using** *assms alphaAbs-qAbs-imp-alphaAbs-all-qAFresh* **by**
blast
moreover have *?Right* ⟹ *?MiddleAll*
unfolding *alphaAbs-all-qAFresh-def alphaAbs-all-distinct-qAFresh-def* **by** *auto*
moreover have *?MiddleAll* ⟹ *?MiddleEx*
using *assms alphaAbs-weakestAll-imp-strongestEx* **by** *fastforce*
moreover have *?MiddleEx* ⟹ *?Left*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*[*of - - X*] **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-all-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\forall\ y.\ qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
alphaAbs-all-qAFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-all-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-imp-alphaAbs-all-distinct-qAFresh*:
assumes *qGood X* **and** *qAbs xs x X* $= *qAbs xs′ x′ X′*
shows *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
proof−
have *alphaAbs-all-equal-or-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh* **by** *blast*
thus *?thesis*
unfolding *alphaAbs-all-distinct-qAFresh-def alphaAbs-all-equal-or-qAFresh-def* **by** *auto*
qed

corollary *alphaAbs-qAbs-imp-all-distinct-qAFresh*:
assumes *qGood X* **and** (*qAbs xs x X* $= *qAbs xs′ x′ X′*)
shows
(*xs* = *xs′* ∧
(∀ *y*. *y* ∉ {*x*,*x′*} ∧ *qAFresh xs y X* ∧ *qAFresh xs y X′* ⟶
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))
proof−
have *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-distinct-qAFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-distinct-qAFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-all-distinct-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs′ x′ X′*
let *?MiddleEx* = *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
let *?MiddleAll* = *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-all-distinct-qAFresh xs x X xs′ x′ X′*
have *?Left* ⟹ *?Right*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-distinct-qAFresh* **by** *blast*
moreover have *?Right* ⟹ *?MiddleAll*
unfolding *alphaAbs-all-distinct-qAFresh-def alphaAbs-all-distinct-qAFresh-def* **by** *auto*
moreover have *?MiddleAll* ⟹ *?MiddleEx*
using *assms alphaAbs-weakestAll-imp-strongestEx* **by** *fastforce*
moreover have *?MiddleEx* ⟹ *?Left*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qAFresh*[*of* - - *X*] **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-all-distinct-qAFresh*:
assumes *qGood X* **and** *qGood X′*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
(*xs* = *xs′* ∧
(∀ *y*. *y* ∉ {*x*,*x′*} ∧ *qAFresh xs y X* ∧ *qAFresh xs y X′* ⟶

$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

proof −

have $(qAbs\ xs\ x\ X\ \$= qAbs\ xs'\ x'\ X') =$
alphaAbs-all-distinct-qAFresh $xs\ x\ X\ xs'\ x'\ X'$

using *assms alphaAbs-qAbs-iff-alphaAbs-all-distinct-qAFresh* **by** *blast*

thus *?thesis* **unfolding** *alphaAbs-all-distinct-qAFresh-def* **.**

qed

2.6.2 Then for "qFresh"

definition *alphaAbs-ex-equal-or-qFresh*
where
alphaAbs-ex-equal-or-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\exists\ y.\ (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \wedge$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-ex-qFresh*
where
alphaAbs-ex-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\exists\ y.\ qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-ex-distinct-qFresh*
where
alphaAbs-ex-distinct-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\exists\ y.\ y \notin \{x,x'\} \wedge qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-all-equal-or-qFresh*
where
alphaAbs-all-equal-or-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \longrightarrow$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-all-qFresh*
where
alphaAbs-all-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$
$(\forall\ y.\ qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \longrightarrow$
$(X \ \#[[y \wedge x]]\text{-}xs) \ \#= (X' \ \#[[y \wedge x']]\text{-}xs)))$

definition *alphaAbs-all-distinct-qFresh*
where
alphaAbs-all-distinct-qFresh $xs\ x\ X\ xs'\ x'\ X' ==$
$(xs = xs' \wedge$

(∀ *y*. *y* ∉ {*x*,*x'*} ∧ *qFresh xs y X* ∧ *qFresh xs y X'* ⟶
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))

lemma *alphaAbs-ex-equal-or-qAFresh-imp-qFresh*:
alphaAbs-ex-equal-or-qAFresh xs x X xs' x' X' ⟹
alphaAbs-ex-equal-or-qFresh xs x X xs' x' X'
unfolding *alphaAbs-ex-equal-or-qAFresh-def alphaAbs-ex-equal-or-qFresh-def*
using *qAFresh-imp-qFresh*[*of* - - *X*] *qAFresh-imp-qFresh*[*of* - - *X'*] **by** *blast*

lemma *alphaAbs-ex-distinct-qAFresh-imp-qFresh*:
alphaAbs-ex-distinct-qAFresh xs x X xs' x' X' ⟹
alphaAbs-ex-distinct-qFresh xs x X xs' x' X'
unfolding *alphaAbs-ex-distinct-qAFresh-def alphaAbs-ex-distinct-qFresh-def*
using *qAFresh-imp-qFresh*[*of* - - *X*] *qAFresh-imp-qFresh*[*of* - - *X'*] **by** *blast*

lemma *alphaAbs-ex-qAFresh-imp-qFresh*:
alphaAbs-ex-qAFresh xs x X xs' x' X' ⟹
alphaAbs-ex-qFresh xs x X xs' x' X'
unfolding *alphaAbs-ex-qAFresh-def alphaAbs-ex-qFresh-def*
using *qAFresh-imp-qFresh*[*of* - - *X*] *qAFresh-imp-qFresh*[*of* - - *X'*] **by** *blast*

lemma *alphaAbs-all-equal-or-qFresh-imp-qAFresh*:
alphaAbs-all-equal-or-qFresh xs x X xs' x' X' ⟹
alphaAbs-all-equal-or-qAFresh xs x X xs' x' X'
unfolding *alphaAbs-all-equal-or-qAFresh-def alphaAbs-all-equal-or-qFresh-def*
using *qAFresh-imp-qFresh*[*of* - - *X*] *qAFresh-imp-qFresh*[*of* - - *X'*] **by** *blast*

lemma *alphaAbs-all-distinct-qFresh-imp-qAFresh*:
alphaAbs-all-distinct-qFresh xs x X xs' x' X' ⟹
alphaAbs-all-distinct-qAFresh xs x X xs' x' X'
using *qAFresh-imp-qFresh*
unfolding *alphaAbs-all-distinct-qAFresh-def alphaAbs-all-distinct-qFresh-def* **by** *fastforce*

lemma *alphaAbs-all-qFresh-imp-qAFresh*:
alphaAbs-all-qFresh xs x X xs' x' X' ⟹
alphaAbs-all-qAFresh xs x X xs' x' X'
using *qAFresh-imp-qFresh*
unfolding *alphaAbs-all-qAFresh-def alphaAbs-all-qFresh-def* **by** *fastforce*

lemma *alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs*:
assumes *GOOD*: *qGood X* **and** *alphaAbs-ex-equal-or-qFresh xs x X xs' x' X'*
shows *qAbs xs x X* $= *qAbs xs' x' X'*
proof−
obtain *y* **where** *xs*: *xs* = *xs'* **and**
yEqFresh: (*y* = *x* ∨ *qFresh xs y X*) ∧ (*y* = *x'* ∨ *qFresh xs y X'*) **and**
alphaXX'yx: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)
using *assms* **unfolding** *alphaAbs-ex-equal-or-qFresh-def* **by** *blast*
have ∃ *Y*. *X* #= *Y* ∧ (*y* = *x* ∨ *qAFresh xs y Y*)

proof(*cases y = x*)
assume *Case1*: $y = x$ **hence** $X \#= X$ **using** *GOOD alpha-refl* **by** *auto*
thus *?thesis* **using** *Case1* **by** *fastforce*
next
assume *Case2*: $y \neq x$ **hence** *qFresh xs y X* **using** *yEqFresh* **by** *blast*
then obtain Y **where** $X \#= Y$ **and** *qAFresh xs y Y*
using *GOOD qFresh-imp-ex-qAFresh1* **by** *fastforce*
thus *?thesis* **by** *auto*
qed
then obtain Y **where** *alphaXY*: $X \#= Y$ **and** *yEqAFresh*: $y = x \vee$ *qAFresh xs y Y* **by** *blast*
hence $(X \#[[y \wedge x]]\text{-}xs) \#= (Y \#[[y \wedge x]]\text{-}xs)$
using *GOOD qSwap-preserves-alpha* **by** *fastforce*
hence *alphaYXyx*: $(Y \#[[y \wedge x]]\text{-}xs) \#= (X \#[[y \wedge x]]\text{-}xs)$ **using** *alpha-sym* **by** *auto*
have *goodY*: *qGood Y* **using** *alphaXY GOOD alpha-preserves-qGood* **by** *auto*
hence *goodYyx*: *qGood*$(Y \#[[y \wedge x]]\text{-}xs)$ **using** *qSwap-preserves-qGood* **by** *auto*

have *good'*: *qGood X'*
using *GOOD alphaXX'yx alpha-qSwap-preserves-qGood* **by** *fastforce*
have $\exists\ Y'.\ X' \#= Y' \wedge (y = x' \vee$ *qAFresh xs y Y'*$)$
proof(*cases y = x'*)
assume *Case1*: $y = x'$ **hence** $X' \#= X'$ **using** *good' alpha-refl* **by** *auto*
thus *?thesis* **using** *Case1* **by** *fastforce*
next
assume *Case2*: $y \neq x'$ **hence** *qFresh xs y X'* **using** *yEqFresh* **by** *blast*
then obtain Y' **where** $X' \#= Y'$ **and** *qAFresh xs y Y'*
using *good' qFresh-imp-ex-qAFresh1* **by** *fastforce*
thus *?thesis* **by** *auto*
qed
then obtain Y' **where** *alphaX'Y'*: $X' \#= Y'$ **and**
yEqAFresh': $y = x' \vee$ *qAFresh xs y Y'* **by** *blast*
hence $(X' \#[[y \wedge x']]\text{-}xs) \#= (Y' \#[[y \wedge x']]\text{-}xs)$
using *good'* **by** (*auto simp add*: *qSwap-preserves-alpha*)
hence $(Y \#[[y \wedge x]]\text{-}xs) \#= (Y' \#[[y \wedge x']]\text{-}xs)$
using *goodYyx alphaYXyx alphaXX'yx alpha-trans* **by** *blast*
hence *alphaAbs-ex-equal-or-qAFresh xs x Y xs x' Y'*
unfolding *alphaAbs-ex-equal-or-qAFresh-def* **using** *yEqAFresh yEqAFresh'* **by** *fastforce*
hence *qAbs xs x Y* \$= *qAbs xs x' Y'*
using *goodY alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh*[*of Y xs x xs*] **by** *fastforce*
moreover have *qAbs xs x X* \$= *qAbs xs x Y*
using *alphaXY GOOD qAbs-preserves-alpha* **by** *fastforce*
moreover
{**have** *1*: $Y' \#= X'$ **using** *alphaX'Y' alpha-sym* **by** *auto*
hence *qGood Y'* **using** *good' alpha-preserves-qGood* **by** *auto*
hence *qAbs xs x' Y'* \$= *qAbs xs x' X'*
using *1 GOOD qAbs-preserves-alpha* **by** *fastforce*

}
moreover have *qGoodAbs*(*qAbs xs x X*) **using** *GOOD* **by** *simp*
ultimately have *qAbs xs x X* $= *qAbs xs x′ X′*
using *alphaAbs-trans-twice* **by** *blast*
thus *?thesis* **using** *xs* **by** *simp*
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qFresh*:
assumes *GOOD*: *qGood X*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
alphaAbs-ex-equal-or-qFresh xs x X xs′ x′ X′
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs′ x′ X′*
let *?Middle* = *alphaAbs-ex-equal-or-qAFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-ex-equal-or-qFresh xs x X xs′ x′ X′*
have *?Right* ⟹ *?Left*
using *assms alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs* **by** *blast*
moreover have *?Left* ⟹ *?Middle*
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh* **by** *blast*
moreover have *?Middle* ⟹ *?Right* **using**
alphaAbs-ex-equal-or-qAFresh-imp-qFresh **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-ex-equal-or-qFresh*:
assumes *GOOD*: *qGood X*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
(*xs* = *xs′* ∧
(∃ *y*. (*y* = *x* ∨ *qFresh xs y X*) ∧ (*y* = *x′* ∨ *qFresh xs y X′*) ∧
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X′* #[[*y* ∧ *x′*]]-*xs*)))
proof−
have (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
alphaAbs-ex-equal-or-qFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-ex-equal-or-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-qFresh*:
assumes *GOOD*: *qGood X*
shows (*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
alphaAbs-ex-qFresh xs x X xs′ x′ X′
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs′ x′ X′*
let *?Middle1* = *alphaAbs-ex-qAFresh xs x X xs′ x′ X′*
let *?Middle2* = *alphaAbs-ex-equal-or-qFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-ex-qFresh xs x X xs′ x′ X′*
have *?Left* ⟹ *?Middle1* **unfolding** *alphaAbs-ex-qAFresh-def* **by** *auto*
moreover have *?Middle1* ⟹ *?Right* **using** *alphaAbs-ex-qAFresh-imp-qFresh*
by *fastforce*

moreover have *?Right* $\Longrightarrow$ *?Middle2*
unfolding *alphaAbs-ex-qFresh-def alphaAbs-ex-equal-or-qFresh-def* **by** *auto*
moreover have *?Middle2* $\Longrightarrow$ *?Left*
using *assms alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs* **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-ex-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
alphaAbs-ex-qFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-ex-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-ex-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
alphaAbs-ex-distinct-qFresh xs x X xs′ x′ X′
proof−
let *?Left* = *qAbs xs x X* $= *qAbs xs′ x′ X′*
let *?Middle1* = *alphaAbs-ex-distinct-qAFresh xs x X xs′ x′ X′*
let *?Middle2* = *alphaAbs-ex-equal-or-qFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-ex-distinct-qFresh xs x X xs′ x′ X′*
have *?Left* $\Longrightarrow$ *?Middle1* **unfolding** *alphaAbs-ex-distinct-qAFresh-def* **by** *auto*
moreover have *?Middle1* $\Longrightarrow$ *?Right* **using** *alphaAbs-ex-distinct-qAFresh-imp-qFresh*
by *fastforce*
moreover have *?Right* $\Longrightarrow$ *?Middle2*
unfolding *alphaAbs-ex-distinct-qFresh-def alphaAbs-ex-equal-or-qFresh-def* **by**
auto
moreover have *?Middle2* $\Longrightarrow$ *?Left*
using *assms alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs* **by** *fastforce*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-ex-distinct-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ y \notin \{x, x'\} \wedge qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
alphaAbs-ex-distinct-qFresh xs x X xs′ x′ X′

using *assms alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-ex-distinct-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh*:
assumes *qGood X* **and** *qAbs xs x X* $= *qAbs xs' x' X'*
shows *alphaAbs-all-equal-or-qFresh xs x X xs' x' X'*
proof−
have *qGoodAbs*(*qAbs xs x X*) **using** *assms* **by** *auto*
hence *qGoodAbs*(*qAbs xs' x' X'*) **using** *assms alphaAbs-preserves-qGoodAbs* **by** *blast*
hence *GOOD*: *qGood X* ∧ *qGood X'* **using** *assms* **by** *auto*
have *xs*: *xs* = *xs'* **using** *assms* **by** *auto*
show *?thesis*
unfolding *alphaAbs-all-equal-or-qFresh-def* **using** *xs*
proof(*intro conjI impI allI*, *simp*)
fix *y*
assume *yEqFresh*: (*y* = *x* ∨ *qFresh xs y X*) ∧ (*y* = *x'* ∨ *qFresh xs y X'*)
have ∃ *Y*. *X* #= *Y* ∧ (*y* = *x* ∨ *qAFresh xs y Y*)
proof(*cases y* = *x*)
assume *Case1*: *y* = *x* **hence** *X* #= *X* **using** *GOOD alpha-refl* **by** *auto*
thus *?thesis* **using** *Case1* **by** *fastforce*
next
assume *Case2*: *y* ≠ *x* **hence** *qFresh xs y X* **using** *yEqFresh* **by** *blast*
then obtain *Y* **where** *X* #= *Y* **and** *qAFresh xs y Y*
using *GOOD qFresh-imp-ex-qAFresh1* **by** *blast*
thus *?thesis* **by** *auto*
qed
then obtain *Y* **where** *alphaXY*: *X* #= *Y* **and** *yEqAFresh*: *y* = *x* ∨ *qAFresh xs y Y* **by** *blast*
hence *alphaXYyx*: (*X* #[[*y* ∧ *x*]]-*xs*) #= (*Y* #[[*y* ∧ *x*]]-*xs*)
using *GOOD* **by** (*auto simp add*: *qSwap-preserves-alpha*)
have *goodY*: *qGood Y* **using** *GOOD alphaXY alpha-preserves-qGood* **by** *auto*

have ∃ *Y'*. *X'* #= *Y'* ∧ (*y* = *x'* ∨ *qAFresh xs y Y'*)
proof(*cases y* = *x'*)
assume *Case1*: *y* = *x'* **hence** *X'* #= *X'* **using** *GOOD alpha-refl* **by** *auto*
thus *?thesis* **using** *Case1* **by** *fastforce*
next
assume *Case2*: *y* ≠ *x'* **hence** *qFresh xs y X'* **using** *yEqFresh* **by** *blast*
then obtain *Y'* **where** *X'* #= *Y'* **and** *qAFresh xs y Y'*
using *GOOD qFresh-imp-ex-qAFresh1* **by** *blast*
thus *?thesis* **by** *auto*
qed
then obtain *Y'* **where** *alphaX'Y'*: *X'* #= *Y'* **and**
yEqAFresh': *y* = *x'* ∨ *qAFresh xs y Y'* **by** *blast*
hence (*X'* #[[*y* ∧ *x'*]]-*xs*) #= (*Y'* #[[*y* ∧ *x'*]]-*xs*)
using *GOOD* **by** (*auto simp add*: *qSwap-preserves-alpha*)
hence *alphaY'X'yx'*: (*Y'* #[[*y* ∧ *x'*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*) **using** *al-*

pha-sym **by** *auto*
have *goodY'*: *qGood Y'* **using** *GOOD alphaX'Y' alpha-preserves-qGood* **by** *auto*

have *1*: Y #= X **using** *alphaXY alpha-sym* **by** *auto*
hence *qGood Y* **using** *GOOD alpha-preserves-qGood* **by** *auto*
hence *2*: *qAbs xs x Y* $= *qAbs xs x X*
using *1 GOOD qAbs-preserves-alpha* **by** *blast*
moreover have *qAbs xs x' X'* $= *qAbs xs x' Y'*
using *alphaX'Y' GOOD qAbs-preserves-alpha* **by** *blast*
moreover
{**have** *qGoodAbs*(*qAbs xs x X*) **using** *GOOD* **by** *simp*
hence *qGoodAbs*(*qAbs xs x Y*) **using** *2 alphaAbs-preserves-qGoodAbs* **by** *fastforce*
}
ultimately have *qAbs xs x Y* $= *qAbs xs x' Y'*
using *assms xs alphaAbs-trans-twice* **by** *blast*
hence *alphaAbs-all-equal-or-qAFresh xs x Y xs x' Y'*
using *goodY goodY' alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh* **by** *blast*
hence (Y #[[$y \wedge x$]]-*xs*) #= (Y' #[[$y \wedge x'$]]-*xs*)
unfolding *alphaAbs-all-equal-or-qAFresh-def*
using *yEqAFresh yEqAFresh'* **by** *auto*
moreover have *qGood* (X #[[$y \wedge x$]]-*xs*) **using** *GOOD qSwap-preserves-qGood* **by** *auto*
ultimately show (X #[[$y \wedge x$]]-*xs*) #= (X' #[[$y \wedge x'$]]-*xs*)
using *alphaXYyx alphaY'X'yx' alpha-trans-twice* **by** *blast*
qed
qed

corollary *alphaAbs-qAbs-imp-all-equal-or-qFresh*:
assumes *qGood X* **and** (*qAbs xs x X* $= *qAbs xs' x' X'*)
shows
($xs = xs' \wedge$
($\forall$ y. ($y = x \vee$ *qFresh xs y X*) $\wedge$ ($y = x' \vee$ *qFresh xs y X'*) $\longrightarrow$
(X #[[$y \wedge x$]]-*xs*) #= (X' #[[$y \wedge x'$]]-*xs*)))
proof−
have *alphaAbs-all-equal-or-qFresh xs x X xs' x' X'*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-equal-or-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qFresh*:
assumes *qGood X* **and** *qGood X'*
shows (*qAbs xs x X* $= *qAbs xs' x' X'*) =
alphaAbs-all-equal-or-qFresh xs x X xs' x' X'
proof−
let *?Left* = (*qAbs xs x X* $= *qAbs xs' x' X'*)
let *?Middle* = *alphaAbs-all-equal-or-qAFresh xs x X xs' x' X'*
let *?Right* = *alphaAbs-all-equal-or-qFresh xs x X xs' x' X'*

have *?Left* $\Longrightarrow$ *?Right*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh* **by** *blast*
moreover have *?Right* $\Longrightarrow$ *?Middle*
using *alphaAbs-all-equal-or-qFresh-imp-qAFresh* **by** *fastforce*
moreover have *?Middle ==> ?Left*
using *assms alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-all-equal-or-qFresh*:
assumes *qGood X* **and** *qGood X′*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
alphaAbs-all-equal-or-qFresh xs x X xs′ x′ X′
using *assms alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-equal-or-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-imp-alphaAbs-all-qFresh*:
assumes *qGood X* **and** *qAbs xs x X \$= qAbs xs′ x′ X′*
shows *alphaAbs-all-qFresh xs x X xs′ x′ X′*
proof−
let *?Left* = (*qAbs xs x X \$= qAbs xs′ x′ X′*)
let *?Middle* = *alphaAbs-all-equal-or-qFresh xs x X xs′ x′ X′*
let *?Right* = *alphaAbs-all-qFresh xs x X xs′ x′ X′*
have *?Left* $\Longrightarrow$ *?Middle*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh* **by** *blast*
moreover have *?Middle* $\Longrightarrow$ *?Right*
unfolding *alphaAbs-all-equal-or-qFresh-def alphaAbs-all-qFresh-def* **by** *auto*
ultimately show *?thesis* **using** *assms* **by** *blast*
qed

corollary *alphaAbs-qAbs-imp-all-qFresh*:
assumes *qGood X* **and** (*qAbs xs x X \$= qAbs xs′ x′ X′*)
shows
$(xs = xs' \wedge$
$(\forall\ y.\ qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
proof−
have *alphaAbs-all-qFresh xs x X xs′ x′ X′*
using *assms alphaAbs-qAbs-imp-alphaAbs-all-qFresh* **by** *blast*
thus *?thesis* **unfolding** *alphaAbs-all-qFresh-def* **.**
qed

lemma *alphaAbs-qAbs-iff-alphaAbs-all-qFresh*:

assumes *qGood X* **and** *qGood X'*
shows (*qAbs xs x X* $= *qAbs xs' x' X'*) =
  alphaAbs-all-qFresh xs x X xs' x' X'
proof –
  let *?Left* = (*qAbs xs x X* $= *qAbs xs' x' X'*)
  let *?Middle* = *alphaAbs-all-qAFresh xs x X xs' x' X'*
  let *?Right* = *alphaAbs-all-qFresh xs x X xs' x' X'*
  have *?Left* ⟹ *?Right*
  using *assms alphaAbs-qAbs-imp-alphaAbs-all-qFresh* **by** *blast*
  moreover have *?Right* ⟹ *?Middle*
  using *alphaAbs-all-qFresh-imp-qAFresh* **by** *fastforce*
  moreover have *?Middle* ⟹ *?Left*
  using *assms alphaAbs-qAbs-iff-alphaAbs-all-qAFresh* **by** *blast*
  ultimately show *?thesis* **by** *blast*
qed

corollary *alphaAbs-qAbs-iff-all-qFresh*:
assumes *qGood X* **and** *qGood X'*
shows (*qAbs xs x X* $= *qAbs xs' x' X'*) =
  (*xs* = *xs'* ∧
   (∀ *y*. *qFresh xs y X* ∧ *qFresh xs y X'* ⟶
    (*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))
proof –
  have (*qAbs xs x X* $= *qAbs xs' x' X'*) =
   alphaAbs-all-qFresh xs x X xs' x' X'
  using *assms alphaAbs-qAbs-iff-alphaAbs-all-qFresh* **by** *blast*
  thus *?thesis* **unfolding** *alphaAbs-all-qFresh-def* **.**
qed

end

end

3 Environments and Substitution for Quasi-Terms

theory *QuasiTerms-Environments-Substitution*
imports *QuasiTerms-PickFresh-Alpha*
begin

Inside this theory, since anyway all the interesting properties hold only modulo alpha, we forget completely about qAFresh and use only qFresh.

In this section we define, for quasi-terms, parallel substitution according to *environments*. This is the most general kind of substitution – an environment, i.e., a partial map from variables to quasi-terms, indicates which quasi-term (if any) to be substituted for each variable; substitution is then applied to a subject quasi-term and an environment. In order to “keep up" with the notion of good quasi-term, we define good environments and show that substitution preserves goodness. Since, unlike swapping, substitution does not behave well w.r.t. quasi-terms (but only w.r.t. terms, i.e., to alpha-equivalence classes), here we prove the minimum amount of properties required for properly lifting parallel substitution to terms. Then compositionality properties of parallel substitution will be proved directly for terms.

3.1 Environments

type-synonym *('index,'bindex,'varSort,'var,'opSym)qEnv* =
  'varSort ⇒ 'var ⇒ ('index,'bindex,'varSort,'var,'opSym)qTerm option

context *FixVars*
begin

definition *qGoodEnv* :: *('index,'bindex,'varSort,'var,'opSym)qEnv ⇒ bool*
where
qGoodEnv rho ==
 (∀ xs. liftAll qGood (rho xs)) ∧
 (∀ ys. |{y. rho ys y ≠ None}| <o |UNIV :: 'var set|)

definition *qFreshEnv* **where**
qFreshEnv zs z rho ==
 rho zs z = None ∧ (∀ xs. liftAll (qFresh zs z) (rho xs))

definition *alphaEnv* **where**
alphaEnv =
 {(rho,rho'). ∀ xs. sameDom (rho xs) (rho' xs) ∧
  liftAll2 (λX X'. X #= X') (rho xs) (rho' xs)}

abbreviation *alphaEnv-abbrev* ::
('index,'bindex,'varSort,'var,'opSym)qEnv ⇒
 ('index,'bindex,'varSort,'var,'opSym)qEnv ⇒ bool (**infix** ‹&=› *50*)
where
rho &= rho' == (rho,rho') ∈ alphaEnv

definition *pickQFreshEnv*
where
pickQFreshEnv xs V XS Rho ==
 pickQFresh xs (V ∪ (⋃ rho ∈ Rho. {x. rho xs x ≠ None}))
  (XS ∪ (⋃ rho ∈ Rho. {X. ∃ ys y. rho ys y = Some X}))

lemma *qGoodEnv-imp-card-of-qTerm*:
assumes *qGoodEnv rho*
shows *|{X. ∃ y. rho ys y = Some X}| <o |UNIV :: 'var set|*
proof−
  let *?rel = {(y,X). rho ys y = Some X}*

let *?Left* = {*X*. ∃ *y*. *rho ys y* = *Some X*}
let *?Left′* = {*y*. ∃ *X*. *rho ys y* = *Some X*}
have ⋀ *y X X′*. (*y*,*X*) ∈ *?rel* ∧ (*y*,*X′*) ∈ *?rel* ⟶ *X* = *X′* **by** *force*
hence |*?Left*| ≤*o* |*?Left′*| **using** *card-of-inj-rel*[*of ?rel*] **by** *auto*
moreover have |*?Left′*| <*o* |*UNIV* :: *′var set*| **using** *assms* **unfolding** *qGood-Env-def* **by** *auto*
ultimately show *?thesis* **using** *ordLeq-ordLess-trans* **by** *blast*
qed

lemma *qGoodEnv-imp-card-of-qTerm2*:
assumes *qGoodEnv rho*
shows |{*X*. ∃ *ys y*. *rho ys y* = *Some X*}| <*o* |*UNIV* :: *′var set*|
proof−
let *?Left* = {*X*. ∃ *ys y*. *rho ys y* = *Some X*}
let *?F* = λ *ys*. {*X*. ∃ *y*. *rho ys y* = *Some X*}
have *?Left* = (⋃ *ys*. *?F ys*) **by** *auto*
moreover have ∀ *ys*. |*?F ys*| <*o* |*UNIV* :: *′var set*|
using *assms qGoodEnv-imp-card-of-qTerm* **by** *auto*
ultimately show *?thesis*
using *var-regular-INNER varSort-lt-var-INNER* **by**(*force simp add*: *regular-UNION*)
qed

lemma *qGoodEnv-iff*:
qGoodEnv rho =
((∀ *xs*. *liftAll qGood* (*rho xs*)) ∧
(∀ *ys*. |{*y*. *rho ys y* ≠ *None*}| <*o* |*UNIV* :: *′var set*|) ∧
|{*X*. ∃ *ys y*. *rho ys y* = *Some X*}| <*o* |*UNIV* :: *′var set*|)
unfolding *qGoodEnv-def* **apply** *auto*
apply(*rule qGoodEnv-imp-card-of-qTerm2*) **unfolding** *qGoodEnv-def* **by** *simp*

lemma *alphaEnv-refl*:
qGoodEnv rho ⟹ *rho* &= *rho*
using *alpha-refl*
unfolding *alphaEnv-def qGoodEnv-def liftAll-def liftAll2-def sameDom-def* **by** *fast-force*

lemma *alphaEnv-sym*:
rho &= *rho′* ⟹ *rho′* &= *rho*
using *alpha-sym* **unfolding** *alphaEnv-def liftAll2-def sameDom-def* **by** *fastforce*

lemma *alphaEnv-trans*:
assumes *good*: *qGoodEnv rho* **and**
alpha1: *rho* &= *rho′* **and** *alpha2*: *rho′* &= *rho″*
shows *rho* &= *rho″*
using *assms* **unfolding** *alphaEnv-def*
apply(*auto*)
using *sameDom-trans* **apply** *blast*
unfolding *liftAll2-def* **proof**(*auto*)
fix *xs x X X″*

assume *rho*: *rho xs x* = *Some X* **and** *rho''*: *rho'' xs x* = *Some X''*
moreover have (*rho xs x* = *None*) = (*rho' xs x* = *None*)
using *alpha1* **unfolding** *alphaEnv-def sameDom-def* **by** *auto*
ultimately obtain *X'* **where** *rho'*: *rho' xs x* = *Some X'* **by** *auto*
hence *X* #= *X'* **using** *alpha1 rho* **unfolding** *alphaEnv-def liftAll2-def* **by** *auto*
moreover have *X'* #= *X''*
using *alpha2 rho' rho''* **unfolding** *alphaEnv-def liftAll2-def* **by** *auto*
moreover have *qGood X* **using** *good rho* **unfolding** *qGoodEnv-def liftAll-def* **by** *auto*
ultimately show *X* #= *X''* **using** *alpha-trans* **by** *blast*
qed

lemma *pickQFreshEnv-card-of*:
assumes *Vvar*: |*V*| <o |*UNIV* :: *'var set*| **and** *XSvar*: |*XS*| <o |*UNIV* :: *'var set*| **and**
good: ∀ *X* ∈ *XS*. *qGood X* **and**
Rhovar: |*Rho*| <o |*UNIV* :: *'var set*| **and** *RhoGood*: ∀ *rho* ∈ *Rho*. *qGoodEnv rho*
shows
pickQFreshEnv xs V XS Rho ∉ *V* ∧
(∀ *X* ∈ *XS*. *qFresh xs* (*pickQFreshEnv xs V XS Rho*) *X*) ∧
(∀ *rho* ∈ *Rho*. *qFreshEnv xs* (*pickQFreshEnv xs V XS Rho*) *rho*)
proof−
let *?z* = *pickQFreshEnv xs V XS Rho*
let *?V2* = ⋃ *rho* ∈ *Rho*. {*x*. *rho xs x* ≠ *None*} **let** *?W* = *V* ∪ *?V2*
let *?XS2* = ⋃ *rho* ∈ *Rho*. {*X*. ∃ *ys y*. *rho ys y* = *Some X*} **let** *?YS* = *XS* ∪ *?XS2*
have |*?W*| <o |*UNIV* :: *'var set*|
proof−
have ∀ *rho* ∈ *Rho*. |{*x*. *rho xs x* ≠ *None*}| <o |*UNIV* :: *'var set*|
using *RhoGood* **unfolding** *qGoodEnv-iff* **using** *qGoodEnv-iff* **by** *auto*
hence |*?V2*| <o |*UNIV* :: *'var set*|
using *var-regular-INNER Rhovar* **by** (*auto simp add*: *regular-UNION*)
thus *?thesis* **using** *var-infinite-INNER Vvar card-of-Un-ordLess-infinite* **by** *auto*
qed
moreover have |*?YS*| <o |*UNIV* :: *'var set*|
proof−
have ∀ *rho* ∈ *Rho*. |{*X*. ∃ *ys y*. *rho ys y* = *Some X*}| <o |*UNIV* :: *'var set*|
using *RhoGood* **unfolding** *qGoodEnv-iff* **by** *auto*
hence |*?XS2*| <o |*UNIV* :: *'var set*|
using *var-regular-INNER Rhovar* **by** (*auto simp add*: *regular-UNION*)
thus *?thesis* **using** *var-infinite-INNER XSvar card-of-Un-ordLess-infinite* **by** *auto*
qed
moreover have ∀ *Y* ∈ *?YS*. *qGood Y*
using *good RhoGood* **unfolding** *qGoodEnv-iff liftAll-def* **by** *blast*
ultimately
have *?z* ∉ *?W* ∧ (∀ *Y* ∈ *?YS*. *qFresh xs ?z Y*)

unfolding *pickQFreshEnv-def* **using** *pickQFresh-card-of*[*of ?W ?YS*] **by** *auto*
thus *?thesis* **unfolding** *qFreshEnv-def liftAll-def* **by**(*auto*)
qed

lemma *pickQFreshEnv*:
assumes *Vvar*: |*V*| <*o* |*UNIV* :: '*var set*| ∨ *finite V*
and *XSvar*: |*XS*| <*o* |*UNIV* :: '*var set*| ∨ *finite XS*
and *good*: ∀ *X* ∈ *XS*. *qGood X*
and *Rhovar*: |*Rho*| <*o* |*UNIV* :: '*var set*| ∨ *finite Rho*
and *RhoGood*: ∀ *rho* ∈ *Rho*. *qGoodEnv rho*
shows
pickQFreshEnv xs V XS Rho ∉ *V* ∧
(∀ *X* ∈ *XS*. *qFresh xs* (*pickQFreshEnv xs V XS Rho*) *X*) ∧
(∀ *rho* ∈ *Rho*. *qFreshEnv xs* (*pickQFreshEnv xs V XS Rho*) *rho*)
proof−
have *1*: |*V*| <*o* |*UNIV* :: '*var set*| ∧ |*XS*| <*o* |*UNIV* :: '*var set*| ∧ |*Rho*| <*o* |*UNIV* :: '*var set*|
using *assms var-infinite-INNER* **by**(*auto simp add*: *finite-ordLess-infinite2*)
show *?thesis*
apply(*rule pickQFreshEnv-card-of*)
using *assms 1* **by** *auto*
qed

corollary *obtain-qFreshEnv*:
fixes *XS*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm set* **and**
Rho::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv set* **and** *rho*
assumes *Vvar*: |*V*| <*o* |*UNIV* :: '*var set*| ∨ *finite V*
and *XSvar*: |*XS*| <*o* |*UNIV* :: '*var set*| ∨ *finite XS*
and *good*: ∀ *X* ∈ *XS*. *qGood X*
and *Rhovar*: |*Rho*| <*o* |*UNIV* :: '*var set*| ∨ *finite Rho*
and *RhoGood*: ∀ *rho* ∈ *Rho*. *qGoodEnv rho*
shows
∃ *z*. *z* ∉ *V* ∧
(∀ *X* ∈ *XS*. *qFresh xs z X*) ∧ (∀ *rho* ∈ *Rho*. *qFreshEnv xs z rho*)
apply(*rule exI*[*of - pickQFreshEnv xs V XS Rho*])
using *assms* **by**(*rule pickQFreshEnv*)

3.2 Parallel substitution

definition *aux-qPsubst-ignoreFirst* ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv* ∗ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
+
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv* ∗ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTermItem*
where
aux-qPsubst-ignoreFirst K ==
case K of Inl (*rho*,*X*) ⇒ *termIn X*
|*Inr* (*rho*,*A*) ⇒ *absIn A*

lemma *aux-qPsubst-ignoreFirst-qTermLessQSwapped-wf*:
wf(*inv-image qTermQSwappedLess aux-qPsubst-ignoreFirst*)
using *qTermQSwappedLess-wf wf-inv-image* **by** *auto*

function
qPsubst ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
and
qPsubstAbs ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
where
qPsubst rho (*qVar xs x*) = (*case rho xs x of None* ⇒ *qVar xs x*| *Some X* ⇒ *X*)
|
qPsubst rho (*qOp delta inp binp*) =
qOp delta (*lift* (*qPsubst rho*) *inp*) (*lift* (*qPsubstAbs rho*) *binp*)
|
qPsubstAbs rho (*qAbs xs x X*) =
(*let x'* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*} *in qAbs xs x'* (*qPsubst rho* (*X* #[[*x'* ∧
x]]-*xs*)))
by(*pat-completeness*, *auto*)
termination
apply(*relation inv-image qTermQSwappedLess aux-qPsubst-ignoreFirst*)
apply(*simp add*: *aux-qPsubst-ignoreFirst-qTermLessQSwapped-wf*)
by(*auto simp add*: *qTermQSwappedLess-def qTermLess-modulo-def*
aux-qPsubst-ignoreFirst-def qSwap-qSwapped)

abbreviation *qPsubst-abbrev* ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv*
⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* (‹- #[[-]]›)
where *X* #[[*rho*]] == *qPsubst rho X*

abbreviation *qPsubstAbs-abbrev* ::
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* ⇒ ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv*
⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* (‹- $[[-]]›)
where *A* $[[*rho*]] == *qPsubstAbs rho A*

lemma *qPsubstAll-preserves-qGoodAll*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and** *rho*
assumes *GOOD-ENV*: *qGoodEnv rho*
shows
(*qGood X* ⟶ *qGood* (*X* #[[*rho*]])) ∧ (*qGoodAbs A* ⟶ *qGoodAbs* (*A* $[[*rho*]]))
proof(*induction rule*: *qTerm-induct*[*of* - - *X A*])

case (*Var xs x*)
show *?case*
using *GOOD-ENV* **unfolding** *qGoodEnv-iff liftAll-def*
by(*cases rho xs x*, *auto*)
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
assume *g*: *qGood* (*qOp delta inp binp*)
hence *0*: *liftAll qGood* (*lift* (*qPsubst rho*) *inp*) ∧
liftAll qGoodAbs (*lift* (*qPsubstAbs rho*) *binp*)
using *Op* **unfolding** *liftAll-lift-comp comp-def*
by (*simp-all add*: *Let-def liftAll-mp*)
have {*i*. *lift* (*qPsubst rho*) *inp i* ≠ *None*} = {*i*. *inp i* ≠ *None*} ∧
{*i*. *lift* (*qPsubstAbs rho*) *binp i* ≠ *None*} = {*i*. *binp i* ≠ *None*}
by *simp* (*meson lift-Some*)
hence |{*i*. ∃ *y*. *lift* (*qPsubst rho*) *inp i* = *Some y*}| <*o* |*UNIV*:: *'var set*|
and |{*i*. ∃ *y*. *lift* (*qPsubstAbs rho*) *binp i* = *Some y*}| <*o* |*UNIV*:: *'var set*|
using *g* **by** (*auto simp*: *liftAll-def*)
thus *qGood qOp delta inp binp* #[[*rho*]] **using** *0* **by** *simp*
qed
next
case (*Abs xs x X*)
show *?case* **proof** *safe*
assume *g*: *qGoodAbs* (*qAbs xs x X*)
let *?x'* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*} **let** *?X'* = *X* #[[*?x'* ∧ *x*]]-*xs*
have *qGood ?X'* **using** *g qSwap-preserves-qGood* **by** *auto*
moreover have (*X*,*?X'*) ∈ *qSwapped* **using** *qSwap-qSwapped* **by** *fastforce*
ultimately have *qGood* (*qPsubst rho ?X'*) **using** *Abs.IH* **by** *simp*
thus *qGoodAbs* ((*qAbs xs x X*) $[[*rho*]]) **by** (*simp add*: *Let-def*)
qed
qed

corollary *qPsubst-preserves-qGood*:
⟦*qGoodEnv rho*; *qGood X*⟧ ⟹ *qGood* (*X* #[[*rho*]])
using *qPsubstAll-preserves-qGoodAll* **by** *auto*

corollary *qPsubstAbs-preserves-qGoodAbs*:
⟦*qGoodEnv rho*; *qGoodAbs A*⟧ ⟹ *qGoodAbs* (*A* $[[*rho*]])
using *qPsubstAll-preserves-qGoodAll* **by** *auto*

lemma *qPsubstAll-preserves-qFreshAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and** *rho*
assumes *GOOD-ENV*: *qGoodEnv rho*
shows
(*qFresh zs z X* ⟶
(*qGood X* ∧ *qFreshEnv zs z rho* ⟶ *qFresh zs z* (*X* #[[*rho*]]))) ∧
(*qFreshAbs zs z A* ⟶
(*qGoodAbs A* ∧ *qFreshEnv zs z rho* ⟶ *qFreshAbs zs z* (*A* $[[*rho*]])))

proof(*induction rule*: *qTerm-induct*[*of* - - *X A*])
  case (*Var xs x*)
  then show *?case*
  unfolding *qFreshEnv-def liftAll-def* **by** (*cases rho xs x*) *auto*
next
  case (*Op delta inp binp*)
  thus *?case*
  by (*auto simp add*: *lift-def liftAll-def qFreshEnv-def split*: *option.splits*)
next
  case (*Abs xs x X*)
  show *?case* **proof** *safe*
    assume *q*: *qFreshAbs zs z* (*qAbs xs x X*)
    qGoodAbs (*qAbs xs x X*) *qFreshEnv zs z rho*
    let *?x'* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*} **let** *?X'* = *X* #[[*?x'* ∧ *x*]]-*xs*
    have *x'*: *qFresh xs ?x' X* ∧ *qFreshEnv xs ?x' rho*
    using *q GOOD-ENV* **by**(*auto simp add*: *pickQFreshEnv*)
    hence *goodX'*: *qGood ?X'* **using** *q qSwap-preserves-qGood* **by** *auto*
    have *XX'*: (*X*,*?X'*) ∈ *qSwapped* **using** *qSwap-qSwapped* **by** *fastforce*
    have (*zs* = *xs* ∧ *z* = *?x'*) ∨ *qFresh zs z* (*qPsubst rho ?X'*)
    by (*meson qSwap-preserves-qFresh-distinct*
    Abs.IH(*1*) *XX' goodX' q qAbs-alphaAbs-qSwap-qFresh qFreshAbs.simps*
    qFreshAbs-preserves-alphaAbs1 qSwap-preserves-qGood2 x')
    thus *qFreshAbs zs z* ((*qAbs xs x X*) $[[*rho*]])
    by *simp* (*meson qFreshAbs.simps*)+
  qed
qed

lemma *qPsubst-preserves-qFresh*:
⟦*qGood X*; *qGoodEnv rho*; *qFresh zs z X*; *qFreshEnv zs z rho*⟧
 ⟹ *qFresh zs z* (*X* #[[*rho*]])
by(*simp add*: *qPsubstAll-preserves-qFreshAll*)

lemma *qPsubstAbs-preserves-qFreshAbs*:
⟦*qGoodAbs A*; *qGoodEnv rho*; *qFreshAbs zs z A*; *qFreshEnv zs z rho*⟧
 ⟹ *qFreshAbs zs z* (*A* $[[*rho*]])
by(*simp add*: *qPsubstAll-preserves-qFreshAll*)

While in general we try to avoid proving facts in parallel, here we seem to have no choice – it is the first time we must use mutual induction:

lemma *qPsubstAll-preserves-alphaAll-qSwapAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* **and**
      A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* **and**
      rho::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qEnv*
assumes *goodRho*: *qGoodEnv rho*
shows
(*qGood X* ⟶
  (∀ *Y*. *X* #= *Y* ⟶ (*X* #[[*rho*]]) #= (*Y* #[[*rho*]])) ∧
  (∀ *xs z1 z2*. *qFreshEnv xs z1 rho* ∧ *qFreshEnv xs z2 rho* ⟶
            ((*X* #[[*z1* ∧ *z2*]]-*xs*) #[[*rho*]]) #= ((*X* #[[*rho*]]) #[[*z1* ∧ *z2*]]-*xs*))) ∧

```
(qGoodAbs A ⟶
  (∀ B. A $= B ⟶ (A $[[rho]]) $= (B $[[rho]])) ∧
  (∀ xs z1 z2. qFreshEnv xs z1 rho ∧ qFreshEnv xs z2 rho ⟶
              ((A $[[z1 ∧ z2]]-xs) $[[rho]]) $= ((A $[[rho]]) $[[z1 ∧ z2]]-xs)))
proof(induction rule: qGood-qTerm-induct-mutual)
  case (Var1 xs x)
  then show ?case
 by (metis alpha-refl goodRho qGood.simps(1) qPsubst-preserves-qGood qVar-alpha-iff)
next
  case (Var2 xs x)
  show ?case proof safe
    fix s::'sort and zs z1 z2
    assume FreshEnv: qFreshEnv zs z1 rho qFreshEnv zs z2 rho
    hence n: rho zs z1 = None ∧ rho zs z2 = None unfolding qFreshEnv-def by
simp
    let ?Left = qPsubst rho ((qVar xs x) #[[z1 ∧ z2]]-zs)
    let ?Right = (qPsubst rho (qVar xs x)) #[[z1 ∧ z2]]-zs
    have qGood (qVar xs x) by simp
    hence qGood ((qVar xs x) #[[z1 ∧ z2]]-zs)
    using qSwap-preserves-qGood by blast
    hence goodLeft: qGood ?Left using goodRho qPsubst-preserves-qGood by blast
    show ?Left #= ?Right
    proof(cases rho xs x)
      case None
      hence rho xs (x @xs[z1 ∧ z2]-zs) = None
      using n unfolding sw-def by auto
      thus ?thesis using None by simp
    next
      case (Some X)
      hence xs ≠ zs ∨ x ∉ {z1,z2} using n by auto
      hence (x @xs[z1 ∧ z2]-zs) = x unfolding sw-def by auto
      moreover
      {have qFresh zs z1 X ∧ qFresh zs z2 X
       using Some FreshEnv unfolding qFreshEnv-def liftAll-def by auto
         moreover have qGood X using Some goodRho unfolding qGoodEnv-def
liftAll-def by auto
       ultimately have X #= (X #[[z1 ∧ z2]]-zs)
       by(auto simp: alpha-qFresh-qSwap-id alpha-sym)
      }
      ultimately show ?thesis using Some by simp
    qed
  qed
next
  case (Op1 delta inp binp)
  show ?case proof safe
    fix Y assume q: qOp delta inp binp #= Y
    then obtain inp' binp' where Y: Y = qOp delta inp' binp' and
       *: (∀ i. (inp i = None) = (inp' i = None)) ∧
           (∀ i. (binp i = None) = (binp' i = None)) and
```

∗∗: (∀ i X X′. inp i = Some X ∧ inp′ i = Some X′ ⟶ X #= X′) ∧
(∀ i A A′. binp i = Some A ∧ binp′ i = Some A′ ⟶ A $= A′)
unfolding *qOp-alpha-iff sameDom-def liftAll2-def* **by** *auto*
show (*qOp delta inp binp*) #[[*rho*]] #= (*Y* #[[*rho*]])
using *Op1* ∗∗
by (*simp add*: *Y sameDom-def liftAll2-def*)
(*fastforce simp add*: ∗ *lift-None lift-Some*
liftAll-def lift-def split: *option.splits*)
qed
next
case (*Op2 delta inp binp*)
thus *?case*
by (*auto simp*: *sameDom-def liftAll2-def lift-None lift-def liftAll-def split*: *option.splits*)
next
case (*Abs1 xs x X*)
show *?case* **proof** *safe*
fix *B*
assume *alpha-xXB*: *qAbs xs x X* $= *B*
then obtain *y Y* **where** *B*: *B* = *qAbs xs y Y* **unfolding** *qAbs-alphaAbs-iff* **by** *auto*
have *qGoodAbs B* **using** ‹*qGood X*› *alpha-xXB alphaAbs-preserves-qGoodAbs* **by** *force*
hence *goodY*: *qGood Y* **unfolding** *B* **by** *simp*
let *?x′* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*}
let *?y′* = *pickQFreshEnv xs* {*y*} {*Y*} {*rho*}
obtain *x′* **and** *y′* **where** *x′y′-def*: *x′* = *?x′ y′* = *?y′* **and**
x′y′-rev: *?x′* = *x′ ?y′* = *y′* **by** *blast*
have *x′y′-freshXY*: *qFresh xs x′ X* ∧ *qFresh xs y′ Y*
unfolding *x′y′-def* **using** ‹*qGood X*› *goodY goodRho* **by** (*auto simp add*: *pickQFreshEnv*)
have *x′y′-fresh-rho*: *qFreshEnv xs x′ rho* ∧ *qFreshEnv xs y′ rho*
unfolding *x′y′-def* **using** ‹*qGood X*› *goodY goodRho* **by** (*auto simp add*: *pickQFreshEnv*)
have *x′y′-not-xy*: *x′* ≠ *x* ∧ *y′* ≠ *y*
unfolding *x′y′-def* **using** ‹*qGood X*› *goodY goodRho*
using *pickQFreshEnv*[*of* {*x*} {*X*}] *pickQFreshEnv*[*of* {*y*} {*Y*}] **by** *force*
have *goodXx′x*: *qGood* (*X* #[[*x′* ∧ *x*]]-*xs*) **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *good*: *qGood*(*qPsubst rho* (*X* #[[*x′* ∧ *x*]]-*xs*))
using *goodRho qPsubst-preserves-qGood* **by** *auto*
have *goodYy′y*: *qGood* (*Y* #[[*y′* ∧ *y*]]-*xs*) **using** *goodY qSwap-preserves-qGood* **by** *auto*
obtain *z* **where** *z-not*: *z* ∉ {*x*,*y*,*x′*,*y′*} **and**
z-fresh-XY: *qFresh xs z X* ∧ *qFresh xs z Y*
and *z-fresh-rho*: *qFreshEnv xs z rho* **using** ‹*qGood X*› *goodY goodRho*
using *obtain-qFreshEnv*[*of* {*x*,*y*,*x′*,*y′*} {*X*, *Y*} {*rho*}] **by** *auto*

let *?Xx′x* = *X* #[[*x′* ∧ *x*]]-*xs* **let** *?Yy′y* = *Y* #[[*y′* ∧ *y*]]-*xs*

let *?Xx'xzx'* = *?Xx'x* #[[*z* ∧ *x'*]]-*xs* **let** *?Yy'yzy'* = *?Yy'y* #[[*z* ∧ *y'*]]-*xs*
let *?Xzx* = *X* #[[*z* ∧ *x*]]-*xs* **let** *?Yzy* = *Y* #[[*z* ∧ *y*]]-*xs*

have *goodXx'x*: *qGood ?Xx'x* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodXx'xzx'*: *qGood ?Xx'xzx'* **using** *qSwap-preserves-qGood* **by** *auto*
have *qGood* (*?Xx'x* #[[*rho*]]) **using** *goodXx'x goodRho qPsubst-preserves-qGood*
by *auto*
hence *goodXx'x-rho-zx'*: *qGood* ((*?Xx'x* #[[*rho*]]) #[[*z* ∧ *x'*]]-*xs*)
using *qSwap-preserves-qGood* **by** *auto*
have *goodYy'y*: *qGood ?Yy'y* **using** *goodY qSwap-preserves-qGood* **by** *auto*

have *skelXx'x*: *qSkel ?Xx'x* = *qSkel X* **using** *qSkel-qSwap* **by** *fastforce*
hence *skelXx'xzx'*: *qSkel ?Xx'xzx'* = *qSkel X* **by** (*auto simp add*: *qSkel-qSwap*)
have *qSkelAbs B* = *qSkelAbs* (*qAbs xs x X*)
using *alpha-xXB alphaAll-qSkelAll* **by** *fastforce*
hence *qSkel Y* = *qSkel X* **unfolding** *B* **by**(*auto simp add*: *fun-eq-iff*)
hence *skelYy'y*: *qSkel ?Yy'y* = *qSkel X* **by**(*auto simp add*: *qSkel-qSwap*)

have ((*?Xx'x* #[[*rho*]]) #[[*z* ∧ *x'*]]-*xs*) #= (*?Xx'xzx'* #[[*rho*]])
using *skelXx'x goodXx'x z-fresh-rho x'y'-fresh-rho*
Abs1.IH(*2*)[*of ?Xx'x*] **by** (*auto simp add*: *alpha-sym*)
moreover
{**have** *?Xx'xzx'* #= *?Xzx*
using ‹*qGood X*› *x'y'-freshXY z-fresh-XY alpha-qFresh-qSwap-compose* **by**
fastforce
moreover have *?Xzx* #= *?Yzy* **using** *alpha-xXB* **unfolding** *B*
using *z-fresh-XY* ‹*qGood X*› *goodY*
by (*simp only*: *alphaAbs-qAbs-iff-all-qFresh*)
moreover have *?Yzy* #= *?Yy'yzy'* **using** *goodY x'y'-freshXY z-fresh-XY*
by(*auto simp add*: *alpha-qFresh-qSwap-compose alpha-sym*)
ultimately have *?Xx'xzx'* #= *?Yy'yzy'* **using** *goodXx'xzx' alpha-trans* **by**
blast
hence (*?Xx'xzx'* #[[*rho*]]) #= (*?Yy'yzy'* #[[*rho*]])
using *goodXx'xzx' skelXx'xzx' Abs1.IH*(*1*) **by** *auto*
}
moreover have (*?Yy'yzy'* #[[*rho*]]) #= ((*?Yy'y* #[[*rho*]]) #[[*z* ∧ *y'*]]-*xs*)
using *skelYy'y goodYy'y z-fresh-rho x'y'-fresh-rho*
Abs1.IH(*2*)[*of ?Yy'y*] *alpha-sym* **by** *fastforce*
ultimately
have ((*?Xx'x* #[[*rho*]]) #[[*z* ∧ *x'*]]-*xs*) #= ((*?Yy'y* #[[*rho*]]) #[[*z* ∧ *y'*]]-*xs*)
using *goodXx'x-rho-zx' alpha-trans* **by** *blast*
thus (*qAbs xs x X*) $[[*rho*]] $= (*B* $[[*rho*]])
unfolding *B* **apply** *simp* **unfolding** *Let-def*
unfolding *x'y'-rev*
using *good z-not* **apply**(*simp only*: *alphaAbs-qAbs-iff-ex-qFresh*)
by (*auto intro*!: *exI*[*of - z*]
simp: *alphaAbs-qAbs-iff-ex-qFresh goodRho goodXx'x qPsubstAll-preserves-qFreshAll*

qSwap-preserves-qFresh-distinct z-fresh-XY goodYy'y qPsubst-preserves-qFresh

z-fresh-rho)
qed
next
case (*Abs2 xs x X*)
show *?case* **proof** *safe*
fix *zs z1 z2*
assume *z1z2-fresh-rho*: *qFreshEnv zs z1 rho qFreshEnv zs z2 rho*
let *?x'* = *pickQFreshEnv xs* {*x* @*xs*[*z1* ∧ *z2*]-*zs*} {*X* #[[*z1* ∧ *z2*]]-*zs*} {*rho*}
let *?x''* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*}
obtain *x' x''* **where** *x'x''-def*: *x'* = *?x' x''* = *?x''* **and**
x'x''-rev: *?x'* = *x' ?x''* = *x''* **by** *blast*
let *?xa* = *x* @*xs*[*z1* ∧ *z2*]-*zs* **let** *?xa''* = *x''* @*xs*[*z1* ∧ *z2*]-*zs*
obtain *u* **where** *u* ∉ {*x*,*x'*,*x''*,*z1*,*z2*} **and**
u-fresh-X: *qFresh xs u X* **and** *u-fresh-rho*: *qFreshEnv xs u rho*
using ‹*qGood X*› *goodRho* **using** *obtain-qFreshEnv*[*of* {*x*,*x'*,*x''*,*z1*,*z2*} {*X*} {*rho*}] **by** *auto*
hence *u-not*: *u* ∉ {*x*,*x'*,*x''*,*z1*,*z2*,*?xa*,*?xa''*} **unfolding** *sw-def* **by** *auto*
let *?ua* = *u* @*xs* [*z1* ∧ *z2*]-*zs*
let *?Xz1z2* = *X* #[[*z1* ∧ *z2*]]-*zs*
let *?Xz1z2x'xa* = *?Xz1z2* #[[*x'* ∧ *?xa*]]-*xs*
let *?Xz1z2x'xa-rho* = *?Xz1z2x'xa* #[[*rho*]]
let *?Xz1z2x'xa-rho-ux'* = *?Xz1z2x'xa-rho* #[[*u* ∧ *x'*]]-*xs*
let *?Xz1z2x'xaux'* = *?Xz1z2x'xa* #[[*u* ∧ *x'*]]-*xs*
let *?Xz1z2x'xaux'-rho* = *?Xz1z2x'xaux'* #[[*rho*]]
let *?Xz1z2uxa* = *?Xz1z2* #[[*u* ∧ *?xa*]]-*xs*
let *?Xz1z2uaxa* = *?Xz1z2* #[[*?ua* ∧ *?xa*]]-*xs*
let *?Xux* = *X* #[[*u* ∧ *x*]]-*xs*
let *?Xuxz1z2* = *?Xux* #[[*z1* ∧ *z2*]]-*zs*
let *?Xx''x* = *X* #[[*x''* ∧ *x*]]-*xs*
let *?Xx''xux''* = *?Xx''x* #[[*u* ∧ *x''*]]-*xs*
let *?Xx''xux''z1z2* = *?Xx''xux''* #[[*z1* ∧ *z2*]]-*zs*
let *?Xx''xz1z2* = *?Xx''x* #[[*z1* ∧ *z2*]]-*zs*
let *?Xx''xz1z2uaxa''* = *?Xx''xz1z2* #[[*?ua* ∧ *?xa''*]]-*xs*
let *?Xx''xz1z2uaxa''-rho* = *?Xx''xz1z2uaxa''* #[[*rho*]]
let *?Xx''xz1z2uxa''* = *?Xx''xz1z2* #[[*u* ∧ *?xa''*]]-*xs*
let *?Xx''xz1z2uxa''-rho* = *?Xx''xz1z2uxa''* #[[*rho*]]
let *?Xx''xz1z2-rho* = *?Xx''xz1z2* #[[*rho*]]
let *?Xx''xz1z2-rho-uxa''* = *?Xx''xz1z2-rho* #[[*u* ∧ *?xa''*]]-*xs*
let *?Xx''x-rho* = *?Xx''x* #[[*rho*]]
let *?Xx''x-rho-z1z2* = *?Xx''x-rho* #[[*z1* ∧ *z2*]]-*zs*
let *?Xx''x-rho-z1z2uxa''* = *?Xx''x-rho-z1z2* #[[*u* ∧ *?xa''*]]-*xs*

have *goodXz1z2*: *qGood ?Xz1z2* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
have *x'x''-fresh-Xz1z2*: *qFresh xs x' ?Xz1z2* ∧ *qFresh xs x'' X*
unfolding *x'x''-def* **using** ‹*qGood X*› *goodXz1z2 goodRho* **by** (*auto simp add*: *pickQFreshEnv*)
have *x'x''-fresh-rho*: *qFreshEnv xs x' rho* ∧ *qFreshEnv xs x'' rho*
unfolding *x'x''-def* **using** ‹*qGood X*› *goodXz1z2 goodRho* **by** (*auto simp add*:

pickQFreshEnv)
have *ua-eq-u*: *?ua* = *u* **using** *u-not* **unfolding** *sw-def* **by** *auto*

have *goodXz1z2x′xa*: *qGood ?Xz1z2x′xa* **using** *goodXz1z2 qSwap-preserves-qGood* **by** *auto*
have *goodXux*: *qGood ?Xux* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodXuxz1z2*: *qGood ?Xuxz1z2* **using** *qSwap-preserves-qGood* **by** *auto*
have *goodXx′′x*: *qGood ?Xx′′x* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodXx′′xz1z2*: *qGood ?Xx′′xz1z2* **using** *qSwap-preserves-qGood* **by** *auto*
hence *qGood ?Xx′′xz1z2-rho* **using** *goodRho qPsubst-preserves-qGood* **by** *auto*
hence *goodXx′′xz1z2-rho*: *qGood ?Xx′′xz1z2-rho*
using *goodRho qPsubst-preserves-qGood* **by** *auto*
have *goodXz1z2x′xaux′*: *qGood ?Xz1z2x′xaux′*
using *goodXz1z2x′xa qSwap-preserves-qGood* **by** *auto*
have *goodXz1z2x′xa-rho*: *qGood ?Xz1z2x′xa-rho*
using *goodXz1z2x′xa goodRho qPsubst-preserves-qGood* **by** *auto*
hence *goodXz1z2x′xa-rho-ux′*: *qGood ?Xz1z2x′xa-rho-ux′*
using *qSwap-preserves-qGood* **by** *auto*

have *xa′′-fresh-rho*: *qFreshEnv xs ?xa′′ rho*
using *x′x′′-fresh-rho z1z2-fresh-rho* **unfolding** *sw-def* **by** *auto*
have *u-fresh-Xz1z2*: *qFresh xs u ?Xz1z2*
using *u-fresh-X u-not* **by**(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *qFresh xs u ?Xz1z2x′xa* **using** *u-not* **by**(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *u-fresh-Xz1z2x′xa-rho*: *qFresh xs u ?Xz1z2x′xa-rho*
using *u-fresh-rho u-fresh-X goodRho goodXz1z2x′xa qPsubst-preserves-qFresh* **by** *auto*
have *qFresh xs u ?Xx′′x*
using *u-fresh-X u-not* **by**(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *qFresh xs u ?Xx′′x-rho* **using** *goodRho goodXx′′x u-fresh-rho*
by(*auto simp add*: *qPsubst-preserves-qFresh*)
hence *u-fresh-Xx′′x-rho-z1z2*: *qFresh xs u ?Xx′′x-rho-z1z2*
using *u-not* **by**(*auto simp add*: *qSwap-preserves-qFresh-distinct*)

have *skel-Xz1z2x′xa*: *qSkel ?Xz1z2x′xa* = *qSkel X* **by**(*auto simp add*: *qSkel-qSwap*)
hence *skel-Xz1z2x′xaux′*: *qSkel ?Xz1z2x′xaux′* = *qSkel X* **by**(*auto simp add*: *qSkel-qSwap*)
have *skel-Xx′′x*: *qSkel ?Xx′′x* = *qSkel X* **by**(*auto simp add*: *qSkel-qSwap*)
hence *skel-Xx′′xz1z2*: *qSkel ?Xx′′xz1z2* = *qSkel X* **by**(*auto simp add*: *qSkel-qSwap*)

have *?Xz1z2x′xaux′-rho #= ?Xz1z2x′xa-rho-ux′*
using *x′x′′-fresh-rho u-fresh-rho skel-Xz1z2x′xa goodXz1z2x′xa*
using *Abs2.IH*(*2*)[*of ?Xz1z2x′xa*] **by** *auto*
hence *?Xz1z2x′xa-rho-ux′ #= ?Xz1z2x′xaux′-rho* **using** *alpha-sym* **by** *auto*
moreover
{**have** *?Xz1z2x′xaux′ #= ?Xz1z2uxa*
using *goodXz1z2 u-fresh-Xz1z2 x′x′′-fresh-Xz1z2*
using *alpha-qFresh-qSwap-compose* **by** *fastforce*

```
    moreover have ?Xz1z2uxa = ?Xuxz1z2
    using ua-eq-u qSwap-compose[of zs z1 z2 xs x u X] by(auto simp: qSwap-sym)
    moreover
    {have ?Xux #= ?Xx''xux''
     using ‹qGood X› u-fresh-X x'x''-fresh-Xz1z2
     by(auto simp: alpha-qFresh-qSwap-compose alpha-sym)
     hence ?Xuxz1z2 #= ?Xx''xux''z1z2
     using goodXux by (auto simp add: qSwap-preserves-alpha)
    }
    moreover have ?Xx''xux''z1z2 = ?Xx''xz1z2uxa''
    using ua-eq-u qSwap-compose[of zs z1 z2 - - - ?Xx''x] by auto
    ultimately have ?Xz1z2x'xaux' #= ?Xx''xz1z2uxa''
    using goodXz1z2x'xaux' alpha-trans by auto
    hence ?Xz1z2x'xaux'-rho #= ?Xx''xz1z2uxa''-rho
    using goodXz1z2x'xaux' skel-Xz1z2x'xaux' Abs2.IH(1) by auto
   }
   moreover have ?Xx''xz1z2uxa''-rho #= ?Xx''xz1z2-rho-uxa''
   using xa''-fresh-rho u-fresh-rho skel-Xx''xz1z2 goodXx''xz1z2
   using Abs2.IH(2)[of ?Xx''xz1z2] by auto
   moreover
   {have ?Xx''xz1z2-rho #= ?Xx''x-rho-z1z2
    using z1z2-fresh-rho skel-Xx''x goodXx''x
    using Abs2.IH(2)[of ?Xx''x] by auto
    hence ?Xx''xz1z2-rho-uxa'' #= ?Xx''x-rho-z1z2uxa''
    using goodXx''xz1z2-rho by(auto simp add: qSwap-preserves-alpha)
   }
   ultimately have ?Xz1z2x'xa-rho-ux' #= ?Xx''x-rho-z1z2uxa''
   using goodXz1z2x'xa-rho-ux' alpha-trans by blast
   thus ((qAbs xs x X) $[[z1 ∧ z2]]-zs) $[[rho]] $=
        (((qAbs xs x X) $[[rho]]) $[[z1 ∧ z2]]-zs)
   using goodXz1z2x'xa-rho
   goodXz1z2x'xa u-not u-fresh-Xz1z2x'xa-rho u-fresh-Xx''x-rho-z1z2
   apply(simp add: Let-def x'x''-rev del: alpha.simps alphaAbs.simps )
   by (auto simp only: Let-def alphaAbs-qAbs-iff-ex-qFresh)
  qed
qed

corollary qPsubst-preserves-alpha1:
assumes qGoodEnv rho and qGood X ∨ qGood Y and X #= Y
shows (X #[[rho]]) #= (Y #[[rho]])
using alpha-preserves-qGood assms qPsubstAll-preserves-alphaAll-qSwapAll by blast

corollary qPsubstAbs-preserves-alphaAbs1:
assumes qGoodEnv rho and qGoodAbs A ∨ qGoodAbs B and A $= B
shows (A $[[rho]]) $= (B $[[rho]])
using alphaAbs-preserves-qGoodAbs assms qPsubstAll-preserves-alphaAll-qSwapAll
by blast

corollary alpha-qFreshEnv-qSwap-qPsubst-commute:
```

⟦*qGoodEnv rho*; *qGood X*; *qFreshEnv zs z1 rho*; *qFreshEnv zs z2 rho*⟧ ⟹
((*X* #[[*z1* ∧ *z2*]]-*zs*) #[[*rho*]]) #= ((*X* #[[*rho*]]) #[[*z1* ∧ *z2*]]-*zs*)
by(*simp add*: *qPsubstAll-preserves-alphaAll-qSwapAll*)

corollary *alphaAbs-qFreshEnv-qSwapAbs-qPsubstAbs-commute*:
⟦*qGoodEnv rho*; *qGoodAbs A*;
qFreshEnv zs z1 rho; *qFreshEnv zs z2 rho*⟧ ⟹
((*A* $[[*z1* ∧ *z2*]]-*zs*) $[[*rho*]]) $= ((*A* $[[*rho*]]) $[[*z1* ∧ *z2*]]-*zs*)
by(*simp add*: *qPsubstAll-preserves-alphaAll-qSwapAll*)

lemma *qPsubstAll-preserves-alphaAll2*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and**
rho'::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qEnv* **and** *rho''*
assumes *rho'-alpha-rho''*: *rho'* &= *rho''* **and**
goodRho': *qGoodEnv rho'* **and** *goodRho''*: *qGoodEnv rho''*
shows
(*qGood X* ⟶ (*X* #[[*rho'*]]) #= (*X* #[[*rho''*]])) ∧
(*qGoodAbs A* ⟶ (*A* $[[*rho'*]]) $= (*A* $[[*rho''*]]))
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
then show *?case*
proof (*cases rho' xs x*)
case *None*
hence *rho'' xs x* = *None* **using** *rho'-alpha-rho''* **unfolding** *alphaEnv-def sameDom-def* **by** *auto*
thus *?thesis* **using** *None* **by** *simp*
next
case (*Some X'*)
then obtain *X''* **where** *rho''*: *rho'' xs x* = *Some X''*
using *assms* **unfolding** *alphaEnv-def sameDom-def* **by** *force*
hence *X'* #= *X''* **using** *Some rho'-alpha-rho''*
unfolding *alphaEnv-def liftAll2-def* **by** *auto*
thus *?thesis* **using** *Some rho''* **by** *simp*
qed
next
case (*Op delta inp binp*)
then show *?case*
by (*auto simp*: *lift-def liftAll-def liftAll2-def sameDom-def Let-def*
split: *option.splits*)
next
case (*Abs xs x X*)
let *?x'* = *pickQFreshEnv xs* {*x*} {*X*} {*rho'*}
let *?x''* = *pickQFreshEnv xs* {*x*} {*X*} {*rho''*}
obtain *x' x''* **where** *x'x''-def*: *x'* = *?x' x''* = *?x''* **and**
x'x''-rev: *?x'* = *x' ?x''* = *x''* **by** *blast*
have *x'x''-fresh-X*: *qFresh xs x' X* ∧ *qFresh xs x'' X*
unfolding *x'x''-def* **using** ‹*qGood X*› *goodRho' goodRho''* **by** (*auto simp add*: *pickQFreshEnv*)

have *x′-fresh-rho′*: *qFreshEnv xs x′ rho′*
unfolding *x′x′′-def* **using** ‹*qGood X*› *goodRho′ goodRho′′* **by** (*auto simp add*: *pickQFreshEnv*)
have *x′′-fresh-rho′′*: *qFreshEnv xs x′′ rho′′*
unfolding *x′x′′-def* **using** ‹*qGood X*› *goodRho′ goodRho′′* **by** (*auto simp add*: *pickQFreshEnv*)
obtain *u* **where** *u-not*: $u \notin \{x,x',x''\}$ **and**
u-fresh-X: *qFresh xs u X* **and**
u-fresh-rho′: *qFreshEnv xs u rho′* **and** *u-fresh-rho′′*: *qFreshEnv xs u rho′′*
using ‹*qGood X*› *goodRho′ goodRho′′*
using *obtain-qFreshEnv*[*of* {*x*,*x′*,*x′′*} {*X*} {*rho′*,*rho′′*}] **by** *auto*

let *?Xx′x* = *X* #[[*x′* ∧ *x*]]-*xs*
let *?Xx′x-rho′* = *?Xx′x* #[[*rho′*]]
let *?Xx′x-rho′-ux′* = *?Xx′x-rho′* #[[*u* ∧ *x′*]]-*xs*
let *?Xx′xux′* = *?Xx′x* #[[*u* ∧ *x′*]]-*xs*
let *?Xx′xux′-rho′* = *?Xx′xux′* #[[*rho′*]]
let *?Xux* = *X* #[[*u* ∧ *x*]]-*xs*
let *?Xux-rho′* = *?Xux* #[[*rho′*]]
let *?Xux-rho′′* = *?Xux* #[[*rho′′*]]
let *?Xx′′x* = *X* #[[*x′′* ∧ *x*]]-*xs*
let *?Xx′′xux′′* = *?Xx′′x* #[[*u* ∧ *x′′*]]-*xs*
let *?Xx′′xux′′-rho′′* = *?Xx′′xux′′* #[[*rho′′*]]
let *?Xx′′x-rho′′* = *?Xx′′x* #[[*rho′′*]]
let *?Xx′′x-rho′′-ux′′* = *?Xx′′x-rho′′* #[[*u* ∧ *x′′*]]-*xs*

have *goodXx′x*: *qGood ?Xx′x* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodXx′x-rho′*: *qGood ?Xx′x-rho′* **using** ‹*qGood X*› *goodRho′ qPsubst-preserves-qGood* **by** *auto*
hence *goodXx′x-rho′-ux′*: *qGood ?Xx′x-rho′-ux′*
using ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
have *goodXx′xux′*: *qGood ?Xx′xux′* **using** *goodXx′x qSwap-preserves-qGood* **by** *auto*
have *goodXux*: *qGood ?Xux* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
have *goodXx′′x*: *qGood ?Xx′′x* **using** ‹*qGood X*› *qSwap-preserves-qGood* **by** *auto*
hence *goodXx′′x-rho′′*: *qGood ?Xx′′x-rho′′*
using ‹*qGood X*› *goodRho′′ qPsubst-preserves-qGood* **by** *auto*

have *qFresh xs u ?Xx′x* **using** *u-not u-fresh-X*
by(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *fresh-Xx′x-rho′*: *qFresh xs u ?Xx′x-rho′*
using *u-fresh-rho′ goodXx′x goodRho′* **by**(*auto simp add*: *qPsubst-preserves-qFresh*)
have *qFresh xs u ?Xx′′x* **using** *u-not u-fresh-X*
by(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *fresh-Xx′′x-rho′′*: *qFresh xs u ?Xx′′x-rho′′*
using *u-fresh-rho′′ goodXx′′x goodRho′′* **by**(*auto simp add*: *qPsubst-preserves-qFresh*)

have *Xux*: (*X*,*?Xux*) :*qSwapped* **by**(*simp add*: *qSwap-qSwapped*)

have *?Xx′x-rho′-ux′* #= *?Xx′xux′-rho′*
using *goodRho′ goodXx′x u-fresh-rho′ x′-fresh-rho′*
by(*auto simp*: *alpha-qFreshEnv-qSwap-qPsubst-commute alpha-sym*)
moreover
{**have** *?Xx′xux′* #= *?Xux* **using** ‹*qGood X*› *u-fresh-X x′x′′-fresh-X*
using *alpha-qFresh-qSwap-compose* **by** *fastforce*
hence *?Xx′xux′-rho′* #= *?Xux-rho′* **using** *goodXx′xux′ goodRho′*
using *qPsubst-preserves-alpha1* **by** *auto*
}
moreover have *?Xux-rho′* #= *?Xux-rho′′* **using** *Xux Abs.IH* **by** *auto*
moreover
{**have** *?Xux* #= *?Xx′′xux′′* **using** ‹*qGood X*› *u-fresh-X x′x′′-fresh-X*
by(*auto simp add*: *alpha-qFresh-qSwap-compose alpha-sym*)
hence *?Xux-rho′′* #= *?Xx′′xux′′-rho′′* **using** *goodXux goodRho′′*
using *qPsubst-preserves-alpha1* **by** *auto*
}
moreover have *?Xx′′xux′′-rho′′* #= *?Xx′′x-rho′′-ux′′*
using *goodRho′′ goodXx′′x u-fresh-rho′′ x′′-fresh-rho′′*
by(*auto simp*: *alpha-qFreshEnv-qSwap-qPsubst-commute*)
ultimately have *?Xx′x-rho′-ux′* #= *?Xx′′x-rho′′-ux′′*
using *goodXx′x-rho′-ux′ alpha-trans* **by** *blast*
hence *qAbs xs ?x′* (*qPsubst rho′* (*X* #[[*?x′* ∧ *x*]]-*xs*)) $=
qAbs xs ?x′′(*qPsubst rho′′*(*X* #[[*?x′′*∧ *x*]]-*xs*))
unfolding *x′x′′-rev* **using** *goodXx′x-rho′ fresh-Xx′x-rho′ fresh-Xx′′x-rho′′*
by (*auto simp only*: *alphaAbs-qAbs-iff-ex-qFresh*)
thus *?case* **by** (*metis qPsubstAbs.simps*)
qed

corollary *qPsubst-preserves-alpha2*:
⟦*qGood X*; *qGoodEnv rho′*; *qGoodEnv rho′′*; *rho′* &= *rho′′*⟧
⟹ (*X* #[[*rho′*]]) #= (*X* #[[*rho′′*]])
by(*simp add*: *qPsubstAll-preserves-alphaAll2*)

corollary *qPsubstAbs-preserves-alphaAbs2*:
⟦*qGoodAbs A*; *qGoodEnv rho′*; *qGoodEnv rho′′*; *rho′* &= *rho′′*⟧
⟹ (*A* $[[*rho′*]]) $= (*A* $[[*rho′′*]])
by(*simp add*: *qPsubstAll-preserves-alphaAll2*)

lemma *qPsubst-preserves-alpha*:
assumes *qGood X* ∨ *qGood X′* **and** *qGoodEnv rho* **and** *qGoodEnv rho′*
and *X* #= *X′* **and** *rho* &= *rho′*
shows (*X* #[[*rho*]]) #= (*X′* #[[*rho′*]])
by (*metis* (*no-types*, *lifting*) *assms alpha-trans qPsubst-preserves-alpha1*
qPsubst-preserves-alpha2 qPsubst-preserves-qGood)

lemma *qPsubstAbs-preserves-alphaAbs*:
assumes *qGoodAbs A* ∨ *qGoodAbs A′* **and** *qGoodEnv rho* **and** *qGoodEnv rho′*
and *A* $= *A′* **and** *rho* &= *rho′*
shows (*A* $[[*rho*]]) $= (*A′* $[[*rho′*]])

using *assms*
by (*meson alphaAbs-trans qPsubstAbs-preserves-alphaAbs1*
qPsubstAbs-preserves-qGoodAbs qPsubstAll-preserves-alphaAll2)

lemma *qFresh-qPsubst-commute-qAbs*:
assumes *good-X*: *qGood X* **and** *good-rho*: *qGoodEnv rho* **and**
x-fresh-rho: *qFreshEnv xs x rho*
shows ((*qAbs xs x X*) $[[*rho*]]) $= *qAbs xs x* (*X* #[[*rho*]])
proof−

let *?x′* = *pickQFreshEnv xs* {*x*} {*X*} {*rho*}
obtain *x′* **where** *x′-def*: *x′* = *?x′* **and** *x′-rev*: *?x′* = *x′* **by** *blast*
have *x′-not*: *x′* ≠ *x* **unfolding** *x′-def*
using *assms pickQFreshEnv*[*of* {*x*} {*X*}] **by** *auto*
have *x′-fresh-X*: *qFresh xs x′ X* **unfolding** *x′-def*
using *assms pickQFreshEnv*[*of* {*x*} {*X*}] **by** *auto*
have *x′-fresh-rho*: *qFreshEnv xs x′ rho* **unfolding** *x′-def*
using *assms pickQFreshEnv*[*of* {*x*} {*X*}] **by** *auto*
obtain *u* **where** *u-not*: *u* ∉ {*x*,*x′*} **and**
u-fresh-X: *qFresh xs u X* **and** *u-fresh-rho*: *qFreshEnv xs u rho*
using *good-X good-rho obtain-qFreshEnv*[*of* {*x*,*x′*} {*X*} {*rho*}] **by** *auto*
let *?Xx′x* = *X* #[[*x′* ∧ *x*]]-*xs*
let *?Xx′x-rho* = *?Xx′x* #[[*rho*]]
let *?Xx′x-rho-ux′* = *?Xx′x-rho* #[[*u* ∧ *x′*]]-*xs*
let *?Xx′xux′* = *?Xx′x* #[[*u* ∧ *x′*]]-*xs*
let *?Xx′xux′-rho* = *?Xx′xux′* #[[*rho*]]
let *?Xux* = *X* #[[*u* ∧ *x*]]-*xs*
let *?Xux-rho* = *?Xux* #[[*rho*]]
let *?Xrho* = *X* #[[*rho*]]
let *?Xrho-ux* = *?Xrho* #[[*u* ∧ *x*]]-*xs*

have *good-Xx′x*: *qGood ?Xx′x* **using** *good-X qSwap-preserves-qGood* **by** *auto*
hence *good-Xx′x-rho*: *qGood ?Xx′x-rho* **using** *good-rho qPsubst-preserves-qGood*
by *auto*
hence *good-Xx′x-rho-ux′*: *qGood ?Xx′x-rho-ux′* **using** *qSwap-preserves-qGood* **by**
auto
have *good-Xx′xux′*: *qGood ?Xx′xux′* **using** *good-Xx′x qSwap-preserves-qGood* **by**
auto

have *u-fresh-Xx′x*: *qFresh xs u ?Xx′x*
using *u-fresh-X u-not* **by**(*auto simp add*: *qSwap-preserves-qFresh-distinct*)
hence *u-fresh-Xx′x-rho*: *qFresh xs u ?Xx′x-rho*
using *good-rho good-Xx′x u-fresh-rho* **by**(*auto simp add*: *qPsubst-preserves-qFresh*)
have *u-fresh-Xrho*: *qFresh xs u ?Xrho*
using *good-rho good-X u-fresh-X u-fresh-rho* **by**(*auto simp add*: *qPsubst-preserves-qFresh*)
−
have *?Xx′x-rho-ux′* #= *?Xx′xux′-rho*
using *good-Xx′x good-rho u-fresh-rho x′-fresh-rho*
using *alpha-qFreshEnv-qSwap-qPsubst-commute alpha-sym* **by** *blast*

moreover
{have *?Xx′xux′ #= ?Xux*
using *good-X u-fresh-X x′-fresh-X* **by** (*auto simp add*: *alpha-qFresh-qSwap-compose*)
hence *?Xx′xux′-rho #= ?Xux-rho*
using *good-Xx′xux′ good-rho qPsubst-preserves-alpha1* **by** *auto*
}
moreover have *?Xux-rho #= ?Xrho-ux*
using *good-X good-rho u-fresh-rho x-fresh-rho*
using *alpha-qFreshEnv-qSwap-qPsubst-commute* **by** *blast*
ultimately have *?Xx′x-rho-ux′ #= ?Xrho-ux*
using *good-Xx′x-rho-ux′ alpha-trans* **by** *blast*
thus *?thesis* **apply** (*simp add*: *Let-def del*: *alpha.simps alphaAbs.simps*)
unfolding *x′-rev* **using** *good-Xx′x-rho*
using *u-fresh-Xx′x-rho u-fresh-Xrho* **by** (*auto simp only*: *alphaAbs-qAbs-iff-ex-qFresh*)

qed

end

end
theory *Pick* **imports** *Main*
begin

definition *pick X* ≡ *SOME x. x* ∈ *X*

lemma *pick*[*simp*]: *x* ∈ *X* ⟹ *pick X* ∈ *X*
unfolding *pick-def* **by** (*metis someI-ex*)

lemma *pick-NE*[*simp*]: *X* ≠ {} ⟹ *pick X* ∈ *X* **by** *auto*

end

4 Some preliminaries on equivalence relations and quotients

theory *Equiv-Relation2* **imports** *Preliminaries Pick*
begin

Unary predicates vs. sets:

definition *S2P A* ≡ λ *x. x* ∈ *A*

lemma *S2P-app*[*simp*]: *S2P r x* ⟷ *x* ∈ *r*
unfolding *S2P-def* **by** *auto*

lemma *S2P-Collect*[*simp*]: *S2P* (*Collect* φ) = φ
apply(*rule ext*)+ **by** *simp*

lemma *Collect-S2P*[*simp*]: *Collect* (*S2P r*) = *r*
by (*metis Collect-mem-eq S2P-Collect*)

Binary predicates vs. relatipons:

definition *P2R* $\varphi \equiv \{(x,y).\ \varphi\ x\ y\}$
definition *R2P* $r \equiv \lambda\ x\ y.\ (x,y) \in r$

lemma *in-P2R*[*simp*]: $xy \in P2R\ \varphi \longleftrightarrow \varphi\ (fst\ xy)\ (snd\ xy)$
unfolding *P2R-def* **by** *auto*

lemma *in-P2R-pair*[*simp*]: $(x,y) \in P2R\ \varphi \longleftrightarrow \varphi\ x\ y$
by *simp*

lemma *R2P-app*[*simp*]: $R2P\ r\ x\ y \longleftrightarrow (x,y) \in r$
unfolding *R2P-def* **by** *auto*

lemma *R2P-P2R*[*simp*]: $R2P\ (P2R\ \varphi) = \varphi$
apply(*rule ext*)+ **by** *simp*

lemma *P2R-R2P*[*simp*]: *P2R* (*R2P r*) = *r*
using *Collect-mem-eq P2R-def R2P-P2R case-prod-curry* **by** *metis*

definition *reflP P* $\varphi \equiv (\forall\ x\ y.\ \varphi\ x\ y \lor \varphi\ y\ x \longrightarrow P\ x) \land (\forall\ x.\ P\ x \longrightarrow \varphi\ x\ x)$
definition *symP* $\varphi \equiv \forall\ x\ y.\ \varphi\ x\ y \longrightarrow \varphi\ y\ x$
definition *transP* **where** *transP* $\varphi \equiv \forall\ x\ y\ z.\ \varphi\ x\ y \land \varphi\ y\ z \longrightarrow \varphi\ x\ z$
definition *equivP A* $\varphi \equiv$ *reflP A* $\varphi \land$ *symP* $\varphi \land$ *transP* φ

lemma *refl-on-P2R*[*simp*]: *refl-on* (*Collect P*) (*P2R* φ) $\longleftrightarrow$ *reflP P* φ
unfolding *reflP-def refl-on-def* **by** *force*

lemma *reflP-R2P*[*simp*]: *reflP* (*S2P A*) (*R2P r*) $\longleftrightarrow$ *refl-on A r*
unfolding *reflP-def refl-on-def* **by** *auto*

lemma *sym-P2R*[*simp*]: *sym* (*P2R* φ) $\longleftrightarrow$ *symP* φ
unfolding *symP-def sym-def* **by** *auto*

lemma *symP-R2P*[*simp*]: *symP* (*R2P r*) $\longleftrightarrow$ *sym r*
unfolding *symP-def sym-def* **by** *auto*

lemma *trans-P2R*[*simp*]: *trans* (*P2R* φ) $\longleftrightarrow$ *transP* φ
unfolding *transP-def trans-def* **by** *auto*

lemma *transP-R2P*[*simp*]: *transP* (*R2P r*) $\longleftrightarrow$ *trans r*
unfolding *transP-def trans-def* **by** *auto*

lemma *equiv-P2R*[*simp*]: *equiv* (*Collect P*) (*P2R* φ) $\longleftrightarrow$ *equivP P* φ
unfolding *equivP-def equiv-def* **by** *auto*

lemma *equivP-R2P*[*simp*]: *equivP* (*S2P A*) (*R2P r*) $\longleftrightarrow$ *equiv A r*

unfolding *equivP-def equiv-def* **by** *auto*

lemma *in-P2R-Im-singl*[*simp*]: $y \in P2R\ \varphi\ ``\ \{x\} \longleftrightarrow \varphi\ x\ y$ **by** *simp*

definition *proj* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ set$ **where**
$proj\ \varphi\ x \equiv \{y.\ \varphi\ x\ y\}$

lemma *proj-P2R*: $proj\ \varphi\ x = P2R\ \varphi\ ``\ \{x\}$ **unfolding** *proj-def* **by** *auto*

lemma *proj-P2R-raw*: $proj\ \varphi = (\lambda\ x.\ P2R\ \varphi\ ``\ \{x\})$
apply(*rule ext*) **unfolding** *proj-P2R* **..**

definition *univ* :: $('a \Rightarrow 'b) \Rightarrow ('a\ set \Rightarrow 'b)$
where *univ f X* == *f* (*SOME x. x* ∈ *X*)

definition *quotientP* ::
$('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a\ set \Rightarrow bool)$ (**infixl** ‹'/'/'/› *90*)
where $P\ ///\ \varphi \equiv S2P\ ((Collect\ P)\ //\ (P2R\ \varphi))$

lemma *proj-preserves*:
$P\ x \Longrightarrow (P\ ///\ \varphi)\ (proj\ \varphi\ x)$
unfolding *proj-P2R quotientP-def*
by (*metis S2P-def mem-Collect-eq quotientI*)

lemma *proj-in-iff*:
assumes *equivP P* φ
shows $(P///\varphi)\ (proj\ \varphi\ x) \longleftrightarrow P\ x$
using *assms* **unfolding** *quotientP-def proj-def*
by (*metis* (*mono-tags*) *Collect-mem-eq Equiv-Relation2.proj-def*
Equiv-Relation2.proj-preserves S2P-Collect empty-Collect-eq equivP-def
equiv-P2R in-quotient-imp-non-empty quotientP-def reflP-def)

lemma *proj-iff*[*simp*]:
$[\![equivP\ P\ \varphi;\ P\ x;\ P\ y]\!] \Longrightarrow proj\ \varphi\ x = proj\ \varphi\ y \longleftrightarrow \varphi\ x\ y$
unfolding *proj-P2R*
by (*metis* (*full-types*) *equiv-P2R equiv-class-eq-iff equiv-class-self*
in-P2R-pair mem-Collect-eq proj-P2R proj-def)

lemma *in-proj*[*simp*]: $[\![equivP\ P\ \varphi;\ P\ x]\!] \Longrightarrow x \in proj\ \varphi\ x$
unfolding *proj-P2R equiv-def refl-on-def equiv-P2R*[*symmetric*]
by *auto*

lemma *proj-image*[*simp*]: $(proj\ \varphi)\ `\ (Collect\ P) = Collect\ (P///\varphi)$
unfolding *proj-P2R-raw quotientP-def quotient-def* **by** *auto*

lemma *in-quotientP-imp-non-empty*:
assumes *equivP P* φ **and** $(P///\varphi)\ X$
shows $X \neq \{\}$
by (*metis R2P-P2R S2P-Collect S2P-def assms equivP-R2P*

in-quotient-imp-non-empty quotientP-def)

lemma *in-quotientP-imp-in-rel*:
⟦*equivP P* φ; $(P///\varphi)$ X; $x \in X$; $y \in X$⟧ $\Longrightarrow$ φ x y
unfolding *equiv-P2R*[*symmetric*] *quotientP-def quotient-eq-iff*
by (*metis S2P-def in-P2R-pair quotient-eq-iff*)

lemma *in-quotientP-imp-closed*:
⟦*equivP P* φ; $(P///\varphi)$ X; $x \in X$; φ x y⟧ $\Longrightarrow$ $y \in X$
using *S2P-Collect S2P-def equivP-def proj-P2R-raw proj-def*
quotientE quotientP-def transP-def
by *metis*

lemma *in-quotientP-imp-subset*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows $X \subseteq$ *Collect P*
by (*metis* (*mono-tags*, *lifting*) *CollectI assms equivP-def in-quotientP-imp-in-rel reflP-def subsetI*)

lemma *equivP-pick-in*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows *pick* $X \in X$
by (*metis assms in-quotientP-imp-non-empty pick-NE*)

lemma *equivP-pick-preserves*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows *P* (*pick X*)
by (*metis assms equivP-pick-in in-quotientP-imp-subset mem-Collect-eq set-rev-mp*)

lemma *proj-pick*:
assumes φ: *equivP P* φ **and** *X*: $(P///\varphi)$ X
shows *proj* φ (*pick X*) $= X$
by (*smt* (*verit*) *proj-def Equiv-Relation2.proj-iff Equiv-Relation2.proj-image X*
φ *equivP-pick-in equivP-pick-preserves image-iff mem-Collect-eq*)

lemma *pick-proj*:
assumes *equivP P* φ **and** *P x*
shows φ (*pick* (*proj* φ *x*)) *x*
by (*metis assms equivP-def in-proj mem-Collect-eq pick proj-def symP-def*)

lemma *equivP-pick-iff*[*simp*]:
assumes φ: *equivP P* φ **and** *X*: $(P///\varphi)$ X **and** *Y*: $(P///\varphi)$ Y
shows φ (*pick X*) (*pick Y*) $\longleftrightarrow X = Y$
by (*metis Equiv-Relation2.proj-iff X Y* φ *equivP-pick-preserves proj-pick*)

lemma *equivP-pick-inj-on*:
assumes *equivP P* φ
shows *inj-on pick* (*Collect* $(P///\varphi)$)
using *assms* **unfolding** *inj-on-def*

by (*metis assms equivP-pick-iff mem-Collect-eq*)

definition *congruentP* **where**
congruentP φ *f* $\equiv \forall\ x\ y.\ \varphi\ x\ y \longrightarrow f\ x = f\ y$

abbreviation *RESPECTS-P* (**infixr** ‹*respectsP*› *80*) **where**
f respectsP r == *congruentP r f*

lemma *congruent-P2R*: *congruent* (*P2R* φ) *f* = *congruentP* φ *f*
unfolding *congruent-def congruentP-def* **by** *auto*

lemma *univ-commute*[*simp*]:
assumes *equivP P* φ **and** *f respectsP* φ **and** *P x*
shows (*univ f*) (*proj* φ *x*) = *f x*
unfolding *congruent-P2R*[*symmetric*]
by (*metis* (*full-types*) *assms pick-def congruentP-def pick-proj univ-def*)

lemma *univ-unique*:
assumes *equivP P* φ **and** *f respectsP* φ **and** $\bigwedge x.\ P\ x \Longrightarrow G\ (proj\ \varphi\ x) = f\ x$
shows $\forall\ X.\ (P///\varphi)\ X \longrightarrow G\ X = univ\ f\ X$
by (*metis assms equivP-pick-preserves proj-pick univ-commute*)

lemma *univ-preserves*:
assumes *equivP P* φ **and** *f respectsP* φ **and** $\bigwedge x.\ P\ x \Longrightarrow f\ x \in B$
shows $\forall\ X.\ (P///\varphi)\ X \longrightarrow univ\ f\ X \in B$
by (*metis Equiv-Relation2.univ-commute assms*
equivP-pick-preserves proj-pick)

end

5 Transition from Quasi-Terms to Terms

theory *Transition-QuasiTerms-Terms*
imports *QuasiTerms-Environments-Substitution Equiv-Relation2*
begin

This section transits from quasi-terms to terms: defines terms as alpha-equivalence classes of quasi-terms (and also abstractions as alpha-equivalence classes of quasi-abstractions), then defines operators on terms corresponding to those on quasi-terms: variable injection, binding operation, freshness, swapping, parallel substitution, etc. Properties previously shown invariant under alpha-equivalence, including induction principles, are lifted from quasi-terms. Moreover, a new powerful induction principle, allowing freshness assumptions, is proved for terms.

As a matter of notation: Starting from this section, we change the notations for quasi-item meta-variables, prefixing their names with a "q" – e.g., qX, qA, qinp, qenv, etc. The old names are now assigned to the “real" items: terms, abstractions, inputs, environments.

5.1 Preparation: Integrating quasi-inputs as first-class citizens

context *FixVars*
begin

From now on it will be convenient to also define fresh, swap, good and alpha-equivalence for quasi-inpus.

definition *qSwapInp* **where**
qSwapInp xs x y qinp == lift (qSwap xs x y) qinp

definition *qSwapBinp* **where**
qSwapBinp xs x y qbinp == lift (qSwapAbs xs x y) qbinp

abbreviation *qSwapInp-abbrev* (‹- %[[- ∧ -]]′--› *200*) **where**
(*qinp* %[[*z1* ∧ *z2*]]-*zs*) == *qSwapInp zs z1 z2 qinp*

abbreviation *qSwapBinp-abbrev* (‹- %%[[- ∧ -]]′--› *200*) **where**
(*qbinp* %%[[*z1* ∧ *z2*]]-*zs*) == *qSwapBinp zs z1 z2 qbinp*

lemma *qSwap-qSwapInp*:
((*qOp delta qinp qbinp*) #[[*x* ∧ *y*]]-*xs*) =
qOp delta (*qinp* %[[*x* ∧ *y*]]-*xs*) (*qbinp* %%[[*x* ∧ *y*]]-*xs*)
unfolding *qSwapInp-def qSwapBinp-def* **by** *simp*

declare *qSwap.simps*(*2*) [*simp del*]
declare *qSwap-qSwapInp*[*simp*]

lemmas *qSwapAll-simps* = *qSwap.simps*(*1*) *qSwap-qSwapInp*

definition *qPsubstInp* **where**
qPsubstInp qrho qinp == lift (qPsubst qrho) qinp

definition *qPsubstBinp* **where**
qPsubstBinp qrho qbinp == lift (qPsubstAbs qrho) qbinp

abbreviation *qPsubstInp-abbrev* (‹- %[[-]]› *200*)
where (*qinp* %[[*qrho*]]) == *qPsubstInp qrho qinp*

abbreviation *qPsubstBinp-abbrev* (‹- %%[[-]]› *200*)

where (*qbinp* %%[[*qrho*]]) == *qPsubstBinp qrho qbinp*

lemma *qPsubst-qPsubstInp*:
((*qOp delta qinp qbinp*) #[[*rho*]]) = *qOp delta* (*qinp* %[[*rho*]]) (*qbinp* %%[[*rho*]])
unfolding *qPsubstInp-def qPsubstBinp-def* **by** *simp*

declare *qPsubst.simps*(*2*) [*simp del*]
declare *qPsubst-qPsubstInp*[*simp*]

lemmas *qPsubstAll-simps* = *qPsubst.simps*(*1*) *qPsubst-qPsubstInp*

definition *qSkelInp*
where *qSkelInp qinp* = *lift qSkel qinp*

definition *qSkelBinp*
where *qSkelBinp qbinp* = *lift qSkelAbs qbinp*

lemma *qSkel-qSkelInp*:
qSkel (*qOp delta qinp qbinp*) =
Branch (*qSkelInp qinp*) (*qSkelBinp qbinp*)
unfolding *qSkelInp-def qSkelBinp-def* **by** *simp*

declare *qSkel.simps*(*2*) [*simp del*]
declare *qSkel-qSkelInp*[*simp*]

lemmas *qSkelAll-simps* = *qSkel.simps*(*1*) *qSkel-qSkelInp*

definition *qFreshInp* ::
'varSort ⇒ *'var* ⇒ (*'index*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*)*input* ⇒ *bool*
where
qFreshInp xs x qinp == *liftAll* (*qFresh xs x*) *qinp*

definition *qFreshBinp* ::
'varSort ⇒ *'var* ⇒ (*'bindex*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*)*input* ⇒ *bool*
where
qFreshBinp xs x qbinp == *liftAll* (*qFreshAbs xs x*) *qbinp*

lemma *qFresh-qFreshInp*:
qFresh xs x (*qOp delta qinp qbinp*) =
(*qFreshInp xs x qinp* ∧ *qFreshBinp xs x qbinp*)

unfolding *qFreshInp-def qFreshBinp-def* **by** *simp*

declare *qFresh.simps(2)* [*simp del*]
declare *qFresh-qFreshInp*[*simp*]

lemmas *qFreshAll-simps* = *qFresh.simps(1) qFresh-qFreshInp*

definition *qGoodInp* **where**
qGoodInp qinp ==
liftAll qGood qinp ∧
$|\{i.\ qinp\ i \neq None\}|$ <o |*UNIV* :: '*var set*|

definition *qGoodBinp* **where**
qGoodBinp qbinp ==
liftAll qGoodAbs qbinp ∧
$|\{i.\ qbinp\ i \neq None\}|$ <o |*UNIV* :: '*var set*|

lemma *qGood-qGoodInp*:
qGood (*qOp delta qinp qbinp*) = (*qGoodInp qinp* ∧ *qGoodBinp qbinp*)
unfolding *qGoodInp-def qGoodBinp-def* **by** *auto*

declare *qGood.simps(2)* [*simp del*]
declare *qGood-qGoodInp* [*simp*]

lemmas *qGoodAll-simps* = *qGood.simps(1) qGood-qGoodInp*

definition *alphaInp* **where**
alphaInp ==
{(*qinp*,*qinp*′). *sameDom qinp qinp*′ ∧ *liftAll2* (λ*qX qX*′. *qX* #= *qX*′) *qinp qinp*′}

definition *alphaBinp* **where**
alphaBinp ==
{(*qbinp*,*qbinp*′). *sameDom qbinp qbinp*′ ∧ *liftAll2* (λ*qA qA*′. *qA* $= *qA*′) *qbinp qbinp*′}

abbreviation *alphaInp-abbrev* (**infix** ‹%=› *50*) **where**
qinp %= *qinp*′ == (*qinp*,*qinp*′) ∈ *alphaInp*

abbreviation *alphaBinp-abbrev* (**infix** ‹%%=› *50*) **where**
qbinp %%= *qbinp*′ == (*qbinp*,*qbinp*′) ∈ *alphaBinp*

lemma *alpha-alphaInp*:
(*qOp delta qinp qbinp* #= *qOp delta′ qinp′ qbinp′*) =
(*delta* = *delta′* ∧ *qinp* %= *qinp′* ∧ *qbinp* %%= *qbinp′*)
unfolding *alphaInp-def alphaBinp-def* **by** *auto*

declare *alpha.simps*(*2*) [*simp del*]
declare *alpha-alphaInp*[*simp*]

lemmas *alphaAll-Simps* =
alpha.simps(*1*) *alpha-alphaInp*
alphaAbs.simps

lemma *alphaInp-refl*:
qGoodInp qinp ⟹ *qinp* %= *qinp*
using *alpha-refl*
unfolding *alphaInp-def qGoodInp-def liftAll-def liftAll2-def sameDom-def*
by *fastforce*

lemma *alphaBinp-refl*:
qGoodBinp qbinp ⟹ *qbinp* %%= *qbinp*
using *alphaAbs-refl*
unfolding *alphaBinp-def qGoodBinp-def liftAll-def liftAll2-def sameDom-def*
by *fastforce*

lemma *alphaInp-sym*:
fixes *qinp qinp′* :: (*′index*,(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qTerm*)*input*
shows *qinp* %= *qinp′* ⟹ *qinp′* %= *qinp*
using *alpha-sym* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *blast*

lemma *alphaBinp-sym*:
fixes *qbinp qbinp′* :: (*′bindex*,(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qAbs*)*input*
shows *qbinp* %%= *qbinp′* ⟹ *qbinp′* %%= *qbinp*
using *alphaAbs-sym* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *blast*

lemma *alphaInp-trans*:
assumes *good*: *qGoodInp qinp* **and**
alpha1: *qinp* %= *qinp′* **and** *alpha2*: *qinp′* %= *qinp″*
shows *qinp* %= *qinp″*
proof−
{**fix** *i qX qX″* **assume** *qinp*: *qinp i* = *Some qX* **and** *qinp″*: *qinp″ i* = *Some qX″*
then obtain *qX′* **where** *qinp′*: *qinp′ i* = *Some qX′*
using *alpha1* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by**(*cases qinp′ i*, *force*)
hence *qX* #= *qX′*
using *alpha1 qinp* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *auto*

moreover have *qX′* #= *qX″* **using** *alpha2 qinp′ qinp″*
unfolding *alphaInp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qGood qX* **using** *good qinp* **unfolding** *qGoodInp-def liftAll-def* **by** *auto*
ultimately have *qX* #= *qX″* **using** *alpha-trans* **by** *blast*
}
thus *?thesis* **using** *assms* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *auto*
qed

lemma *alphaBinp-trans*:
assumes *good*: *qGoodBinp qbinp* **and**
alpha1: *qbinp* %%= *qbinp′* **and** *alpha2*: *qbinp′* %%= *qbinp″*
shows *qbinp* %%= *qbinp″*
proof−
{fix *i qA qA″* **assume** *qbinp*: *qbinp i* = *Some qA* **and** *qbinp″*: *qbinp″ i* = *Some qA″*
then obtain *qA′* **where** *qbinp′*: *qbinp′ i* = *Some qA′*
using *alpha1* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by**(*cases qbinp′ i*, *force*)
hence *qA* $= *qA′*
using *alpha1 qbinp* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qA′* $= *qA″* **using** *alpha2 qbinp′ qbinp″*
unfolding *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qGoodAbs qA* **using** *good qbinp* **unfolding** *qGoodBinp-def liftAll-def* **by** *auto*
ultimately have *qA* $= *qA″* **using** *alphaAbs-trans* **by** *blast*
}
thus *?thesis* **using** *assms* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
qed

lemma *qSwapInp-preserves-qGoodInp*:
assumes *qGoodInp qinp*
shows *qGoodInp* (*qinp* %[[*x1* ∧ *x2*]]-*xs*)
proof−
{let *?qinp′* = *lift* (*qSwap xs x1 x2*) *qinp*
fix *xsa* **let** *?Left* = {*i*. *?qinp′ i* ≠ *None*}
have *?Left* = {*i*. *qinp i* ≠ *None*} **by**(*auto simp add*: *lift-None*)
hence |*?Left*| <*o* |*UNIV* :: ′*var set*| **using** *assms* **unfolding** *qGoodInp-def* **by** *auto*
}
thus *?thesis* **using** *assms*
unfolding *qGoodInp-def qSwapInp-def liftAll-lift-comp qGoodInp-def*
unfolding *comp-def liftAll-def*
by (*auto simp add*: *qSwap-preserves-qGood simp del*: *not-None-eq*)
qed

lemma *qSwapBinp-preserves-qGoodBinp*:

assumes *qGoodBinp qbinp*
shows *qGoodBinp* (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*)
proof−
{**let** *?qbinp'* = *lift* (*qSwapAbs xs x1 x2*) *qbinp*
fix *xsa* **let** *?Left* = {*i*. *?qbinp' i* ≠ *None*}
have *?Left* = {*i*. *qbinp i* ≠ *None*} **by**(*auto simp add*: *lift-None*)
hence |*?Left*| <*o* |*UNIV* :: *'var set*| **using** *assms* **unfolding** *qGoodBinp-def* **by** *auto*
}
thus *?thesis* **using** *assms*
unfolding *qGoodBinp-def qSwapBinp-def liftAll-lift-comp*
unfolding *qGoodBinp-def* **unfolding** *comp-def liftAll-def*
by (*auto simp add*: *qSwapAbs-preserves-qGoodAbs simp del*: *not-None-eq*)
qed

lemma *qSwapInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp'* **and** *qinp* %= *qinp'*
shows (*qinp* %[[*x1* ∧ *x2*]]-*xs*) %= (*qinp'* %[[*x1* ∧ *x2*]]-*xs*)
using *assms* **unfolding** *alphaInp-def qSwapInp-def sameDom-def liftAll2-def*
by (*simp add*: *lift-None*)
(*smt* (*verit*) *liftAll-def lift-def option.case-eq-if option.exhaust-sel*
option.sel qGoodInp-def qSwap-preserves-alpha)

lemma *qSwapBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp'* **and** *qbinp* %%= *qbinp'*
shows (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*) %%= (*qbinp'* %%[[*x1* ∧ *x2*]]-*xs*)
using *assms* **unfolding** *alphaBinp-def qSwapBinp-def sameDom-def liftAll2-def*
by (*simp add*: *lift-None*)
(*smt* (*verit*) *liftAll-def lift-def option.case-eq-if option.exhaust-sel option.sel*
qGoodBinp-def qSwapAbs-preserves-alphaAbs)

lemma *qPsubstInp-preserves-qGoodInp*:
assumes *qGoodInp qinp* **and** *qGoodEnv qrho*
shows *qGoodInp* (*qinp* %[[*qrho*]])
using *assms* **unfolding** *qGoodInp-def qPsubstInp-def liftAll-def*
by *simp* (*smt* (*verit*) *Collect-cong lift-def option.case-eq-if*
option.exhaust-sel option.sel qPsubst-preserves-qGood)

lemma *qPsubstBinp-preserves-qGoodBinp*:
assumes *qGoodBinp qbinp* **and** *qGoodEnv qrho*
shows *qGoodBinp* (*qbinp* %%[[*qrho*]])
using *assms* **unfolding** *qGoodBinp-def qPsubstBinp-def liftAll-def*
by *simp* (*smt* (*verit*) *Collect-cong lift-def option.case-eq-if*
option.exhaust-sel option.sel qPsubstAbs-preserves-qGoodAbs)

lemma *qPsubstInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp'* **and** *qGoodEnv qrho* **and** *qinp* %= *qinp'*
shows (*qinp* %[[*qrho*]]) %= (*qinp'* %[[*qrho*]])
using *assms* **unfolding** *alphaInp-def qPsubstInp-def sameDom-def liftAll2-def*

by (*simp add*: *lift-None*)
(*smt* (*verit*) *liftAll-def lift-def option.case-eq-if option.exhaust-sel*
option.sel qGoodInp-def qPsubst-preserves-alpha1)

lemma *qPsubstBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp′* **and** *qGoodEnv qrho* **and** *qbinp* %%= *qbinp′*
shows (*qbinp* %%[[*qrho*]]) %%= (*qbinp′* %%[[*qrho*]])
using *assms* **unfolding** *alphaBinp-def qPsubstBinp-def sameDom-def liftAll2-def*
by (*simp add*: *lift-None*)
(*smt* (*verit*) *liftAll-def lift-def option.case-eq-if option.exhaust-sel*
option.sel qGoodBinp-def qPsubstAbs-preserves-alphaAbs1)

lemma *qFreshInp-preserves-alphaInp-aux*:
assumes *good*: *qGoodInp qinp* ∨ *qGoodInp qinp′* **and** *alpha*: *qinp* %= *qinp′*
and *fresh*: *qFreshInp xs x qinp*
shows *qFreshInp xs x qinp′*
using *assms* **unfolding** *qFreshInp-def liftAll-def* **proof** *clarify*
fix *i qX′* **assume** *qinp′*: *qinp′ i* = *Some qX′*
then obtain *qX* **where** *qinp*: *qinp i* = *Some qX*
using *alpha* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** (*cases qinp i*, *auto*)
hence *qGood qX* ∨ *qGood qX′*
using *qinp′ good* **unfolding** *qGoodInp-def liftAll-def* **by** *auto*
moreover have *qX* #= *qX′*
using *qinp qinp′ alpha* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qFresh xs x qX*
using *fresh qinp* **unfolding** *qFreshInp-def liftAll-def* **by** *simp*
ultimately show *qFresh xs x qX′*
using *qFresh-preserves-alpha* **by** *auto*
qed

lemma *qFreshBinp-preserves-alphaBinp-aux*:
assumes *good*: *qGoodBinp qbinp* ∨ *qGoodBinp qbinp′* **and** *alpha*: *qbinp* %%= *qbinp′*
and *fresh*: *qFreshBinp xs x qbinp*
shows *qFreshBinp xs x qbinp′*
using *assms* **unfolding** *qFreshBinp-def liftAll-def* **proof** *clarify*
fix *i qA′* **assume** *qbinp′*: *qbinp′ i* = *Some qA′*
then obtain *qA* **where** *qbinp*: *qbinp i* = *Some qA*
using *alpha* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** (*cases qbinp i*, *auto*)
hence *qGoodAbs qA* ∨ *qGoodAbs qA′*
using *qbinp′ good* **unfolding** *qGoodBinp-def liftAll-def* **by** *auto*
moreover have *qA* $= *qA′*
using *qbinp qbinp′ alpha* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qFreshAbs xs x qA*
using *fresh qbinp* **unfolding** *qFreshBinp-def liftAll-def* **by** *simp*

ultimately show *qFreshAbs xs x qA'*
using *qFreshAbs-preserves-alphaAbs* **by** *auto*
qed

lemma *qFreshInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp'* **and** *qinp* %= *qinp'*
shows *qFreshInp xs x qinp* ⟷ *qFreshInp xs x qinp'*
using *alphaInp-sym assms qFreshInp-preserves-alphaInp-aux* **by** *blast*

lemma *qFreshBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp'* **and** *qbinp* %%= *qbinp'*
shows *qFreshBinp xs x qbinp* ⟷ *qFreshBinp xs x qbinp'*
using *alphaBinp-sym assms qFreshBinp-preserves-alphaBinp-aux* **by** *blast*

lemmas *qItem-simps* =
qSkelAll-simps qFreshAll-simps qSwapAll-simps qPsubstAll-simps qGoodAll-simps alphaAll-Simps
qSwap-qAFresh-otherSimps qAFresh.simps qGoodItem.simps

end

5.2 Definitions of terms and their operators

type-synonym (*'index,'bindex,'varSort,'var,'opSym*)*term* =
(*'index,'bindex,'varSort,'var,'opSym*)*qTerm set*

type-synonym (*'index,'bindex,'varSort,'var,'opSym*)*abs* =
(*'index,'bindex,'varSort,'var,'opSym*)*qAbs set*

type-synonym (*'index,'bindex,'varSort,'var,'opSym*)*env* =
'varSort ⇒ *'var* ⇒ (*'index,'bindex,'varSort,'var,'opSym*)*term option*

A "parameter" will be something for which freshness makes sense. Here is the most typical case of a parameter in proofs, putting together (as lists) finite collections of variables, terms, abstractions and environments:

datatype (*'index,'bindex,'varSort,'var,'opSym*)*param* =
Par 'var list
(*'index,'bindex,'varSort,'var,'opSym*)*term list*
(*'index,'bindex,'varSort,'var,'opSym*)*abs list*
(*'index,'bindex,'varSort,'var,'opSym*)*env list*

fun *varsOf* **where**
varsOf (*Par xL - - -*) = *set xL*

fun *termsOf* **where**
termsOf (*Par - XL - -*) = *set XL*

fun *absOf* **where**
absOf (*Par* - - *AL* -) = *set AL*

fun *envsOf* **where**
envsOf (*Par* - - - *rhoL*) = *set rhoL*

context *FixVars*
begin

definition *alphaGood* ≡ λ *qX qY*. *qGood qX* ∧ *qGood qY* ∧ *qX* #= *qY*
definition *alphaAbsGood* ≡ λ *qA qB*. *qGoodAbs qA* ∧ *qGoodAbs qB* ∧ *qA* $= *qB*

definition *good* ≡ *qGood* /// *alphaGood*
definition *goodAbs* ≡ *qGoodAbs* /// *alphaAbsGood*

definition *goodInp* **where**
goodInp inp ==
liftAll good inp ∧
|{*i*. *inp i* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|

definition *goodBinp* **where**
goodBinp binp ==
liftAll goodAbs binp ∧
|{*i*. *binp i* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|

definition *goodEnv* **where**
goodEnv rho ==
(∀ *ys*. *liftAll good* (*rho ys*)) ∧
(∀ *ys*. |{*y*. *rho ys y* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|)

definition *asTerm* **where**
asTerm qX ≡ *proj alphaGood qX*

definition *asAbs* **where**
asAbs qA ≡ *proj alphaAbsGood qA*

definition *pickInp* **where**
pickInp inp ≡ *lift pick inp*

definition *pickBinp* **where**
pickBinp binp ≡ *lift pick binp*

definition *asInp* **where**
asInp qinp ≡ *lift asTerm qinp*

definition *asBinp* **where**
asBinp qbinp ≡ *lift asAbs qbinp*

definition *pickE* **where**
pickE rho ≡ λ *xs. lift pick* (*rho xs*)

definition *asEnv* **where**
asEnv qrho ≡ λ *xs. lift asTerm* (*qrho xs*)

definition *Var* **where**
Var xs x ≡ *asTerm*(*qVar xs x*)

definition *Op* **where**
Op delta inp binp ≡ *asTerm* (*qOp delta* (*pickInp inp*) (*pickBinp binp*))

definition *Abs* **where**
Abs xs x X ≡ *asAbs* (*qAbs xs x* (*pick X*))

definition *skel* **where**
skel X ≡ *qSkel* (*pick X*)

definition *skelAbs* **where**
skelAbs A ≡ *qSkelAbs* (*pick A*)

definition *skelInp* **where**
skelInp inp = *qSkelInp* (*pickInp inp*)

definition *skelBinp* **where**
skelBinp binp = *qSkelBinp* (*pickBinp binp*)

lemma *skelInp-def2*:
assumes *goodInp inp*
shows *skelInp inp* = *lift skel inp*
unfolding *skelInp-def*
unfolding *qSkelInp-def pickInp-def skel-def*[*abs-def*]
unfolding *lift-comp comp-def* **by** *simp*

lemma *skelBinp-def2*:
assumes *goodBinp binp*
shows *skelBinp binp* = *lift skelAbs binp*
unfolding *skelBinp-def*
unfolding *qSkelBinp-def pickBinp-def skelAbs-def*[*abs-def*]
unfolding *lift-comp comp-def* **by** *simp*

definition *swap* **where**
swap xs x y X = *asTerm* (*qSwap xs x y* (*pick X*))

abbreviation *swap-abbrev* (‹- #[- ∧ -]′-› *200*) **where**
(*X* #[*z1* ∧ *z2*]-*zs*) ≡ *swap zs z1 z2 X*

definition *swapAbs* **where**
swapAbs xs x y A = *asAbs* (*qSwapAbs xs x y* (*pick A*))

abbreviation *swapAbs-abbrev* (‹- $[- ∧ -]′--› *200*) **where**
(*A* $[*z1* ∧ *z2*]-*zs*) ≡ *swapAbs zs z1 z2 A*

definition *swapInp* **where**
swapInp xs x y inp ≡ *lift* (*swap xs x y*) *inp*

definition *swapBinp* **where**
swapBinp xs x y binp ≡ *lift* (*swapAbs xs x y*) *binp*

abbreviation *swapInp-abbrev* (‹- %[- ∧ -]′--› *200*) **where**
(*inp* %[*z1* ∧ *z2*]-*zs*) ≡ *swapInp zs z1 z2 inp*

abbreviation *swapBinp-abbrev* (‹- %%[- ∧ -]′--› *200*) **where**
(*binp* %%[*z1* ∧ *z2*]-*zs*) ≡ *swapBinp zs z1 z2 binp*

definition *swapEnvDom* **where**
swapEnvDom xs x y rho ≡ λ*zs z*. *rho zs* (*z* @*zs*[*x* ∧ *y*]-*xs*)

definition *swapEnvIm* **where**
swapEnvIm xs x y rho ≡ λ*zs*. *lift* (*swap xs x y*) (*rho zs*)

definition *swapEnv* **where**
swapEnv xs x y ≡ *swapEnvIm xs x y o swapEnvDom xs x y*

abbreviation *swapEnv-abbrev* (‹- &[- ∧ -]′--› *200*) **where**
(*rho* &[*z1* ∧ *z2*]-*zs*) ≡ *swapEnv zs z1 z2 rho*

lemmas *swapEnv-defs* = *swapEnv-def comp-def swapEnvDom-def swapEnvIm-def*

inductive-set *swapped* **where**
Refl: (*X*,*X*) ∈ *swapped*
|
Trans: ⟦(*X*,*Y*) ∈ *swapped*; (*Y*,*Z*) ∈ *swapped*⟧ ⟹ (*X*,*Z*) ∈ *swapped*
|
Swap: (*X*,*Y*) ∈ *swapped* ⟹ (*X*, *Y* #[*x* ∧ *y*]-*zs*) ∈ *swapped*

lemmas *swapped-Clauses* = *swapped.Refl swapped.Trans swapped.Swap*

definition *fresh* **where**
fresh xs x X ≡ *qFresh xs x* (*pick X*)

definition *freshAbs* **where**
freshAbs xs x A ≡ *qFreshAbs xs x* (*pick A*)

definition *freshInp* **where**

freshInp xs x inp ≡ *liftAll* (*fresh xs x*) *inp*

definition *freshBinp* **where**
freshBinp xs x binp ≡ *liftAll* (*freshAbs xs x*) *binp*

definition *freshEnv* **where**
freshEnv xs x rho ==
rho xs x = *None* ∧ (∀ *ys*. *liftAll* (*fresh xs x*) (*rho ys*))

definition *psubst* **where**
psubst rho X ≡ *asTerm*(*qPsubst* (*pickE rho*) (*pick X*))

abbreviation *psubst-abbrev* (‹- #[-]›) **where**
(*X* #[*rho*]) ≡ *psubst rho X*

definition *psubstAbs* **where**
psubstAbs rho A ≡ *asAbs*(*qPsubstAbs* (*pickE rho*) (*pick A*))

abbreviation *psubstAbs-abbrev* (‹- $[-]›) **where**
A $[*rho*] ≡ *psubstAbs rho A*

definition *psubstInp* **where**
psubstInp rho inp ≡ *lift* (*psubst rho*) *inp*

definition *psubstBinp* **where**
psubstBinp rho binp ≡ *lift* (*psubstAbs rho*) *binp*

abbreviation *psubstInp-abbrev* (‹- %[-]›) **where**
inp %[*rho*] ≡ *psubstInp rho inp*

abbreviation *psubstBinp-abbrev* (‹- %%[-]›) **where**
binp %%[*rho*] ≡ *psubstBinp rho binp*

definition *psubstEnv* **where**
psubstEnv rho rho′ ≡
λ *xs x*. *case rho′ xs x of None* ⇒ *rho xs x*
|*Some X* ⇒ *Some* (*X* #[*rho*])

abbreviation *psubstEnv-abbrev* (‹- &[-]›) **where**
rho &[*rho′*] ≡ *psubstEnv rho′ rho*

definition *idEnv* **where**
idEnv ≡ λ*xs*. *Map.empty*

definition *updEnv* ::
(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env* ⇒
′*var* ⇒ (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term* ⇒ ′*varSort* ⇒
(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env*
(‹- [- ← -]′-› *200*) **where**

(*rho* [*x* ← *X*]-*xs*) ≡ λ *ys y*. (*if ys* = *xs* ∧ *y* = *x then Some X else rho ys y*)

(Unary) substitution:

definition *subst* **where**
subst xs X x ≡ *psubst* (*idEnv* [*x* ← *X*]-*xs*)

abbreviation *subst-abbrev* (‹- #[- '/ -]'-› *200*) **where**
(*Y* #[*X* / *x*]-*xs*) ≡ *subst xs X x Y*

definition *substAbs* **where**
substAbs xs X x ≡ *psubstAbs* (*idEnv* [*x* ← *X*]-*xs*)

abbreviation *substAbs-abbrev* (‹- $[- '/ -]'-› *200*) **where**
(*A* $[*X* / *x*]-*xs*) ≡ *substAbs xs X x A*

definition *substInp* **where**
substInp xs X x ≡ *psubstInp* (*idEnv* [*x* ← *X*]-*xs*)

definition *substBinp* **where**
substBinp xs X x ≡ *psubstBinp* (*idEnv* [*x* ← *X*]-*xs*)

abbreviation *substInp-abbrev* (‹- %[- '/ -]'-› *200*) **where**
(*inp* %[*X* / *x*]-*xs*) ≡ *substInp xs X x inp*

abbreviation *substBinp-abbrev* (‹- %%[- '/ -]'-› *200*) **where**
(*binp* %%[*X* / *x*]-*xs*) ≡ *substBinp xs X x binp*

theorem *substInp-def2*:
substInp ys Y y = *lift* (*subst ys Y y*)
unfolding *substInp-def*[*abs-def*] *subst-def psubstInp-def*[*abs-def*] **by** *simp*

theorem *substBinp-def2*:
substBinp ys Y y = *lift* (*substAbs ys Y y*)
unfolding *substBinp-def*[*abs-def*] *substAbs-def psubstBinp-def*[*abs-def*] **by** *simp*

definition *substEnv* **where**
substEnv xs X x ≡ *psubstEnv* (*idEnv* [*x* ← *X*]-*xs*)

abbreviation *substEnv-abbrev* (‹- &[- '/ -]'-› *200*) **where**
(*Y* &[*X* / *x*]-*xs*) ≡ *substEnv xs X x Y*

theorem *substEnv-def2*:
(*rho* &[*Y* / *y*]-*ys*) =
 (λ*xs x*. *case rho xs x of*
  None ⇒ *if* (*xs* = *ys* ∧ *x* = *y*) *then Some Y else None*
  |*Some X* ⇒ *Some* (*X* #[*Y* / *y*]-*ys*))
unfolding *substEnv-def psubstEnv-def subst-def idEnv-def updEnv-def*
apply(*rule ext*)+ **by**(*case-tac rho xs x*, *simp-all*)

Variable-for-variable substitution:

definition *vsubst* **where**
vsubst ys y1 y2 ≡ *subst ys* (*Var ys y1*) *y2*

abbreviation *vsubst-abbrev* (‹- #[- '/'/ -]'-› *200*) **where**
(*X* #[*y1* // *y2*]-*ys*) ≡ *vsubst ys y1 y2 X*

definition *vsubstAbs* **where**
vsubstAbs ys y1 y2 ≡ *substAbs ys* (*Var ys y1*) *y2*

abbreviation *vsubstAbs-abbrev* (‹- $[- '/'/ -]'-› *200*) **where**
(*A* $[*y1* // *y2*]-*ys*) ≡ *vsubstAbs ys y1 y2 A*

definition *vsubstInp* **where**
vsubstInp ys y1 y2 ≡ *substInp ys* (*Var ys y1*) *y2*

definition *vsubstBinp* **where**
vsubstBinp ys y1 y2 ≡ *substBinp ys* (*Var ys y1*) *y2*

abbreviation *vsubstInp-abbrev* (‹- %[- '/'/ -]'-› *200*) **where**
(*inp* %[*y1* // *y2*]-*ys*) ≡ *vsubstInp ys y1 y2 inp*

abbreviation *vsubstBinp-abbrev* (‹- %%[- '/'/ -]'-› *200*) **where**
(*binp* %%[*y1* // *y2*]-*ys*) ≡ *vsubstBinp ys y1 y2 binp*

lemma *vsubstInp-def2*:
(*inp* %[*y1* // *y2*]-*ys*) = *lift* (*vsubst ys y1 y2*) *inp*
unfolding *vsubstInp-def vsubst-def*
by(*auto simp add*: *substInp-def2*)

lemma *vsubstBinp-def2*:
(*binp* %%[*y1* // *y2*]-*ys*) = *lift* (*vsubstAbs ys y1 y2*) *binp*
unfolding *vsubstBinp-def vsubstAbs-def*
by(*auto simp add*: *substBinp-def2*)

definition *vsubstEnv* **where**
vsubstEnv ys y1 y2 ≡ *substEnv ys* (*Var ys y1*) *y2*

abbreviation *vsubstEnv-abbrev* (‹- &[- '/'/ -]'-› *200*) **where**
(*rho* &[*y1* // *y2*]-*ys*) ≡ *vsubstEnv ys y1 y2 rho*

theorem *vsubstEnv-def2*:
(*rho* &[*y1* // *y*]-*ys*) =
 (λ*xs x*. *case rho xs x of*
  None ⇒ *if* (*xs* = *ys* ∧ *x* = *y*) *then Some* (*Var ys y1*) *else None*
 |*Some X* ⇒ *Some* (*X* #[*y1* // *y*]-*ys*))
unfolding *vsubstEnv-def vsubst-def* **by**(*auto simp add*: *substEnv-def2*)

definition *goodPar* **where**
goodPar P ≡ (∀ *X* ∈ *termsOf P*. *good X*) ∧

(∀ *A* ∈ *absOf P*. *goodAbs A*) ∧
(∀ *rho* ∈ *envsOf P*. *goodEnv rho*)

lemma *Par-preserves-good*[*simp*]:
assumes !! *X*. *X* ∈ *set XL* ⟹ *good X*
and !! *A*. *A* ∈ *set AL* ⟹ *goodAbs A*
and !! *rho*. *rho* ∈ *set rhoL* ⟹ *goodEnv rho*
shows *goodPar* (*Par xL XL AL rhoL*)
using *assms* **unfolding** *goodPar-def* **by** *auto*

lemma *termsOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *X* : *termsOf P*
shows *good X*
using *assms* **unfolding** *goodPar-def* **by** *auto*

lemma *absOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *A* : *absOf P*
shows *goodAbs A*
using *assms* **unfolding** *goodPar-def* **by** *auto*

lemma *envsOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *rho* : *envsOf P*
shows *goodEnv rho*
using *assms* **unfolding** *goodPar-def* **by** *blast*

lemmas *param-simps* =
termsOf.simps absOf.simps envsOf.simps
Par-preserves-good
termsOf-preserves-good absOf-preserves-good envsOf-preserves-good

5.3 Items versus quasi-items modulo alpha

Here we "close the accounts" (for a while) with quasi-items – beyond this subsection, there will not be any theorem that mentions quasi-items, except much later when we deal with iteration principles (and need to briefly switch back to quasi-terms in order to define the needed iterative map by the universality of the alpha-quotient).

5.3.1 For terms

lemma *alphaGood-equivP*: *equivP qGood alphaGood*
unfolding *equivP-def reflP-def symP-def transP-def alphaGood-def*
using *alpha-refl alpha-sym alpha-trans* **by** *blast*

lemma *univ-asTerm-alphaGood*[*simp*]:
assumes ∗: *congruentP alphaGood f* **and** ∗∗: *qGood X*
shows *univ f* (*asTerm X*) = *f X*
by (*metis assms alphaGood-equivP asTerm-def univ-commute*)

corollary *univ-asTerm-alpha*[*simp*]:
assumes ∗: *congruentP alpha f* **and** ∗∗: *qGood X*
shows *univ f* (*asTerm X*) = *f X*
apply(*rule univ-asTerm-alphaGood*)
using *assms* **unfolding** *alphaGood-def congruentP-def* **by** *auto*

lemma *pick-inj-on-good*: *inj-on pick* (*Collect good*)
unfolding *good-def* **using** *alphaGood-equivP equivP-pick-inj-on* **by** *auto*

lemma *pick-injective-good*[*simp*]:
⟦*good X*; *good Y*⟧ ⟹ (*pick X* = *pick Y*) = (*X* = *Y*)
using *pick-inj-on-good* **unfolding** *inj-on-def* **by** *auto*

lemma *good-imp-qGood-pick*:
good X ⟹ *qGood* (*pick X*)
unfolding *good-def*
by (*metis alphaGood-equivP equivP-pick-preserves*)

lemma *qGood-iff-good-asTerm*:
good (*asTerm qX*) = *qGood qX*
unfolding *good-def asTerm-def*
using *alphaGood-equivP proj-in-iff* **by** *fastforce*

lemma *pick-asTerm*:
assumes *qGood qX*
shows *pick* (*asTerm qX*) #= *qX*
by (*metis* (*full-types*) *alphaGood-def alphaGood-equivP asTerm-def assms pick-proj*)

lemma *asTerm-pick*:
assumes *good X*
shows *asTerm* (*pick X*) = *X*
by (*metis alphaGood-equivP asTerm-def assms good-def proj-pick*)

lemma *pick-alpha*: *good X* ⟹ *pick X* #= *pick X*
using *good-imp-qGood-pick alpha-refl* **by** *auto*

lemma *alpha-imp-asTerm-equal*:
assumes *qGood qX* **and** *qX* #= *qY*
shows *asTerm qX* = *asTerm qY*
proof−
  have *alphaGood qX qY* **unfolding** *alphaGood-def* **using** *assms*
  by (*metis alpha-preserves-qGood*)
  thus *?thesis* **unfolding** *asTerm-def* **using** *alphaGood-equivP proj-iff*
  by (*metis alpha-preserves-qGood1 assms*)
qed

lemma *asTerm-equal-imp-alpha*:
assumes *qGood qX* **and** *asTerm qX* = *asTerm qY*

shows *qX #= qY*
by (*metis alphaAll-sym alphaAll-trans assms pick-asTerm qGood-iff-good-asTerm*)

lemma *asTerm-equal-iff-alpha*:
assumes *qGood qX* ∨ *qGood qY*
shows (*asTerm qX* = *asTerm qY*) = (*qX #= qY*)
by (*metis alpha-imp-asTerm-equal alpha-sym asTerm-equal-imp-alpha assms*)

lemma *pick-alpha-iff-equal*:
assumes *good X* **and** *good Y*
shows *pick X #= pick Y* ⟷ *X* = *Y*
by (*metis asTerm-equal-iff-alpha asTerm-pick assms good-imp-qGood-pick*)

lemma *pick-swap-qSwap*:
assumes *good X*
shows *pick* (*X #*[*x1* ∧ *x2*]-*xs*) #= ((*pick X*) #[[*x1* ∧ *x2*]]-*xs*)
by (*metis assms good-imp-qGood-pick pick-asTerm qSwap-preserves-qGood1 swap-def*)

lemma *asTerm-qSwap-swap*:
assumes *qGood qX*
shows *asTerm* (*qX #*[[*x1* ∧ *x2*]]-*xs*) = ((*asTerm qX*) #[*x1* ∧ *x2*]-*xs*)
by (*simp add*: *alpha-imp-asTerm-equal alpha-sym assms local.swap-def pick-asTerm qSwap-preserves-alpha qSwap-preserves-qGood1*)

lemma *fresh-asTerm-qFresh*:
assumes *qGood qX*
shows *fresh xs x* (*asTerm qX*) = *qFresh xs x qX*
by (*simp add*: *assms fresh-def pick-asTerm qFresh-preserves-alpha*)

lemma *skel-asTerm-qSkel*:
assumes *qGood qX*
shows *skel* (*asTerm qX*) = *qSkel qX*
by (*simp add*: *alpha-qSkel assms pick-asTerm skel-def*)

lemma *double-swap-qSwap*:
assumes *good X*
shows *qGood* (((*pick X*) #[[*x* ∧ *y*]]-*zs*) #[[*x'* ∧ *y'*]]-*zs'*) ∧
((*X #*[*x* ∧ *y*]-*zs*) #[*x'* ∧ *y'*]-*zs'*) = *asTerm* (((*pick X*) #[[*x* ∧ *y*]]-*zs*) #[[*x'* ∧ *y'*]]-*zs'*)
by (*simp add*: *asTerm-qSwap-swap assms good-imp-qGood-pick local.swap-def qSwap-preserves-qGood1*)

lemma *fresh-swap-qFresh-qSwap*:
assumes *good X*
shows *fresh xs x* (*X #*[*y1* ∧ *y2*]-*ys*) = *qFresh xs x* ((*pick X*) #[[*y1* ∧ *y2*]]-*ys*)
by (*simp add*: *assms fresh-asTerm-qFresh good-imp-qGood-pick local.swap-def qSwap-preserves-qGood*)

5.3.2 For abstractions

lemma *alphaAbsGood-equivP*: *equivP qGoodAbs alphaAbsGood*
unfolding *equivP-def reflP-def symP-def transP-def alphaAbsGood-def*
using *alphaAbs-refl alphaAbs-sym alphaAbs-trans* **by** *blast*

lemma *univ-asAbs-alphaAbsGood*[*simp*]:
assumes *fAbs respectsP alphaAbsGood* **and** *qGoodAbs A*
shows *univ fAbs* (*asAbs A*) = *fAbs A*
by (*metis assms alphaAbsGood-equivP asAbs-def univ-commute*)

corollary *univ-asAbs-alphaAbs*[*simp*]:
assumes *: *fAbs respectsP alphaAbs* **and** **: *qGoodAbs A*
shows *univ fAbs* (*asAbs A*) = *fAbs A*
apply(*rule univ-asAbs-alphaAbsGood*)
using *assms* **unfolding** *alphaAbsGood-def congruentP-def* **by** *auto*

lemma *pick-inj-on-goodAbs*: *inj-on pick* (*Collect goodAbs*)
unfolding *goodAbs-def* **using** *alphaAbsGood-equivP equivP-pick-inj-on* **by** *auto*

lemma *pick-injective-goodAbs*[*simp*]:
⟦*goodAbs A*; *goodAbs B*⟧ ⟹ *pick A* = *pick B* ⟷ *A* = *B*
using *pick-inj-on-goodAbs* **unfolding** *inj-on-def* **by** *auto*

lemma *goodAbs-imp-qGoodAbs-pick*:
goodAbs A ⟹ *qGoodAbs* (*pick A*)
unfolding *goodAbs-def*
using *alphaAbsGood-equivP equivP-pick-preserves* **by** *fastforce*

lemma *qGoodAbs-iff-goodAbs-asAbs*:
goodAbs(*asAbs qA*) = *qGoodAbs qA*
unfolding *goodAbs-def asAbs-def*
using *alphaAbsGood-equivP proj-in-iff* **by** *fastforce*

lemma *pick-asAbs*:
assumes *qGoodAbs qA*
shows *pick* (*asAbs qA*) $= *qA*
by (*metis* (*full-types*) *alphaAbsGood-def alphaAbsGood-equivP asAbs-def assms pick-proj*)

lemma *asAbs-pick*:
assumes *goodAbs A*
shows *asAbs* (*pick A*) = *A*
by (*metis alphaAbsGood-equivP asAbs-def assms goodAbs-def proj-pick*)

lemma *pick-alphaAbs*: *goodAbs A* ⟹ *pick A* $= *pick A*
using *goodAbs-imp-qGoodAbs-pick alphaAbs-refl* **by** *auto*

lemma *alphaAbs-imp-asAbs-equal*:
assumes *qGoodAbs qA* **and** *qA* $= *qB*
shows *asAbs qA* = *asAbs qB*

by (*metis* (*no-types*, *opaque-lifting*) *proj-iff alphaAbsGood-def alphaAbsGood-equivP*

alphaAbs-preserves-qGoodAbs asAbs-def assms)

lemma *asAbs-equal-imp-alphaAbs*:
assumes *qGoodAbs qA* **and** *asAbs qA* = *asAbs qB*
shows *qA* $= *qB*
by (*metis alphaAbs-refl*
alphaAbs-sym alphaAbs-trans-twice assms pick-asAbs qGoodAbs-iff-goodAbs-asAbs)

lemma *asAbs-equal-iff-alphaAbs*:
assumes *qGoodAbs qA* ∨ *qGoodAbs qB*
shows (*asAbs qA* = *asAbs qB*) = (*qA* $= *qB*)
by (*metis alphaAbs-imp-asAbs-equal alphaAbs-preserves-qGoodAbs*
asAbs-equal-imp-alphaAbs assms)

lemma *pick-alphaAbs-iff-equal*:
assumes *goodAbs A* **and** *goodAbs B*
shows (*pick A* $= *pick B*) = (*A* = *B*)
using *asAbs-equal-iff-alphaAbs asAbs-pick assms goodAbs-imp-qGoodAbs-pick* **by** *blast*

lemma *pick-swapAbs-qSwapAbs*:
assumes *goodAbs A*
shows *pick* (*A* $[*x1* ∧ *x2*]-*xs*) $= ((*pick A*) $[[*x1* ∧ *x2*]]-*xs*)
by (*simp add*: *assms goodAbs-imp-qGoodAbs-pick*
pick-asAbs qSwapAbs-preserves-qGoodAbs swapAbs-def)

lemma *asAbs-qSwapAbs-swapAbs*:
assumes *qGoodAbs qA*
shows *asAbs* (*qA* $[[*x1* ∧ *x2*]]-*xs*) = ((*asAbs qA*) $[*x1* ∧ *x2*]-*xs*)
by (*simp add*: *alphaAbs-imp-asAbs-equal alphaAbs-sym assms pick-asAbs*
qSwapAbs-preserves-alphaAbs
qSwapAbs-preserves-qGoodAbs1 swapAbs-def)

lemma *freshAbs-asAbs-qFreshAbs*:
assumes *qGoodAbs qA*
shows *freshAbs xs x* (*asAbs qA*) = *qFreshAbs xs x qA*
by (*simp add*: *assms freshAbs-def pick-asAbs qFreshAbs-preserves-alphaAbs*)

lemma *skelAbs-asAbs-qSkelAbs*:
assumes *qGoodAbs qA*
shows *skelAbs* (*asAbs qA*) = *qSkelAbs qA*
by (*simp add*: *alphaAll-qSkelAll assms pick-asAbs skelAbs-def*)

5.3.3 For inputs

For unbound inputs:

lemma *pickInp-inj-on-goodInp*: *inj-on pickInp* (*Collect goodInp*)

unfolding *pickInp-def*[*abs-def*] *inj-on-def*
proof(*safe*, *rule ext*)
  fix *inp inp′ i*
  assume *good*: *goodInp inp goodInp inp′* **and** ∗: *lift pick inp* = *lift pick inp′*
  show *inp i* = *inp′ i*
  proof(*cases inp i*)
    assume *inp*: *inp i* = *None*
    hence *lift pick inp i* = *None* **by** (*auto simp add*: *lift-None*)
    hence *lift pick inp′ i* = *None* **using** ∗ **by** *simp*
    hence *inp′ i* = *None* **by** (*auto simp add*: *lift-None*)
    thus *?thesis* **using** *inp* **by** *simp*
  next
    fix *X* **assume** *inp*: *inp i* = *Some X*
    hence *lift pick inp i* = *Some* (*pick X*) **unfolding** *lift-def* **by** *simp*
    hence *lift pick inp′ i* = *Some* (*pick X*) **using** ∗ **by** *simp*
    then obtain *X′* **where** *inp′*: *inp′ i* = *Some X′* **and** *XX′*: *pick X* = *pick X′*
    unfolding *lift-def* **by**(*cases inp′ i*, *auto*)
    hence *good X* ∧ *good X′*
    using *inp good goodInp-def liftAll-def* **by** (*metis* (*opaque-lifting*, *full-types*))
    hence *X* = *X′* **using** *XX′* **by** *auto*
    thus *?thesis* **unfolding** *inp inp′* **by** *simp*
  qed
qed

lemma *goodInp-imp-qGoodInp-pickInp*:
assumes *goodInp inp*
shows *qGoodInp* (*pickInp inp*)
unfolding *pickInp-def qGoodInp-def liftAll-def*
proof *safe*
  fix *i qX* **assume** *lift pick inp i* = *Some qX*
  then obtain *X* **where** *inp*: *inp i* = *Some X* **and** *qX*: *qX* = *pick X*
  unfolding *lift-def* **by**(*cases inp i*, *auto*)
  hence *good X* **using** *assms*
  unfolding *goodInp-def liftAll-def* **by** *simp*
  thus *qGood qX* **unfolding** *qX* **using** *good-imp-qGood-pick* **by** *auto*
next
  fix *xs*   **let** *?Left* = {*i*. *lift pick inp i* ≠ *None*}
  have *?Left* = {*i*. *inp i* ≠ *None*} **by**(*force simp add*: *lift-None*)
  thus |*?Left*| <*o* |*UNIV* :: ′*var set*| **using** *assms* **unfolding** *goodInp-def* **by** *auto*
qed

lemma *qGoodInp-iff-goodInp-asInp*:
goodInp (*asInp qinp*) = *qGoodInp qinp*
proof(*unfold asInp-def*)
  let *?inp* = *lift asTerm qinp*
  {**assume** *qgood-qinp*: *qGoodInp qinp*
   have *goodInp ?inp*
   unfolding *goodInp-def liftAll-def* **proof** *safe*
     fix *i X* **assume** *inp*: *?inp i* = *Some X*

then obtain *qX* **where** *qinp*: *qinp i* = *Some qX* **and** *X*: *X* = *asTerm qX*
unfolding *lift-def* **by**(*cases qinp i*, *auto*)
hence *qGood qX*
using *qgood-qinp* **unfolding** *qGoodInp-def liftAll-def* **by** *auto*
thus *good X* **using** *X qGood-iff-good-asTerm* **by** *auto*
next
fix *xs* **let** *?Left* = {*i. lift asTerm qinp i* ≠ *None*}
have *?Left* = {*i. qinp i* ≠ *None*} **by**(*auto simp add*: *lift-None*)
thus |*?Left*| <*o* |*UNIV* :: *'var set*| **using** *qgood-qinp* **unfolding** *qGoodInp-def*
by *auto*
qed
}
moreover
{assume *good-inp*: *goodInp ?inp*
have *qGoodInp qinp*
unfolding *qGoodInp-def liftAll-def* **proof** *safe*
fix *i qX* **assume** *qinp*: *qinp i* = *Some qX* **let** *?X* = *asTerm qX*
have *inp*: *?inp i* = *Some ?X* **unfolding** *lift-def* **using** *qinp* **by** *simp*
hence *good ?X*
using *good-inp* **unfolding** *goodInp-def liftAll-def* **by** *auto*
thus *qGood qX* **using** *qGood-iff-good-asTerm* **by** *auto*
next
fix *xs* **let** *?Left* = {*i. qinp i* ≠ *None*}
have *?Left* = {*i. lift asTerm qinp i* ≠ *None*} **by**(*auto simp add*: *lift-None*)
thus |*?Left*| <*o* |*UNIV* :: *'var set*| **using** *good-inp* **unfolding** *goodInp-def* **by**
auto
qed
}
ultimately show *goodInp ?inp* = *qGoodInp qinp* **by** *blast*
qed

lemma *pickInp-asInp*:
assumes *qGoodInp qinp*
shows *pickInp* (*asInp qinp*) %= *qinp*
using *assms* **unfolding** *pickInp-def asInp-def lift-comp*
by (*smt* (*verit*) *CollectI alphaInp-def asTerm-equal-iff-alpha asTerm-pick case-prodI comp-apply liftAll2-def liftAll-def lift-def option.case*(*2*) *option.sel qGoodInp-def qGood-iff-good-asTerm*
sameDom-lift2)

lemma *asInp-pickInp*:
assumes *goodInp inp*
shows *asInp* (*pickInp inp*) = *inp*
unfolding *asInp-def pickInp-def lift-comp*
proof(*rule ext*)
fix *i* **show** *lift* (*asTerm* ∘ *pick*) *inp i* = *inp i*
unfolding *lift-def* **proof**(*cases inp i*, *simp+*)
fix *X* **assume** *inp i* = *Some X*
hence *good X* **using** *assms* **unfolding** *goodInp-def liftAll-def* **by** *simp*

thus *asTerm* (*pick X*) = *X* **using** *asTerm-pick* **by** *auto*
qed
qed

lemma *pickInp-alphaInp*:
assumes *goodInp*: *goodInp inp*
shows *pickInp inp* %= *pickInp inp*
using *assms goodInp-imp-qGoodInp-pickInp alphaInp-refl* **by** *auto*

lemma *alphaInp-imp-asInp-equal*:
assumes *qGoodInp qinp* **and** *qinp* %= *qinp′*
shows *asInp qinp* = *asInp qinp′*
unfolding *asInp-def* **proof**(*rule ext*)
fix *i* **show** *lift asTerm qinp i* = *lift asTerm qinp′ i*
proof(*cases qinp i*)
assume *Case1*: *qinp i* = *None*
hence *qinp′ i* = *None*
using *assms* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *auto*
thus *?thesis* **using** *Case1* **unfolding** *lift-def* **by** *simp*
next
fix *qX* **assume** *Case2*: *qinp i* = *Some qX*
then obtain *qX′* **where** *qinp′*: *qinp′ i* = *Some qX′*
using *assms* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** (*cases qinp′ i*, *force*)
hence *qX* #= *qX′*
using *assms Case2* **unfolding** *alphaInp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qGood qX* **using** *assms Case2* **unfolding** *qGoodInp-def liftAll-def* **by** *auto*
ultimately show *?thesis*
using *Case2 qinp′ alpha-imp-asTerm-equal* **unfolding** *lift-def* **by** *auto*
qed
qed

lemma *asInp-equal-imp-alphaInp*:
assumes *qGoodInp qinp* **and** *asInp qinp* = *asInp qinp′*
shows *qinp* %= *qinp′*
using *assms* **unfolding** *alphaInp-def liftAll2-def sameDom-def*
by *simp* (*smt* (*verit*) *asInp-def asTerm-equal-iff-alpha liftAll-def lift-def option.case*(*2*)

option.sel qGoodInp-def sameDom-def sameDom-lift2)

lemma *asInp-equal-iff-alphaInp*:
qGoodInp qinp ⟹ (*asInp qinp* = *asInp qinp′*) = (*qinp* %= *qinp′*)
using *asInp-equal-imp-alphaInp alphaInp-imp-asInp-equal* **by** *blast*

lemma *pickInp-alphaInp-iff-equal*:
assumes *goodInp inp* **and** *goodInp inp′*
shows (*pickInp inp* %= *pickInp inp′*) = (*inp* = *inp′*)
by (*metis alphaInp-imp-asInp-equal asInp-equal-imp-alphaInp*

asInp-pickInp assms goodInp-imp-qGoodInp-pickInp)

lemma *pickInp-swapInp-qSwapInp*:
assumes *goodInp inp*
shows *pickInp* (*inp* %[*x1* ∧ *x2*]-*xs*) %= ((*pickInp inp*) %[[*x1* ∧ *x2*]]-*xs*)
using *assms* **unfolding** *alphaInp-def sameDom-def liftAll2-def*
pickInp-def swapInp-def qSwapInp-def lift-comp
by (*simp add*: *lift-None*)
(*smt* (*verit*) *assms comp-apply goodInp-imp-qGoodInp-pickInp liftAll-def lift-def local.swap-def option.case-eq-if option.sel option.simps*(*3*) *pickInp-def*
pick-asTerm qGoodInp-def qSwap-preserves-qGood1)

lemma *asInp-qSwapInp-swapInp*:
assumes *qGoodInp qinp*
shows *asInp* (*qinp* %[[*x1* ∧ *x2*]]-*xs*) = ((*asInp qinp*) %[*x1* ∧ *x2*]-*xs*)
proof−
 {**fix** *i qX* **assume** *qinp i* = *Some qX*
 hence *qGood qX* **using** *assms* **unfolding** *qGoodInp-def liftAll-def* **by** *auto*
 hence *asTerm* (*qX* #[[*x1* ∧ *x2*]]-*xs*) = *swap xs x1 x2* (*asTerm qX*)
 by(*auto simp add*: *asTerm-qSwap-swap*)
 }
 thus *?thesis*
 using *assms*
 by (*smt* (*verit*) *asInp-def comp-apply lift-comp lift-cong qSwapInp-def swapInp-def*)
qed

lemma *swapInp-def2*:
(*inp* %[*x1* ∧ *x2*]-*xs*) = *asInp* ((*pickInp inp*) %[[*x1* ∧ *x2*]]-*xs*)
unfolding *swapInp-def asInp-def pickInp-def qSwapInp-def lift-def swap-def*
apply(*rule ext*) **subgoal for** *i* **by** (*cases inp i*) *auto* **.**

lemma *freshInp-def2*:
freshInp xs x inp = *qFreshInp xs x* (*pickInp inp*)
unfolding *freshInp-def qFreshInp-def pickInp-def lift-def fresh-def liftAll-def*
apply(*rule iff-allI*) **subgoal for** *i* **by** (*cases inp i*) *auto* **.**

For bound inputs:

lemma *pickBinp-inj-on-goodBinp*: *inj-on pickBinp* (*Collect goodBinp*)
unfolding *pickBinp-def*[*abs-def*] *inj-on-def*
proof(*safe*, *rule ext*)
 fix *binp binp′ i*
 assume *good*: *goodBinp binp goodBinp binp′* **and** ∗: *lift pick binp* = *lift pick binp′*
 show *binp i* = *binp′ i*
 proof(*cases binp i*)
  assume *binp*: *binp i* = *None*
  hence *lift pick binp i* = *None* **by** (*auto simp add*: *lift-None*)
  hence *lift pick binp′ i* = *None* **using** ∗ **by** *simp*
  hence *binp′ i* = *None* **by** (*auto simp add*: *lift-None*)

thus *?thesis* **using** *binp* **by** *simp*
next
fix *A* **assume** *binp*: *binp i* = *Some A*
hence *lift pick binp i* = *Some* (*pick A*) **unfolding** *lift-def* **by** *simp*
hence *lift pick binp′ i* = *Some* (*pick A*) **using** ∗ **by** *simp*
then obtain *A′* **where** *binp′*: *binp′ i* = *Some A′* **and** *AA′*: *pick A* = *pick A′*
unfolding *lift-def* **by**(*cases binp′ i*, *auto*)
hence *goodAbs A* ∧ *goodAbs A′*
using *binp good goodBinp-def liftAll-def* **by** (*metis* (*opaque-lifting*, *full-types*))
hence *A* = *A′* **using** *AA′* **by** *auto*
thus *?thesis* **unfolding** *binp binp′* **by** *simp*
qed
qed

lemma *goodBinp-imp-qGoodBinp-pickBinp*:
assumes *goodBinp binp*
shows *qGoodBinp* (*pickBinp binp*)
unfolding *pickBinp-def qGoodBinp-def liftAll-def* **proof** *safe*
fix *i qA* **assume** *lift pick binp i* = *Some qA*
then obtain *A* **where** *binp*: *binp i* = *Some A* **and** *qA*: *qA* = *pick A*
unfolding *lift-def* **by**(*cases binp i*, *auto*)
hence *goodAbs A* **using** *assms*
unfolding *goodBinp-def liftAll-def* **by** *simp*
thus *qGoodAbs qA* **unfolding** *qA* **using** *goodAbs-imp-qGoodAbs-pick* **by** *auto*
next
fix *xs* **let** *?Left* = {*i*. *lift pick binp i* ≠ *None*}
have *?Left* = {*i*. *binp i* ≠ *None*} **by**(*force simp add*: *lift-None*)
thus |*?Left*| <*o* |*UNIV* :: ′*var set*| **using** *assms* **unfolding** *goodBinp-def* **by** *auto*
qed

lemma *qGoodBinp-iff-goodBinp-asBinp*:
goodBinp (*asBinp qbinp*) = *qGoodBinp qbinp*
proof(*unfold asBinp-def*)
let *?binp* = *lift asAbs qbinp*
{**assume** *qgood-qbinp*: *qGoodBinp qbinp*
have *goodBinp ?binp*
unfolding *goodBinp-def liftAll-def* **proof** *safe*
fix *i A* **assume** *binp*: *?binp i* = *Some A*
then obtain *qA* **where** *qbinp*: *qbinp i* = *Some qA* **and** *A*: *A* = *asAbs qA*
unfolding *lift-def* **by**(*cases qbinp i*, *auto*)
hence *qGoodAbs qA*
using *qgood-qbinp* **unfolding** *qGoodBinp-def liftAll-def* **by** *auto*
thus *goodAbs A* **using** *A qGoodAbs-iff-goodAbs-asAbs* **by** *auto*
next
fix *xs* **let** *?Left* = {*i*. *lift asAbs qbinp i* ≠ *None*}
have *?Left* = {*i*. *qbinp i* ≠ *None*} **by**(*auto simp add*: *lift-None*)
thus |*?Left*| <*o* |*UNIV* :: ′*var set*| **using** *qgood-qbinp* **unfolding** *qGoodBinp-def*
by *auto*
qed

}
moreover
{**assume** *good-binp*: *goodBinp ?binp*
have *qGoodBinp qbinp*
unfolding *qGoodBinp-def liftAll-def* **proof** *safe*
fix *i qA* **assume** *qbinp*: *qbinp i = Some qA* **let** *?A = asAbs qA*
have *binp*: *?binp i = Some ?A* **unfolding** *lift-def* **using** *qbinp* **by** *simp*
hence *goodAbs ?A*
using *good-binp* **unfolding** *goodBinp-def liftAll-def* **by** *auto*
thus *qGoodAbs qA* **using** *qGoodAbs-iff-goodAbs-asAbs* **by** *auto*
next
fix *xs* **let** *?Left* = {*i. qbinp i ≠ None*}
have *?Left* = {*i. lift asAbs qbinp i ≠ None*} **by**(*auto simp add*: *lift-None*)
thus |*?Left*| <*o* |*UNIV* :: *'var set*| **using** *good-binp* **unfolding** *goodBinp-def* **by** *auto*
qed
}
ultimately show *goodBinp ?binp = qGoodBinp qbinp* **by** *blast*
qed

lemma *pickBinp-asBinp*:
assumes *qGoodBinp qbinp*
shows *pickBinp* (*asBinp qbinp*) %%= *qbinp*
unfolding *pickBinp-def asBinp-def lift-comp alphaBinp-def* **using** *sameDom-lift2*

by *auto* (*smt* (*verit*) *assms comp-apply liftAll2-def liftAll-def lift-def option.sel option.simps*(*5*) *pick-asAbs qGoodBinp-def*)

lemma *asBinp-pickBinp*:
assumes *goodBinp binp*
shows *asBinp* (*pickBinp binp*) = *binp*
unfolding *asBinp-def pickBinp-def lift-comp*
apply(*rule ext*)
subgoal for *i* **apply**(*cases binp i*)
using *assms asAbs-pick* **unfolding** *goodBinp-def liftAll-def lift-def* **by** *auto* **.**

lemma *pickBinp-alphaBinp*:
assumes *goodBinp*: *goodBinp binp*
shows *pickBinp binp* %%= *pickBinp binp*
using *assms goodBinp-imp-qGoodBinp-pickBinp alphaBinp-refl* **by** *auto*

lemma *alphaBinp-imp-asBinp-equal*:
assumes *qGoodBinp qbinp* **and** *qbinp* %%= *qbinp'*
shows *asBinp qbinp = asBinp qbinp'*
unfolding *asBinp-def* **proof**(*rule ext*)
fix *i* **show** *lift asAbs qbinp i = lift asAbs qbinp' i*
proof(*cases qbinp i*)
case *None*
hence *qbinp' i = None*

using *assms* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
thus *?thesis* **using** *None* **unfolding** *lift-def* **by** *simp*
next
case (*Some qA*)
then obtain *qA′* **where** *qbinp′*: *qbinp′ i* = *Some qA′*
using *assms* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** (*cases qbinp′ i*, *force*)
hence *qA* $= *qA′*
using *assms Some* **unfolding** *alphaBinp-def sameDom-def liftAll2-def* **by** *auto*
moreover have *qGoodAbs qA* **using** *assms Some* **unfolding** *qGoodBinp-def liftAll-def* **by** *auto*
ultimately show *?thesis*
using *Some qbinp′ alphaAbs-imp-asAbs-equal* **unfolding** *lift-def* **by** *auto*
qed
qed

lemma *asBinp-equal-imp-alphaBinp*:
assumes *qGoodBinp qbinp* **and** *asBinp qbinp* = *asBinp qbinp′*
shows *qbinp* %%= *qbinp′*
using *assms* **unfolding** *alphaBinp-def liftAll2-def sameDom-def*
by *simp* (*smt* (*verit*) *asAbs-equal-imp-alphaAbs asBinp-def liftAll-def lift-None lift-def option.inject option.simps*(*5*) *qGoodBinp-def*)

lemma *asBinp-equal-iff-alphaBinp*:
qGoodBinp qbinp ⟹ (*asBinp qbinp* = *asBinp qbinp′*) = (*qbinp* %%= *qbinp′*)
using *asBinp-equal-imp-alphaBinp alphaBinp-imp-asBinp-equal* **by** *blast*

lemma *pickBinp-alphaBinp-iff-equal*:
assumes *goodBinp binp* **and** *goodBinp binp′*
shows (*pickBinp binp* %%= *pickBinp binp′*) = (*binp* = *binp′*)
using *assms goodBinp-imp-qGoodBinp-pickBinp asBinp-pickBinp pickBinp-alphaBinp*

by (*metis asBinp-equal-iff-alphaBinp*)

lemma *pickBinp-swapBinp-qSwapBinp*:
assumes *goodBinp binp*
shows *pickBinp* (*binp* %%[*x1* ∧ *x2*]-*xs*) %%= ((*pickBinp binp*) %%[[*x1* ∧ *x2*]]-*xs*)
using *assms* **unfolding** *pickBinp-def swapBinp-def qSwapBinp-def lift-comp alphaBinp-def sameDom-def liftAll2-def*
by (*simp add*: *goodBinp-def liftAll-def lift-def option.case-eq-if pick-swapAbs-qSwapAbs*)

lemma *asBinp-qSwapBinp-swapBinp*:
assumes *qGoodBinp qbinp*
shows *asBinp* (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*) = ((*asBinp qbinp*) %%[*x1* ∧ *x2*]-*xs*)
unfolding *asBinp-def swapBinp-def qSwapBinp-def lift-comp alphaBinp-def lift-def*
apply(*rule ext*) **subgoal for** *i* **apply**(*cases qbinp i*)
using *assms asAbs-qSwapAbs-swapAbs* **by** (*fastforce simp add*: *liftAll-def qGood-Binp-def*)+ **.**

lemma *swapBinp-def2*:
(*binp* %%[*x1* ∧ *x2*]-*xs*) = *asBinp* ((*pickBinp binp*) %%[[*x1* ∧ *x2*]]-*xs*)
unfolding *swapBinp-def asBinp-def pickBinp-def qSwapBinp-def lift-def swapAbs-def*
apply (*rule ext*) **subgoal for** *i* **by** (*cases binp i*) *simp-all* **.**

lemma *freshBinp-def2*:
freshBinp xs x binp = *qFreshBinp xs x* (*pickBinp binp*)
unfolding *freshBinp-def qFreshBinp-def pickBinp-def lift-def freshAbs-def liftAll-def*
apply (*rule iff-allI*) **subgoal for** *i* **by** (*cases binp i*) *simp-all* **.**

5.3.4 For environments

lemma *goodEnv-imp-qGoodEnv-pickE*:
assumes *goodEnv rho*
shows *qGoodEnv* (*pickE rho*)
unfolding *qGoodEnv-def pickE-def*
apply(*auto simp del*: *not-None-eq*)
using *assms good-imp-qGood-pick* **unfolding** *liftAll-lift-comp comp-def*
by (*auto simp*: *goodEnv-def liftAll-def lift-None*)

lemma *qGoodEnv-iff-goodEnv-asEnv*:
goodEnv (*asEnv qrho*) = *qGoodEnv qrho*
unfolding *asEnv-def* **unfolding** *goodEnv-def liftAll-lift-comp comp-def*
by (*auto simp*: *qGoodEnv-def lift-None liftAll-def qGood-iff-good-asTerm*)

lemma *pickE-asEnv*:
assumes *qGoodEnv qrho*
shows *pickE* (*asEnv qrho*) &= *qrho*
using *assms*
by (*auto simp*: *lift-None liftAll-def lift-def alphaEnv-def sameDom-def liftAll2-def pick-asTerm qGoodEnv-def pickE-def asEnv-def split*: *option.splits*)

lemma *asEnv-pickE*:
assumes *goodEnv rho* **shows** *asEnv* (*pickE rho*) *xs x* = *rho xs x*
using *assms asTerm-pick*
by (*cases rho xs x*) (*auto simp*: *goodEnv-def liftAll-def asEnv-def pickE-def lift-comp lift-def*)

lemma *pickE-alphaEnv*:
assumes *goodEnv*: *goodEnv rho* **shows** *pickE rho* &= *pickE rho*
using *assms goodEnv-imp-qGoodEnv-pickE alphaEnv-refl* **by** *auto*

lemma *alphaEnv-imp-asEnv-equal*:
assumes *qGoodEnv qrho* **and** *qrho* &= *qrho'*
shows *asEnv qrho* = *asEnv qrho'*
apply (*rule ext*)+ **subgoal for** *xs x* **apply**(*cases qrho xs x*)
using *assms asTerm-equal-iff-alpha alpha-imp-asTerm-equal*
by (*auto simp add*: *alphaEnv-def sameDom-def asEnv-def lift-def*
qGoodEnv-def liftAll-def liftAll2-def option.case-eq-if split: *option.splits*)

blast+ **.**

lemma *asEnv-equal-imp-alphaEnv*:
assumes *qGoodEnv qrho* **and** *asEnv qrho = asEnv qrho′*
shows *qrho &= qrho′*
using *assms* **unfolding** *alphaEnv-def sameDom-def liftAll2-def*
apply (*simp add*: *asEnv-def lift-None lift-def qGoodEnv-def liftAll-def*)
by (*smt* (*verit*) *asTerm-equal-imp-alpha option.sel option.simps*(*5*) *option.case-eq-if option.distinct*(*1*))

lemma *asEnv-equal-iff-alphaEnv*:
qGoodEnv qrho ⟹ (*asEnv qrho = asEnv qrho′*) = (*qrho &= qrho′*)
using *asEnv-equal-imp-alphaEnv alphaEnv-imp-asEnv-equal* **by** *blast*

lemma *pickE-alphaEnv-iff-equal*:
assumes *goodEnv rho* **and** *goodEnv rho′*
shows (*pickE rho &= pickE rho′*) = (*rho = rho′*)
proof(*rule iffI*, *safe*, (*rule ext*)+)
fix *xs x*
assume *alpha*: *pickE rho &= pickE rho′*
have *qgood-rho*: *qGoodEnv* (*pickE rho*) **using** *assms goodEnv-imp-qGoodEnv-pickE* **by** *auto*
have *rho xs x = asEnv* (*pickE rho*) *xs x* **using** *assms asEnv-pickE* **by** *fastforce*
also have ... = *asEnv* (*pickE rho′*) *xs x*
using *qgood-rho alpha alphaEnv-imp-asEnv-equal* **by** *fastforce*
also have ... = *rho′ xs x* **using** *assms asEnv-pickE* **by** *fastforce*
finally show *rho xs x = rho′ xs x* **.**
next
have *qGoodEnv*(*pickE rho′*) **using** *assms goodEnv-imp-qGoodEnv-pickE* **by** *auto*
thus *pickE rho′ &= pickE rho′* **using** *alphaEnv-refl* **by** *auto*
qed

lemma *freshEnv-def2*:
freshEnv xs x rho = qFreshEnv xs x (*pickE rho*)
unfolding *freshEnv-def qFreshEnv-def pickE-def lift-def fresh-def liftAll-def*
apply(*cases rho xs x*)
by (*auto intro*!: *iff-allI*) (*metis map-option-case map-option-eq-Some*)

lemma *pick-psubst-qPsubst*:
assumes *good X* **and** *goodEnv rho*
shows *pick* (*X* #[*rho*]) #= ((*pick X*) #[[*pickE rho*]])
by (*simp add*: *assms goodEnv-imp-qGoodEnv-pickE good-imp-qGood-pick pick-asTerm psubst-def qPsubst-preserves-qGood*)

lemma *pick-psubstAbs-qPsubstAbs*:
assumes *goodAbs A* **and** *goodEnv rho*
shows *pick* (*A* $[*rho*]) $= ((*pick A*) $[[*pickE rho*]])
by (*simp add*: *assms goodAbs-imp-qGoodAbs-pick goodEnv-imp-qGoodEnv-pickE pick-asAbs*

psubstAbs-def qPsubstAbs-preserves-qGoodAbs)

lemma *pickInp-psubstInp-qPsubstInp*:
assumes *good*: *goodInp inp* **and** *good-rho*: *goodEnv rho*
shows *pickInp* (*inp* %[*rho*]) %= ((*pickInp inp*) %[[*pickE rho*]])
using *assms* **unfolding** *pickInp-def psubstInp-def qPsubstInp-def lift-comp*
unfolding *alphaInp-def sameDom-def liftAll2-def*
by (*simp add*: *lift-None*)
(*smt* (*verit*) *comp-apply goodEnv-imp-qGoodEnv-pickE goodInp-imp-qGoodInp-pickInp liftAll-def lift-def map-option-case map-option-eq-Some option.sel pickInp-def pick-asTerm psubst-def qGoodInp-def qPsubst-preserves-qGood*)

lemma *pickBinp-psubstBinp-qPsubstBinp*:
assumes *good*: *goodBinp binp* **and** *good-rho*: *goodEnv rho*
shows *pickBinp* (*binp* %%[*rho*]) %%= ((*pickBinp binp*) %%[[*pickE rho*]])
using *assms* **unfolding** *pickBinp-def psubstBinp-def qPsubstBinp-def lift-comp*
unfolding *alphaBinp-def sameDom-def liftAll2-def*
by (*simp add*: *lift-None*)
(*smt* (*verit*) *comp-apply goodBinp-def liftAll-def lift-def map-option-case map-option-eq-Some*

option.sel pick-psubstAbs-qPsubstAbs)

5.3.5 The structural alpha-equivPalence maps commute with the syntactic constructs

lemma *pick-Var-qVar*:
pick (*Var xs x*) #= *qVar xs x*
unfolding *Var-def* **using** *pick-asTerm* **by** *force*

lemma *Op-asInp-asTerm-qOp*:
assumes *qGoodInp qinp* **and** *qGoodBinp qbinp*
shows *Op delta* (*asInp qinp*) (*asBinp qbinp*) = *asTerm* (*qOp delta qinp qbinp*)
using *assms pickInp-asInp pickBinp-asBinp* **unfolding** *Op-def*
by(*auto simp add*: *asTerm-equal-iff-alpha*)

lemma *qOp-pickInp-pick-Op*:
assumes *goodInp inp* **and** *goodBinp binp*
shows *qOp delta* (*pickInp inp*) (*pickBinp binp*) #= *pick* (*Op delta inp binp*)
using *assms goodInp-imp-qGoodInp-pickInp goodBinp-imp-qGoodBinp-pickBinp*
unfolding *Op-def* **using** *pick-asTerm alpha-sym* **by** *force*

lemma *Abs-asTerm-asAbs-qAbs*:
assumes *qGood qX*
shows *Abs xs x* (*asTerm qX*) = *asAbs* (*qAbs xs x qX*)
using *assms pick-asTerm qAbs-preserves-alpha* **unfolding** *Abs-def*
by(*force simp add*: *asAbs-equal-iff-alphaAbs*)

lemma *qAbs-pick-Abs*:
assumes *good X*

shows *qAbs xs x* (*pick X*) $= *pick* (*Abs xs x X*)
using *assms good-imp-qGood-pick pick-asAbs alphaAbs-sym* **unfolding** *Abs-def*
by *force*

lemmas *qItem-versus-item-simps* =
univ-asTerm-alphaGood univ-asAbs-alphaAbsGood
univ-asTerm-alpha univ-asAbs-alphaAbs
pick-injective-good pick-injective-goodAbs

5.4 All operators preserve the "good" predicate

lemma *Var-preserves-good*[*simp*]:
good(*Var xs x*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term*)
by (*metis Var-def qGood.simps*(*1*) *qGood-iff-good-asTerm*)

lemma *Op-preserves-good*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows *good*(*Op delta inp binp*)
using *assms goodInp-imp-qGoodInp-pickInp goodBinp-imp-qGoodBinp-pickBinp*
qGood-iff-good-asTerm **unfolding** *Op-def* **by** *fastforce*

lemma *Abs-preserves-good*[*simp*]:
assumes *good*: *good X*
shows *goodAbs*(*Abs xs x X*)
using *assms good-imp-qGood-pick qGoodAbs-iff-goodAbs-asAbs*
unfolding *Abs-def* **by** *fastforce*

lemmas *Cons-preserve-good* =
Var-preserves-good Op-preserves-good Abs-preserves-good

lemma *swap-preserves-good*[*simp*]:
assumes *good X*
shows *good* (*X* #[*x* ∧ *y*]-*xs*)
using *assms good-imp-qGood-pick qSwap-preserves-qGood qGood-iff-good-asTerm*
unfolding *swap-def* **by** *fastforce*

lemma *swapAbs-preserves-good*[*simp*]:
assumes *goodAbs A*
shows *goodAbs* (*A* $[*x* ∧ *y*]-*xs*)
using *assms goodAbs-imp-qGoodAbs-pick qSwapAbs-preserves-qGoodAbs qGoodAbs-iff-goodAbs-asAbs*

unfolding *swapAbs-def* **by** *fastforce*

lemma *swapInp-preserves-good*[*simp*]:
assumes *goodInp inp*
shows *goodInp* (*inp* %[*x* ∧ *y*]-*xs*)
using *assms*
by (*auto simp*: *goodInp-def lift-def swapInp-def liftAll-def split*: *option.splits*)

lemma *swapBinp-preserves-good*[*simp*]:
assumes *goodBinp binp*
shows *goodBinp* (*binp* %%[*x* ∧ *y*]-*xs*)
using *assms*
by (*auto simp*: *goodBinp-def lift-def swapBinp-def liftAll-def split*: *option.splits*)

lemma *swapEnvDom-preserves-good*:
assumes *goodEnv rho*
shows *goodEnv* (*swapEnvDom xs x y rho*) (**is** *goodEnv ?rho′*)
unfolding *goodEnv-def liftAll-def* **proof** *safe*
 fix *zs z X′* **assume** *rho′*: *?rho′ zs z* = *Some X′*
 hence *rho zs* (*z* @*zs*[*x* ∧ *y*]-*xs*) = *Some X′* **unfolding** *swapEnvDom-def* **by** *simp*
 thus *good X′* **using** *assms* **unfolding** *goodEnv-def liftAll-def* **by** *simp*
next
 fix *xsa ys* **let** *?Left* = {*ya*. *?rho′ ys ya* ≠ *None*}
 have |{*y*} ∪ {*ya*. *rho ys ya* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|
 using *assms var-infinite-INNER card-of-Un-singl-ordLess-infinite*
 unfolding *goodEnv-def* **by** *fastforce*
 hence |{*x*,*y*} ∪ {*ya*. *rho ys ya* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|
 using *var-infinite-INNER card-of-Un-singl-ordLess-infinite* **by** *fastforce*
 moreover
 {have *?Left* ⊆ {*x*,*y*} ∪ {*ya*. *rho ys ya* ≠ *None*}
  unfolding *swapEnvDom-def sw-def*[*abs-def*] **by** *auto*
  hence |*?Left*| ≤*o* |{*x*,*y*} ∪ {*ya*. *rho ys ya* ≠ *None*}|
  using *card-of-mono1* **by** *auto*
 }
 ultimately show |*?Left*| <*o* |*UNIV* :: ′*var set*| **using** *ordLeq-ordLess-trans* **by** *blast*
qed

lemma *swapEnvIm-preserves-good*:
assumes *goodEnv rho*
shows *goodEnv* (*swapEnvIm xs x y rho*)
using *assms* **unfolding** *goodEnv-def swapEnvIm-def liftAll-def*
by (*auto simp*: *lift-def split*: *option.splits*)

lemma *swapEnv-preserves-good*[*simp*]:
assumes *goodEnv rho*
shows *goodEnv* (*rho* &[*x* ∧ *y*]-*xs*)
unfolding *swapEnv-def comp-def*
using *assms* **by**(*auto simp add*: *swapEnvDom-preserves-good swapEnvIm-preserves-good*)

lemmas *swapAll-preserve-good* =
swap-preserves-good swapAbs-preserves-good
swapInp-preserves-good swapBinp-preserves-good
swapEnv-preserves-good

lemma *psubst-preserves-good*[*simp*]:
assumes *goodEnv rho* **and** *good X*

shows *good* (*X* #[*rho*])
using *assms good-imp-qGood-pick goodEnv-imp-qGoodEnv-pickE*
qPsubst-preserves-qGood qGood-iff-good-asTerm **unfolding** *psubst-def* **by** *fastforce*

lemma *psubstAbs-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *goodAbs-A*: *goodAbs A*
shows *goodAbs* (*A* $[*rho*])
using *assms goodAbs-A goodAbs-imp-qGoodAbs-pick goodEnv-imp-qGoodEnv-pickE*

qPsubstAbs-preserves-qGoodAbs qGoodAbs-iff-goodAbs-asAbs **unfolding** *psubstAbs-def*
by *fastforce*

lemma *psubstInp-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *good*: *goodInp inp*
shows *goodInp* (*inp* %[*rho*])
using *assms* **unfolding** *goodInp-def psubstInp-def liftAll-def*
by (*auto simp add*: *lift-def split*: *option.splits*)

lemma *psubstBinp-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *good*: *goodBinp binp*
shows *goodBinp* (*binp* %%[*rho*])
using *assms* **unfolding** *goodBinp-def psubstBinp-def liftAll-def*
by (*auto simp add*: *lift-def split*: *option.splits*)

lemma *psubstEnv-preserves-good*[*simp*]:
assumes *good*: *goodEnv rho* **and** *good'*: *goodEnv rho'*
shows *goodEnv* (*rho* &[*rho'*])
unfolding *goodEnv-def liftAll-def*
proof *safe*
  fix *zs z X'*
  assume ∗: (*rho* &[*rho'*]) *zs z* = *Some X'*
  show *good X'*
  proof(*cases rho zs z*)
    case *None*
    hence *rho' zs z* = *Some X'* **using** ∗ **unfolding** *psubstEnv-def* **by** *auto*
    thus *?thesis* **using** *good'* **unfolding** *goodEnv-def liftAll-def* **by** *auto*
  next
    case (*Some X*)
    hence *X'* = (*X* #[*rho'*]) **using** ∗ **unfolding** *psubstEnv-def* **by** *auto*
     moreover have *good X* **using** *Some good* **unfolding** *goodEnv-def liftAll-def*
by *auto*
    ultimately show *?thesis* **using** *good' psubst-preserves-good* **by** *auto*
  qed
next
  fix *xs ys* **let** *?Left* = {*y*. (*rho* &[*rho'*]) *ys y* ≠ *None*}
  let *?Left1* = {*y*. *rho ys y* ≠ *None*} **let** *?Left2* = {*y*. *rho' ys y* ≠ *None*}
  have |*?Left1*| <*o* |*UNIV* :: *'var set*| ∧ |*?Left2*| <*o* |*UNIV* :: *'var set*|
  using *good good'* **unfolding** *goodEnv-def* **by** *simp*
  hence |*?Left1* ∪ *?Left2*| <*o* |*UNIV* :: *'var set*|

using *var-infinite-INNER card-of-Un-ordLess-infinite* **by** *auto*
moreover
{**have** *?Left* ⊆ *?Left1* ∪ *?Left2* **unfolding** *psubstEnv-def* **by** *auto*
hence |*?Left*| ≤*o* |*?Left1* ∪ *?Left2*| **using** *card-of-mono1* **by** *auto*
}
ultimately show |*?Left*| <*o* |*UNIV* :: *'var set*| **using** *ordLeq-ordLess-trans* **by** *blast*
qed

lemmas *psubstAll-preserve-good* =
psubst-preserves-good psubstAbs-preserves-good
psubstInp-preserves-good psubstBinp-preserves-good
psubstEnv-preserves-good

lemma *idEnv-preserves-good*[*simp*]: *goodEnv idEnv*
unfolding *goodEnv-def idEnv-def liftAll-def*
using *var-infinite-INNER finite-ordLess-infinite2* **by** *auto*

lemma *updEnv-preserves-good*[*simp*]:
assumes *good-X*: *good X* **and** *good-rho*: *goodEnv rho*
shows *goodEnv* (*rho* [*x* ← *X*]-*xs*)
using *assms* **unfolding** *updEnv-def goodEnv-def liftAll-def*
proof *safe*
fix *ys y Y*
assume *good X* **and** ∀ *ys y Y*. *rho ys y* = *Some Y* ⟶ *good Y*
and (*if ys* = *xs* ∧ *y* = *x then Some X else rho ys y*) = *Some Y*
thus *good Y*
by(*cases ys* = *xs* ∧ *y* = *x*) *auto*
next
fix *ys*
let *?V'* = {*y*. (*if ys* = *xs* ∧ *y* = *x then Some X else rho ys y*) ≠ *None*}
let *?V* = λ *ys*. {*y*. *rho ys y* ≠ *None*}
assume ∀ *ys*. |*?V ys*| <*o* |*UNIV* :: *'var set*|
hence |{*x*} ∪ *?V ys*| <*o* |*UNIV* :: *'var set*|
using *var-infinite-INNER card-of-Un-singl-ordLess-infinite* **by** *fastforce*
moreover
{**have** *?V'* ⊆ {*x*} ∪ *?V ys* **by** *auto*
hence |*?V'*| ≤*o* |{*x*} ∪ *?V ys*| **using** *card-of-mono1* **by** *auto*
}
ultimately show |*?V'*| <*o* |*UNIV* :: *'var set*| **using** *ordLeq-ordLess-trans* **by** *blast*
qed

lemma *getEnv-preserves-good*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *Some X*
shows *good X*
using *assms* **unfolding** *goodEnv-def liftAll-def* **by** *simp*

lemmas *envOps-preserve-good* =

idEnv-preserves-good updEnv-preserves-good
getEnv-preserves-good

lemma *subst-preserves-good*[*simp*]:
assumes *good X* **and** *good Y*
shows *good* (*Y* #[*X* / *x*]-*xs*)
unfolding *subst-def*
using *assms* **by** *simp*

lemma *substAbs-preserves-good*[*simp*]:
assumes *good X* **and** *goodAbs A*
shows *goodAbs* (*A* $[*X* / *x*]-*xs*)
unfolding *substAbs-def*
using *assms* **by** *simp*

lemma *substInp-preserves-good*[*simp*]:
assumes *good X* **and** *goodInp inp*
shows *goodInp* (*inp* %[*X* / *x*]-*xs*)
unfolding *substInp-def* **using** *assms* **by** *simp*

lemma *substBinp-preserves-good*[*simp*]:
assumes *good X* **and** *goodBinp binp*
shows *goodBinp* (*binp* %%[*X* / *x*]-*xs*)
unfolding *substBinp-def* **using** *assms* **by** *simp*

lemma *substEnv-preserves-good*[*simp*]:
assumes *good X* **and** *goodEnv rho*
shows *goodEnv* (*rho* &[*X* / *x*]-*xs*)
unfolding *substEnv-def* **using** *assms* **by** *simp*

lemmas *substAll-preserve-good* =
subst-preserves-good substAbs-preserves-good
substInp-preserves-good substBinp-preserves-good
substEnv-preserves-good

lemma *vsubst-preserves-good*[*simp*]:
assumes *good Y*
shows *good* (*Y* #[*x1* // *x*]-*xs*)
unfolding *vsubst-def* **using** *assms* **by** *simp*

lemma *vsubstAbs-preserves-good*[*simp*]:
assumes *goodAbs A*
shows *goodAbs* (*A* $[*x1* // *x*]-*xs*)
unfolding *vsubstAbs-def* **using** *assms* **by** *simp*

lemma *vsubstInp-preserves-good*[*simp*]:
assumes *goodInp inp*
shows *goodInp* (*inp* %[*x1* // *x*]-*xs*)
unfolding *vsubstInp-def* **using** *assms* **by** *simp*

lemma *vsubstBinp-preserves-good*[*simp*]:
assumes *goodBinp binp*
shows *goodBinp* (*binp* %%[*x1* // *x*]-*xs*)
unfolding *vsubstBinp-def* **using** *assms* **by** *simp*

lemma *vsubstEnv-preserves-good*[*simp*]:
assumes *goodEnv rho*
shows *goodEnv* (*rho* &[*x1* // *x*]-*xs*)
unfolding *vsubstEnv-def* **using** *assms* **by** *simp*

lemmas *vsubstAll-preserve-good* =
vsubst-preserves-good vsubstAbs-preserves-good
vsubstInp-preserves-good vsubstBinp-preserves-good
vsubstEnv-preserves-good

lemmas *all-preserve-good* =
Cons-preserve-good
swapAll-preserve-good
psubstAll-preserve-good
envOps-preserve-good
substAll-preserve-good
vsubstAll-preserve-good

5.4.1 The syntactic operators are almost constructors

The only one that does not act precisely like a constructor is "Abs".

theorem *Var-inj*[*simp*]:
(((*Var xs x*)::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term*) = *Var ys y*) =
 (*xs* = *ys* ∧ *x* = *y*)
by (*metis alpha-qVar-iff pick-Var-qVar qTerm.inject*)

lemma *Op-inj*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
and *goodInp inp'* **and** *goodBinp binp'*
shows
(*Op delta inp binp* = *Op delta' inp' binp'*) =
 (*delta* = *delta'* ∧ *inp* = *inp'* ∧ *binp* = *binp'*)
using *assms pickInp-alphaInp-iff-equal pickBinp-alphaBinp-iff-equal*
goodInp-imp-qGoodInp-pickInp goodBinp-imp-qGoodBinp-pickBinp
unfolding *Op-def* **by** (*fastforce simp*: *asTerm-equal-iff-alpha*)

"Abs" is almost injective ("ainj"), with almost injectivity expressed in two ways:
- maximally, using "forall" – this is suitable for elimination of "Abs" equalities;
- minimally, using "exists" – this is suitable for introduction of "Abs" equalities.

lemma *Abs-ainj-all*:
assumes *good*: *good X* **and** *good'*: *good X'*
shows
$(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee fresh\ xs\ y\ X) \wedge (y = x' \vee fresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[y \wedge x]\text{-}xs) = (X'\ \#[y \wedge x']\text{-}xs)))$
proof−
let *?qX = pick X* **let** *?qX' = pick X'*
have *qgood*: $qGood\ ?qX \wedge qGood\ ?qX'$ **using** *good good' good-imp-qGood-pick* **by** *auto*
hence *qgood-qXyx*: $\forall\ y.\ qGood\ (?qX\ \#[[y \wedge x]]\text{-}xs)$
using *qSwap-preserves-qGood* **by** *auto*
have *qGoodAbs*(*qAbs xs x ?qX*) **using** *qgood* **by** *simp*
hence $(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') = (qAbs\ xs\ x\ ?qX\ \$= qAbs\ xs'\ x'\ ?qX')$
unfolding *Abs-def* **by** (*auto simp add*: *asAbs-equal-iff-alphaAbs*)
also
have $\ldots = (xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qFresh\ xs\ y\ ?qX) \wedge (y = x' \vee qFresh\ xs\ y\ ?qX') \longrightarrow$
$(?qX\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (?qX'\ \#[[y \wedge x']]\text{-}xs)))$
using *qgood alphaAbs-qAbs-iff-all-equal-or-qFresh*[*of ?qX ?qX'*] **by** *blast*
also
have $\ldots = (xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee fresh\ xs\ y\ X) \wedge (y = x' \vee fresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[y \wedge x]\text{-}xs) = (X'\ \#[y \wedge x']\text{-}xs)))$
unfolding *fresh-def swap-def* **using** *qgood-qXyx* **by** (*auto simp add*: *asTerm-equal-iff-alpha*)
finally show *?thesis* **.**
qed

lemma *Abs-ainj-ex*:
assumes *good*: *good X* **and** *good'*: *good X'*
shows
$(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ y \notin \{x,x'\} \wedge fresh\ xs\ y\ X \wedge fresh\ xs\ y\ X' \wedge$
$(X\ \#[y \wedge x]\text{-}xs) = (X'\ \#[y \wedge x']\text{-}xs)))$
proof−
let *?qX = pick X* **let** *?qX' = pick X'*
have *qgood*: $qGood\ ?qX \wedge qGood\ ?qX'$ **using** *good good' good-imp-qGood-pick* **by** *auto*
hence *qgood-qXyx*: $\forall\ y.\ qGood\ (?qX\ \#[[y \wedge x]]\text{-}xs)$
using *qSwap-preserves-qGood* **by** *auto*
have *qGoodAbs*(*qAbs xs x ?qX*) **using** *qgood* **by** *simp*
hence $(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') = (qAbs\ xs\ x\ ?qX\ \$= qAbs\ xs'\ x'\ ?qX')$
unfolding *Abs-def* **by** (*auto simp add*: *asAbs-equal-iff-alphaAbs*)
also
have $\ldots = (xs = xs' \wedge$
$(\exists\ y.\ y \notin \{x,x'\} \wedge qFresh\ xs\ y\ ?qX \wedge qFresh\ xs\ y\ ?qX' \wedge$
$(?qX\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (?qX'\ \#[[y \wedge x']]\text{-}xs)))$

using *qgood alphaAbs-qAbs-iff-ex-distinct-qFresh*[*of ?qX xs x xs′ x′ ?qX′*] **by** *blast*
also
have . . . = (*xs* = *xs′* ∧
(∃ *y*. *y* ∉ {*x*,*x′*} ∧ *fresh xs y X* ∧ *fresh xs y X′* ∧
(*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)))
unfolding *fresh-def swap-def* **using** *qgood-qXyx asTerm-equal-iff-alpha* **by** *auto*
finally show *?thesis* **.**
qed

lemma *Abs-cong*[*fundef-cong*]:
assumes *good*: *good X* **and** *good′*: *good X′*
and *y*: *fresh xs y X* **and** *y′*: *fresh xs y X′*
and *eq*: (*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)
shows *Abs xs x X* = *Abs xs x′ X′*
proof−
let *?qX* = *pick X* **let** *?qX′* = *pick X′*
have *qgood*: *qGood ?qX* ∧ *qGood ?qX′* **using** *good good′ good-imp-qGood-pick* **by** *auto*
hence *qgood-qXyx*: ∀ *y*. *qGood* (*?qX* #[[*y* ∧ *x*]]-*xs*)
using *qSwap-preserves-qGood* **by** *auto*
have *qEq*: (*?qX* #[[*y* ∧ *x*]]-*xs*) #= (*?qX′* #[[*y* ∧ *x′*]]-*xs*)
using *eq* **unfolding** *fresh-def swap-def*
using *qgood-qXyx asTerm-equal-iff-alpha* **by** *auto*
have (*qAbs xs x ?qX* $= *qAbs xs x′ ?qX′*)
apply(*rule alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs*)
using *qgood* **apply** *simp*
unfolding *alphaAbs-ex-equal-or-qFresh-def* **using** *y y′ qEq*
unfolding *fresh-def* **by** *auto*
moreover have *qGoodAbs*(*qAbs xs x ?qX*) **using** *qgood* **by** *simp*
ultimately show *Abs xs x X* = *Abs xs x′ X′*
unfolding *Abs-def* **by** (*auto simp add*: *asAbs-equal-iff-alphaAbs*)
qed

lemma *Abs-swap-fresh*:
assumes *good-X*: *good X* **and** *fresh*: *fresh xs x′ X*
shows *Abs xs x X* = *Abs xs x′* (*X* #[*x′* ∧ *x*]-*xs*)
proof−
let *?x′x* = *swap xs x′ x* **let** *?qx′x* = *qSwap xs x′ x*
have *good-pickX*: *qGood* (*pick X*) **using** *good-X good-imp-qGood-pick* **by** *auto*
hence *good-qAbs-pickX*: *qGoodAbs* (*qAbs xs x* (*pick X*)) **by** *simp*
have *good-x′x-pickX*: *qGood* (*?qx′x* (*pick X*))
using *good-pickX qSwap-preserves-qGood* **by** *auto*

have *Abs xs x X* = *asAbs* (*qAbs xs x* (*pick X*)) **unfolding** *Abs-def* **by** *simp*
also
{**have** *qAbs xs x* (*pick X*) $= *qAbs xs x′* (*?qx′x* (*pick X*))
using *good-pickX fresh* **unfolding** *fresh-def* **using** *qAbs-alphaAbs-qSwap-qFresh* **by** *fastforce*
moreover

{**have** *?qx'x* (*pick X*) #= *pick* (*?x'x X*)
using *good-X* **by** (*auto simp add*: *pick-swap-qSwap alpha-sym*)
hence *qAbs xs x'* (*?qx'x* (*pick X*)) $= *qAbs xs x'* (*pick* (*?x'x X*))
using *good-x'x-pickX qAbs-preserves-alpha* **by** *fastforce*
}
ultimately have *qAbs xs x* (*pick X*) $= *qAbs xs x'* (*pick* (*?x'x X*))
using *good-qAbs-pickX alphaAbs-trans* **by** *blast*
hence *asAbs* (*qAbs xs x* (*pick X*)) = *asAbs* (*qAbs xs x'* (*pick* (*?x'x X*)))
using *good-qAbs-pickX* **by** (*auto simp add*: *asAbs-equal-iff-alphaAbs*)
}
also have *asAbs* (*qAbs xs x'* (*pick* (*?x'x X*))) = *Abs xs x'* (*?x'x X*)
unfolding *Abs-def* **by** *auto*
finally show *?thesis* **.**
qed

lemma *Var-diff-Op*[*simp*]:
Var xs x ≠ *Op delta inp binp*
by (*simp add*: *Op-def Var-def asTerm-equal-iff-alpha*)

lemma *Op-diff-Var*[*simp*]:
Op delta inp binp ≠ *Var xs x*
using *Var-diff-Op*[*of - - - inp*] **by** *blast*

theorem *term-nchotomy*:
assumes *good X*
shows
(∃ *xs x*. *X* = *Var xs x*) ∨
(∃ *delta inp binp*. *goodInp inp* ∧ *goodBinp binp* ∧ *X* = *Op delta inp binp*)
proof−
let *?qX* = *pick X*
have *good-qX*: *qGood ?qX* **using** *assms good-imp-qGood-pick* **by** *auto*
have *X*: *X* = *asTerm ?qX* **using** *assms asTerm-pick* **by** *auto*
show *?thesis*
proof(*cases ?qX*)
fix *xs x* **assume** *Case1*: *?qX* = *qVar xs x*
have *X* = *Var xs x* **unfolding** *Var-def* **using** *X Case1* **by** *simp*
thus *?thesis* **by** *blast*
next
fix *delta qinp qbinp* **assume** *Case2*: *?qX* = *qOp delta qinp qbinp*
hence *good-qinp*: *qGoodInp qinp* ∧ *qGoodBinp qbinp* **using** *good-qX* **by** *simp*
let *?inp* = *asInp qinp* **let** *?binp* = *asBinp qbinp*
have *qinp* %= *pickInp ?inp* ∧ *qbinp* %%= *pickBinp ?binp*
using *good-qinp pickInp-asInp alphaInp-sym pickBinp-asBinp alphaBinp-sym*
by *blast*
hence *qOp delta qinp qbinp* #= *qOp delta* (*pickInp ?inp*) (*pickBinp ?binp*) **by**
simp
hence *asTerm* (*qOp delta qinp qbinp*) = *Op delta ?inp ?binp*
unfolding *Op-def* **using** *Case2 good-qX* **by** (*auto simp add*: *asTerm-equal-iff-alpha*)
hence *X* = *Op delta ?inp ?binp* **using** *X Case2* **by** *auto*

moreover have *goodInp ?inp* ∧ *goodBinp ?binp*
using *good-qinp qGoodInp-iff-goodInp-asInp qGoodBinp-iff-goodBinp-asBinp* **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
qed

theorem *abs-nchotomy*:
assumes *goodAbs A*
shows ∃ *xs x X. good X* ∧ *A* = *Abs xs x X*
by (*metis Abs-asTerm-asAbs-qAbs asAbs-pick assms goodAbs-imp-qGoodAbs-pick qGoodAbs.elims*(*2*) *qGood-iff-good-asTerm*)

lemmas *good-freeCons* =
Op-inj Var-diff-Op Op-diff-Var

5.5 Properties lifted from quasi-terms to terms

5.5.1 Simplification rules

theorem *swap-Var-simp*[*simp*]:
((*Var xs x*) #[*y1* ∧ *y2*]-*ys*) = *Var xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*)
by (*metis QuasiTerms-Swap-Fresh.qSwapAll-simps*(*1*) *Var-def asTerm-qSwap-swap qItem-simps*(*9*))

lemma *swap-Op-simp*[*simp*]:
assumes *goodInp inp goodBinp binp*
shows ((*Op delta inp binp*) #[*x1* ∧ *x2*]-*xs*) =
Op delta (*inp* %[*x1* ∧ *x2*]-*xs*) (*binp* %%[*x1* ∧ *x2*]-*xs*)
by (*metis Op-asInp-asTerm-qOp Op-def asTerm-qSwap-swap assms*(*1*) *assms*(*2*) *goodBinp-imp-qGoodBinp-pickBinp goodInp-imp-qGoodInp-pickInp qGood-qGoodInp qSwapBinp-preserves-qGoodBinp qSwapInp-preserves-qGoodInp qSwap-qSwapInp swapBinp-def2 swapInp-def2*)

lemma *swapAbs-simp*[*simp*]:
assumes *good X*
shows ((*Abs xs x X*) $[*y1* ∧ *y2*]-*ys*) = *Abs xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*) (*X* #[*y1* ∧ *y2*]-*ys*)
by (*metis Abs-asTerm-asAbs-qAbs Abs-preserves-good alphaAbs-preserves-qGoodAbs2 asAbs-qSwapAbs-swapAbs assms goodAbs-imp-qGoodAbs-pick good-imp-qGood-pick local.Abs-def local.swap-def qAbs-pick-Abs qSwapAbs.simps qSwap-preserves-qGood1*)

lemmas *good-swapAll-simps* =
swap-Op-simp swapAbs-simp

theorem *fresh-Var-simp*[*simp*]:
fresh ys y (*Var xs x* :: (*'index,'bindex,'varSort,'var,'opSym*)*term*) ⟷
(*ys* ≠ *xs* ∨ *y* ≠ *x*)
by (*simp add*: *Var-def fresh-asTerm-qFresh*)

lemma *fresh-Op-simp*[*simp*]:
assumes *goodInp inp goodBinp binp*
shows
fresh xs x (*Op delta inp binp*) ⟷
(*freshInp xs x inp* ∧ *freshBinp xs x binp*)
by (*metis Op-def Op-preserves-good assms*(*1*) *assms*(*2*) *freshBinp-def2 freshInp-def2 fresh-asTerm-qFresh qFresh-qFreshInp qGood-iff-good-asTerm*)

lemma *freshAbs-simp*[*simp*]:
assumes *good X*
shows *freshAbs ys y* (*Abs xs x X*) ⟷ (*ys* = *xs* ∧ *y* = *x* ∨ *fresh ys y X*)
proof−
  let *?fr* = *fresh ys y* **let** *?qfr* = *qFresh ys y*
  let *?frA* = *freshAbs ys y* **let** *?qfrA* = *qFreshAbs ys y*
  have *qGood* (*pick X*) **using** *assms* **by**(*auto simp add*: *good-imp-qGood-pick*)
  hence *good-qAbs-pick-X*: *qGoodAbs* (*qAbs xs x* (*pick X*))
  using *assms good-imp-qGood-pick* **by** *auto*

  have *?frA* (*Abs xs x X*) = *?qfrA* ((*pick o asAbs*) (*qAbs xs x* (*pick X*)))
  unfolding *freshAbs-def Abs-def* **by** *simp*
  also
  {have (*pick o asAbs*) (*qAbs xs x* (*pick X*)) $= *qAbs xs x* (*pick X*)
   using *good-qAbs-pick-X pick-asAbs* **by** *fastforce*
   hence *?qfrA* ((*pick o asAbs*) (*qAbs xs x* (*pick X*))) = *?qfrA* (*qAbs xs x* (*pick X*))
   using *good-qAbs-pick-X qFreshAbs-preserves-alphaAbs* **by** *blast*
  }
  also have *?qfrA*(*qAbs xs x* (*pick X*)) = (*ys* = *xs* ∧ *y* = *x* ∨ *?qfr* (*pick X*)) **by** *simp*
  also have ... = (*ys* = *xs* ∧ *y* = *x* ∨ *?fr X*) **unfolding** *fresh-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemmas *good-freshAll-simps* =
fresh-Op-simp freshAbs-simp

theorem *skel-Var-simp*[*simp*]:
skel (*Var xs x*) = *Branch Map.empty Map.empty*
by (*metis alpha-qSkel pick-Var-qVar qSkel.simps*(*1*) *skel-def*)

lemma *skel-Op-simp*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows *skel* (*Op delta inp binp*) = *Branch* (*skelInp inp*) (*skelBinp binp*)
by (*metis* (*no-types*, *lifting*) *alpha-qSkel assms*
   qOp-pickInp-pick-Op qSkel-qSkelInp skelBinp-def skelInp-def skel-def)

lemma *skelAbs-simp*[*simp*]:
assumes *good X*
shows *skelAbs* (*Abs xs x X*) = *Branch* (*λi. Some* (*skel X*)) *Map.empty*

```
by (metis alphaAll-qSkelAll assms qAbs-pick-Abs qSkelAbs.simps skelAbs-def skel-def)

lemmas good-skelAll-simps =
skel-Op-simp skelAbs-simp

lemma psubst-Var:
assumes goodEnv rho
shows ((Var xs x) #[rho]) =
        (case rho xs x of None ⇒ Var xs x
                          |Some X ⇒ X)
proof−
  let ?X = Var xs x  let ?qX = qVar xs x
  let ?qrho = pickE rho
  have good-qX: qGood ?qX using assms by simp
 moreover have good-qrho: qGoodEnv ?qrho using assms goodEnv-imp-qGoodEnv-pickE
by auto
  ultimately have good-qXrho: qGood (?qX #[[?qrho]])
   using assms qPsubst-preserves-qGood by(auto simp del: qGoodAll-simps qP-
subst.simps)

  have (?X #[rho]) = asTerm ((pick (asTerm ?qX)) #[[?qrho]])
  unfolding Var-def psubst-def by simp
  also
  {have ?qX  #= pick (asTerm ?qX) using good-qX pick-asTerm alpha-sym by
fastforce
   hence (?qX #[[?qrho]]) #= ((pick (asTerm ?qX)) #[[?qrho]])
   using good-qrho good-qX qPsubst-preserves-alpha1[of - ?qX] by fastforce
   hence asTerm ((pick (asTerm ?qX))  #[[?qrho]]) = asTerm (?qX #[[?qrho]])
   using good-qXrho asTerm-equal-iff-alpha[of ?qX #[[?qrho]]] by blast
  }
  also have asTerm (?qX #[[?qrho]]) =
            asTerm (case ?qrho xs x of None ⇒ qVar xs x
                                    |Some qY ⇒ qY) unfolding Var-def by simp
  finally have 1: (?X #[rho]) =  asTerm (case ?qrho xs x of None ⇒ qVar xs x
                                                |Some qY ⇒ qY) .
  show ?thesis
  proof(cases rho xs x)
    assume Case1: rho xs x = None
    hence ?qrho xs x = None unfolding pickE-def lift-def by simp
    thus ?thesis using 1 Case1 unfolding Var-def by simp
  next
    fix X assume Case2: rho xs x = Some X
    hence good X using assms unfolding goodEnv-def liftAll-def by auto
    hence asTerm (pick X) = X using asTerm-pick by auto
    moreover have qrho: ?qrho xs x = Some (pick X)
    using Case2 unfolding pickE-def lift-def by simp
    ultimately show ?thesis using 1 Case2 unfolding Var-def by simp
  qed
qed
```

corollary *psubst-Var-simp1*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *None*
shows ((*Var xs x*) #[*rho*]) = *Var xs x*
using *assms* **by**(*simp add*: *psubst-Var*)

corollary *psubst-Var-simp2*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *Some X*
shows ((*Var xs x*) #[*rho*]) = *X*
using *assms* **by**(*simp add*: *psubst-Var*)

lemma *psubst-Op-simp*[*simp*]:
assumes *good-inp*: *goodInp inp goodBinp binp*
and *good-rho*: *goodEnv rho*
shows
((*Op delta inp binp*) #[*rho*]) = *Op delta* (*inp* %[*rho*]) (*binp* %%[*rho*])
proof−
  let *?qrho* = *pickE rho*
  let *?sbs* = *psubst rho* **let** *?qsbs* = *qPsubst ?qrho*
  let *?sbsI* = *psubstInp rho* **let** *?qsbsI* = *qPsubstInp ?qrho*
  let *?sbsB* = *psubstBinp rho* **let** *?qsbsB* = *qPsubstBinp ?qrho*
  let *?op* = *Op delta* **let** *?qop* = *qOp delta*
  have *good-qop-pickInp-inp*: *qGood* (*?qop* (*pickInp inp*) (*pickBinp binp*))
  using *good-inp goodInp-imp-qGoodInp-pickInp*
        goodBinp-imp-qGoodBinp-pickBinp **by** *auto*
  hence *qGood* ((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*)))
  using *good-imp-qGood-pick qGood-iff-good-asTerm* **by** *fastforce*
  moreover have *good-qrho*: *qGoodEnv ?qrho*
  using *good-rho goodEnv-imp-qGoodEnv-pickE* **by** *auto*
  ultimately have *good*: *qGood* (*?qsbs*((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*))))
  using *qPsubst-preserves-qGood* **by** *auto*

  have *?sbs* (*?op inp binp*) =
     asTerm (*?qsbs* ((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*))))
  unfolding *psubst-def Op-def* **by** *simp*
  also
  {**have** (*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*)) #=
      ?qop (*pickInp inp*) (*pickBinp binp*)
   using *good-qop-pickInp-inp pick-asTerm* **by** *fastforce*
   hence *?qsbs*((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*))) #=
      ?qsbs(*?qop* (*pickInp inp*) (*pickBinp binp*))
   using *good-qop-pickInp-inp good-qrho qPsubst-preserves-alpha1* **by** *fastforce*
   moreover have *?qsbs* (*?qop* (*pickInp inp*) (*pickBinp binp*)) =
        ?qop (*?qsbsI* (*pickInp inp*)) (*?qsbsB* (*pickBinp binp*)) **by** *simp*
   moreover
   {**have** *?qsbsI* (*pickInp inp*) %= *pickInp* (*?sbsI inp*) ∧
      ?qsbsB (*pickBinp binp*) %%= *pickBinp* (*?sbsB binp*)
    using *good-rho good-inp pickInp-psubstInp-qPsubstInp*[*of inp rho*]

pickBinp-psubstBinp-qPsubstBinp[*of binp rho*] *alphaInp-sym alphaBinp-sym*
by *auto*
hence *?qop* (*?qsbsI* (*pickInp inp*)) (*?qsbsB* (*pickBinp binp*)) *#=*
?qop (*pickInp* (*?sbsI inp*)) (*pickBinp* (*?sbsB binp*)) **by** *simp*
}
ultimately have *?qsbs*((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*)))
#=
?qop (*pickInp* (*?sbsI inp*)) (*pickBinp* (*?sbsB binp*))
using *good alpha-trans* **by** *force*
hence *asTerm* (*?qsbs*((*pick o asTerm*) (*?qop* (*pickInp inp*) (*pickBinp binp*)))) =
asTerm (*?qop* (*pickInp* (*?sbsI inp*)) (*pickBinp* (*?sbsB binp*)))
using *good* **by** (*auto simp add*: *asTerm-equal-iff-alpha*)
}
also have *asTerm* (*?qop* (*pickInp* (*?sbsI inp*)) (*pickBinp* (*?sbsB binp*))) =
?op (*?sbsI inp*) (*?sbsB binp*) **unfolding** *Op-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *psubstAbs-simp*[*simp*]:
assumes *good-X*: *good X* **and** *good-rho*: *goodEnv rho* **and**
x-fresh-rho: *freshEnv xs x rho*
shows ((*Abs xs x X*) $[*rho*]) = *Abs xs x* (*X* #[*rho*])
proof−
let *?qrho* = *pickE rho*
let *?sbs* = *psubst rho* **let** *?qsbs* = *qPsubst ?qrho*
let *?sbsA* = *psubstAbs rho* **let** *?qsbsA* = *qPsubstAbs ?qrho*
have *good-qrho*: *qGoodEnv ?qrho*
using *good-rho goodEnv-imp-qGoodEnv-pickE* **by** *auto*
have *good-pick-X*: *qGood* (*pick X*) **using** *good-X good-imp-qGood-pick* **by** *auto*
hence *good-qsbs-pick-X*: *qGood*(*?qsbs* (*pick X*))
using *good-qrho qPsubst-preserves-qGood* **by** *auto*
have *good-qAbs-pick-X*: *qGoodAbs* (*qAbs xs x* (*pick X*))
using *good-X good-imp-qGood-pick* **by** *auto*
hence *qGoodAbs* ((*pick o asAbs*) (*qAbs xs x* (*pick X*)))
using *goodAbs-imp-qGoodAbs-pick qGoodAbs-iff-goodAbs-asAbs* **by** *fastforce*
hence *good*: *qGoodAbs* (*?qsbsA* ((*pick o asAbs*) (*qAbs xs x* (*pick X*))))
using *good-qrho qPsubstAbs-preserves-qGoodAbs* **by** *auto*
have *x-fresh-qrho*: *qFreshEnv xs x ?qrho*
using *x-fresh-rho* **unfolding** *freshEnv-def2* **by** *auto*

have *?sbsA* (*Abs xs x X*) = *asAbs* (*?qsbsA* ((*pick o asAbs*) (*qAbs xs x* (*pick X*))))
unfolding *psubstAbs-def Abs-def* **by** *simp*
also
{have (*pick o asAbs*) (*qAbs xs x* (*pick X*)) $= *qAbs xs x* (*pick X*)
using *good-qAbs-pick-X pick-asAbs* **by** *fastforce*
hence *?qsbsA*((*pick o asAbs*) (*qAbs xs x* (*pick X*))) $= *?qsbsA*(*qAbs xs x* (*pick*
X))
using *good-qAbs-pick-X good-qrho qPsubstAbs-preserves-alphaAbs1* **by** *force*
moreover have *?qsbsA*(*qAbs xs x* (*pick X*)) $= *qAbs xs x* (*?qsbs* (*pick X*))

using *qFresh-qPsubst-commute-qAbs good-pick-X good-qrho x-fresh-qrho* **by** *auto*
moreover
{have *?qsbs* (*pick X*) #= *pick* (*?sbs X*)
using *good-rho good-X pick-psubst-qPsubst alpha-sym* **by** *fastforce*
hence *qAbs xs x* (*?qsbs* (*pick X*)) $= *qAbs xs x* (*pick* (*?sbs X*))
using *good-qsbs-pick-X qAbs-preserves-alpha* **by** *fastforce*
}
ultimately
have *?qsbsA*((*pick o asAbs*) (*qAbs xs x* (*pick X*))) $= *qAbs xs x* (*pick* (*?sbs X*))
using *good alphaAbs-trans* **by** *blast*
hence *asAbs* (*?qsbsA*((*pick o asAbs*) (*qAbs xs x* (*pick X*)))) =
asAbs (*qAbs xs x* (*pick* (*?sbs X*)))
using *good asAbs-equal-iff-alphaAbs* **by** *auto*
}
also have *asAbs* (*qAbs xs x* (*pick* (*?sbs X*))) = *Abs xs x* (*?sbs X*)
unfolding *Abs-def* **by** *simp*
finally show *?thesis* **.**
qed

lemmas *good-psubstAll-simps* =
psubst-Var-simp1 psubst-Var-simp2
psubst-Op-simp psubstAbs-simp

theorem *getEnv-idEnv*[*simp*]: *idEnv xs x* = *None*
unfolding *idEnv-def* **by** *simp*

lemma *getEnv-updEnv*[*simp*]:
(*rho* [*x* ← *X*]-*xs*) *ys y* = (*if ys* = *xs* ∧ *y* = *x then Some X else rho ys y*)
unfolding *updEnv-def* **by** *auto*

theorem *getEnv-updEnv1*:
ys ≠ *xs* ∨ *y* ≠ *x* ⟹ (*rho* [*x* ← *X*]-*xs*) *ys y* = *rho ys y*
by *auto*

theorem *getEnv-updEnv2*:
(*rho* [*x* ← *X*]-*xs*) *xs x* = *Some X*
by *auto*

lemma *subst-Var-simp1*[*simp*]:
assumes *good Y*
and *ys* ≠ *xs* ∨ *y* ≠ *x*
shows ((*Var xs x*) #[*Y* / *y*]-*ys*) = *Var xs x*
using *assms* **unfolding** *subst-def* **by** *auto*

lemma *subst-Var-simp2*[*simp*]:
assumes *good Y*
shows ((*Var xs x*) #[*Y* / *x*]-*xs*) = *Y*
using *assms* **unfolding** *subst-def* **by** *auto*

lemma *subst-Op-simp*[*simp*]:
assumes *good Y*
and *goodInp inp* **and** *goodBinp binp*
shows
((*Op delta inp binp*) #[*Y* / *y*]-*ys*) =
Op delta (*inp* %[*Y* / *y*]-*ys*) (*binp* %%[*Y* / *y*]-*ys*)
using *assms* **unfolding** *subst-def substInp-def substBinp-def* **by** *auto*

lemma *substAbs-simp*[*simp*]:
assumes *good*: *good Y* **and** *good-X*: *good X* **and**
x-dif-y: *xs* ≠ *ys* ∨ *x* ≠ *y* **and** *x-fresh*: *fresh xs x Y*
shows ((*Abs xs x X*) $[*Y* / *y*]-*ys*) = *Abs xs x* (*X* #[*Y* / *y*]-*ys*)
proof−
have *freshEnv xs x* (*idEnv* [*y* ← *Y*]-*ys*) **unfolding** *freshEnv-def liftAll-def*
using *x-dif-y x-fresh* **by** *auto*
thus *?thesis* **using** *assms* **unfolding** *subst-def substAbs-def* **by** *auto*
qed

lemmas *good-substAll-simps* =
subst-Var-simp1 subst-Var-simp2
subst-Op-simp substAbs-simp

theorem *vsubst-Var-simp*[*simp*]:
((*Var xs x*) #[*y1* // *y*]-*ys*) = *Var xs* (*x* @*xs*[*y1* / *y*]-*ys*)
unfolding *vsubst-def*
apply(*case-tac ys* = *xs* ∧ *y* = *x*) **by** *simp-all*

lemma *vsubst-Op-simp*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows
((*Op delta inp binp*) #[*y1* // *y*]-*ys*) =
Op delta (*inp* %[*y1* // *y*]-*ys*) (*binp* %%[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstInp-def vsubstBinp-def* **by** *auto*

lemma *vsubstAbs-simp*[*simp*]:
assumes *good X* **and**
xs ≠ *ys* ∨ *x* ∉ {*y*,*y1*}
shows ((*Abs xs x X*) $[*y1* // *y*]-*ys*) = *Abs xs x* (*X* #[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstAbs-def* **by** *auto*

lemmas *good-vsubstAll-simps* =
vsubst-Op-simp vsubstAbs-simp

lemmas *good-allOpers-simps* =
good-swapAll-simps
good-freshAll-simps
good-skelAll-simps
good-psubstAll-simps
good-substAll-simps

good-vsubstAll-simps

5.5.2 The ability to pick fresh variables

lemma *single-non-fresh-ordLess-var*:
good X $\Longrightarrow$ $|\{x.\ \neg$ *fresh xs x X*$\}|$ $<o$ $|$*UNIV* :: *'var set*$|$
unfolding *fresh-def*
by(*auto simp add*: *good-imp-qGood-pick single-non-qFresh-ordLess-var*)

lemma *single-non-freshAbs-ordLess-var*:
goodAbs A $\Longrightarrow$ $|\{x.\ \neg$ *freshAbs xs x A*$\}|$ $<o$ $|$*UNIV* :: *'var set*$|$
unfolding *freshAbs-def*
by(*auto simp add*: *goodAbs-imp-qGoodAbs-pick single-non-qFreshAbs-ordLess-var*)

lemma *obtain-fresh1*:
fixes *XS*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term set* **and**
Rho::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env set* **and** *rho*
assumes *Vvar*: $|V|$ $<o$ $|$*UNIV* :: *'var set*$|$ $\vee$ *finite V* **and** *XSvar*: $|XS|$ $<o$ $|$*UNIV* :: *'var set*$|$ $\vee$ *finite XS* **and**
good: $\forall$ *X* $\in$ *XS*. *good X* **and**
Rhovar: $|Rho|$ $<o$ $|$*UNIV* :: *'var set*$|$ $\vee$ *finite Rho* **and** *RhoGood*: $\forall$ *rho* $\in$ *Rho*. *goodEnv rho*
shows
$\exists$ *z*. *z* $\notin$ *V* $\wedge$
($\forall$ *X* $\in$ *XS*. *fresh xs z X*) $\wedge$
($\forall$ *rho* $\in$ *Rho*. *freshEnv xs z rho*)
proof−
let *?qXS* = *pick* ‘ *XS* **let** *?qRho* = *pickE* ‘ *Rho*
have $|$*?qXS*$|$ $\leq o$ $|XS|$ **using** *card-of-image* **by** *auto*
hence *1*: $|$*?qXS*$|$ $<o$ $|$*UNIV* :: *'var set*$|$ $\vee$ *finite ?qXS*
using *ordLeq-ordLess-trans card-of-ordLeq-finite XSvar* **by** *blast*
have $|$*?qRho*$|$ $\leq o$ $|Rho|$ **using** *card-of-image* **by** *auto*
hence *2*: $|$*?qRho*$|$ $<o$ $|$*UNIV* :: *'var set*$|$ $\vee$ *finite ?qRho*
using *ordLeq-ordLess-trans card-of-ordLeq-finite Rhovar* **by** *blast*
have *3*: $\forall$ *qX* $\in$ *?qXS*. *qGood qX* **using** *good good-imp-qGood-pick* **by** *auto*
have $\forall$ *qrho* $\in$ *?qRho*. *qGoodEnv qrho* **using** *RhoGood goodEnv-imp-qGoodEnv-pickE* **by** *auto*
then obtain *z* **where**
z $\notin$ *V* $\wedge$ ($\forall$ *qX* $\in$ *?qXS*. *qFresh xs z qX*) $\wedge$
($\forall$ *qrho* $\in$ *?qRho*. *qFreshEnv xs z qrho*)
using *Vvar 1 2 3 obtain-qFreshEnv*[*of V ?qXS ?qRho*] **by** *fastforce*
thus *?thesis* **unfolding** *fresh-def freshEnv-def2* **by** *auto*
qed

lemma *obtain-fresh*:
fixes *V*::*'var set* **and**
XS::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term set* **and**
AS::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs set* **and**
Rho::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env set*

```
assumes Vvar: |V| <o |UNIV :: 'var set| ∨ finite V and
        XSvar: |XS| <o |UNIV :: 'var set| ∨ finite XS and
        ASvar: |AS| <o |UNIV :: 'var set| ∨ finite AS and
        Rhovar: |Rho| <o |UNIV :: 'var set| ∨ finite Rho and
        good: ∀ X ∈ XS. good X and
        ASGood: ∀ A ∈ AS. goodAbs A and
        RhoGood: ∀ rho ∈ Rho. goodEnv rho
shows
∃ z. z ∉ V ∧
     (∀ X ∈ XS. fresh xs z X) ∧
     (∀ A ∈ AS. freshAbs xs z A) ∧
     (∀ rho ∈ Rho. freshEnv xs z rho)
proof−
  have XS: |XS| <o |UNIV :: 'var set| and AS: |AS| <o |UNIV :: 'var set|
  using XSvar ASvar finite-ordLess-var by auto
  let ?phi = % A Y. (good Y ∧ (EX ys y. A = Abs ys y Y))
  {fix A assume A ∈ AS
   hence goodAbs A using ASGood by simp
   hence EX Y. ?phi A Y using abs-nchotomy[of A] by auto
  }
  then obtain f where 1: ALL A : AS. ?phi A (f A)
  using bchoice[of AS ?phi] by auto
  let ?YS = f ` AS
  have 2: ALL Y : ?YS. good Y using 1 by simp
  have |?YS| <=o |AS| using card-of-image by auto
  hence |?YS| <o |UNIV :: 'var set|
  using AS ordLeq-ordLess-trans by blast
  hence |XS Un ?YS| <o |UNIV :: 'var set|
  using XS by (auto simp add: var-infinite-INNER card-of-Un-ordLess-infinite)
  then obtain z where z: z ∉ V
  and XSYS: ∀ X ∈ XS Un ?YS. fresh xs z X
  and Rho: ∀ rho ∈ Rho. freshEnv xs z rho
  using Vvar Rhovar good 2 RhoGood
        obtain-fresh1[of V XS Un ?YS Rho xs] by blast
  moreover
  {fix A
   obtain Y where Y-def: Y = f A by blast
   assume A : AS
   hence fresh xs z Y unfolding Y-def using XSYS by simp
   moreover obtain ys y where Y: good Y and A: A = Abs ys y Y
   unfolding Y-def using ‹A : AS› 1 by auto
   ultimately have freshAbs xs z A unfolding A using z by auto
  }
  ultimately show ?thesis by auto
qed
```

5.5.3 Compositionality

```
lemma swap-ident[simp]:
```

assumes *good X*
shows (*X* #[*x* ∧ *x*]-*xs*) = *X*
using *assms asTerm-pick qSwap-ident* **unfolding** *swap-def* **by** *auto*

lemma *swap-compose*:
assumes *good-X*: *good X*
shows ((*X* #[*x* ∧ *y*]-*zs*) #[*x'* ∧ *y'*]-*zs'*) =
    ((*X* #[*x'* ∧ *y'*]-*zs'*) #[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]-*zs*)
using *assms qSwap-compose*[*of* - - - - - - *pick X*] **by**(*auto simp add*: *double-swap-qSwap*)

lemma *swap-commute*:
⟦*good X*; *zs* ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {}⟧ ⟹
 ((*X* #[*x* ∧ *y*]-*zs*) #[*x'* ∧ *y'*]-*zs'*) = ((*X* #[*x'* ∧ *y'*]-*zs'*) #[*x* ∧ *y*]-*zs*)
using *swap-compose*[*of X zs' x' y' zs x y*] **by**(*auto simp add*: *sw-def*)

lemma *swap-involutive*[*simp*]:
assumes *good-X*: *good X*
shows ((*X* #[*x* ∧ *y*]-*zs*) #[*x* ∧ *y*]-*zs*) = *X*
using *assms asTerm-pick*[*of X*] **by** (*auto simp add*: *double-swap-qSwap*)

theorem *swap-sym*: (*X* #[*x* ∧ *y*]-*zs*) = (*X* #[*y* ∧ *x*]-*zs*)
unfolding *swap-def* **by**(*auto simp add*: *qSwap-sym*)

lemma *swap-involutive2*[*simp*]:
assumes *good X*
shows ((*X* #[*x* ∧ *y*]-*zs*) #[*y* ∧ *x*]-*zs*) = *X*
using *assms* **by**(*simp add*: *swap-sym*)

lemma *swap-preserves-fresh*[*simp*]:
assumes *good X*
shows *fresh xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*) (*X* #[*y1* ∧ *y2*]-*ys*) = *fresh xs x X*
unfolding *fresh-def*[*of* - - *X*] **using** *assms qSwap-preserves-qFresh*[*of* - - - - - *pick* *X*]
by(*auto simp add*: *fresh-swap-qFresh-qSwap*)

lemma *swap-preserves-fresh-distinct*:
assumes *good X* **and**
    xs ≠ *ys* ∨ *x* ∉ {*y1*,*y2*}
shows *fresh xs x* (*X* #[*y1* ∧ *y2*]-*ys*) = *fresh xs x X*
unfolding *fresh-def*[*of* - - *X*] **using** *assms*
by(*auto simp*: *fresh-swap-qFresh-qSwap qSwap-preserves-qFresh-distinct*)

lemma *fresh-swap-exchange1*:
assumes *good X*
shows *fresh xs x2* (*X* #[*x1* ∧ *x2*]-*xs*) = *fresh xs x1 X*
unfolding *fresh-def*[*of* - - *X*]
using *assms* **by**(*auto simp*: *fresh-swap-qFresh-qSwap qFresh-qSwap-exchange1*)

lemma *fresh-swap-exchange2*:

assumes *good X* **and** $\{x1,x2\} \subseteq$ *var xs*
shows *fresh xs x2* (*X* #[*x2* ∧ *x1*]-*xs*) = *fresh xs x1 X*
using *assms* **by**(*simp add*: *fresh-swap-exchange1 swap-sym*)

lemma *fresh-swap-id*[*simp*]:
assumes *good X* **and** *fresh xs x1 X fresh xs x2 X*
shows (*X* #[*x1* ∧ *x2*]-*xs*) = *X*
by (*metis* (*no-types*, *lifting*) *assms alpha-imp-asTerm-equal alpha-qFresh-qSwap-id asTerm-pick*
fresh-def good-imp-qGood-pick local.swap-def qSwap-preserves-qGood1)

lemma *freshAbs-swapAbs-id*[*simp*]:
assumes *goodAbs A freshAbs xs x1 A freshAbs xs x2 A*
shows (*A* $[*x1* ∧ *x2*]-*xs*) = *A*
using *assms*
by (*meson alphaAbs-qFreshAbs-qSwapAbs-id alphaAll-trans freshAbs-def goodAbs-imp-qGoodAbs-pick*
pick-alphaAbs-iff-equal pick-swapAbs-qSwapAbs swapAbs-preserves-good)

lemma *fresh-swap-compose*:
assumes *good X fresh xs y X fresh xs z X*
shows ((*X* #[*y* ∧ *x*]-*xs*) #[*z* ∧ *y*]-*xs*) = (*X* #[*z* ∧ *x*]-*xs*)
using *assms* **by** (*simp add*: *sw-def swap-compose*)

lemma *skel-swap*:
assumes *good X*
shows *skel* (*X* #[*x1* ∧ *x2*]-*xs*) = *skel X*
using *assms* **by** (*metis alpha-qSkel pick-swap-qSwap qSkel-qSwap skel-def*)

5.5.4 Compositionality for environments

lemma *swapEnv-ident*[*simp*]:
assumes *goodEnv rho*
shows (*rho* &[*x* ∧ *x*]-*xs*) = *rho*
using *assms* **unfolding** *swapEnv-defs lift-def*
by (*intro ext*) (*auto simp*: *option.case-eq-if*)

lemma *swapEnv-compose*:
assumes *good*: *goodEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*x'* ∧ *y'*]-*zs'*) =
((*rho* &[*x'* ∧ *y'*]-*zs'*) &[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]-*zs*)
proof(*rule ext*)+
let *?xsw* = *x* @*zs*[*x'* ∧ *y'*]-*zs'* **let** *?ysw* = *y* @*zs*[*x'* ∧ *y'*]-*zs'*
let *?xswsw* = *?xsw* @*zs*[*x'* ∧ *y'*]-*zs'* **let** *?yswsw* = *?ysw* @*zs*[*x'* ∧ *y'*]-*zs'*
let *?rhosw1* = *rho* &[*x* ∧ *y*]-*zs* **let** *?rhosw11* = *?rhosw1* &[*x'* ∧ *y'*]-*zs'*
let *?rhosw2* = *rho* &[*x'* ∧ *y'*]-*zs'* **let** *?rhosw22* = *?rhosw2* &[*?xsw* ∧ *?ysw*]-*zs*
let *?Sw1* = λ*X*. (*X* #[*x* ∧ *y*]-*zs*) **let** *?Sw11* = λ*X*. ((*?Sw1 X*) #[*x'* ∧ *y'*]-*zs'*)

let *?Sw2* = λ*X*. (*X* #[*x'* ∧ *y'*]-*zs'*) **let** *?Sw22* = λ*X*. ((*?Sw2 X*) #[*?xsw* ∧ *?ysw*]-*zs*)
fix *us u*
let *?usw1* = *u* @*us* [*x'* ∧ *y'*]-*zs'* **let** *?usw11* = *?usw1* @*us* [*x* ∧ *y*]-*zs*
let *?usw2* = *u* @*us* [*?xsw* ∧ *?ysw*]-*zs* **let** *?usw22* = *?usw2* @*us* [*x'* ∧ *y'*]-*zs'*
have (*?xsw* @*zs*[*x'* ∧ *y'*]-*zs'*) = *x* **and** (*?ysw* @*zs*[*x'* ∧ *y'*]-*zs'*) = *y* **by** *auto*
have *?usw22* = (*?usw1* @*us*[*?xswsw* ∧ *?yswsw*]-*zs*) **using** *sw-compose* **.**
hence ∗: *?usw22* = *?usw11* **by** *simp*
show *?rhosw11 us u* = *?rhosw22 us u*
proof(*cases rho us ?usw11*)
case *None*
hence *?rhosw11 us u* = *None* **unfolding** *swapEnv-defs lift-def* **by** *simp*
also have ... = *?rhosw22 us u*
using *None* **unfolding** ∗ *swapEnv-defs lift-def* **by** *simp*
finally show *?thesis* **.**
next
case (*Some X*)
hence *good X* **using** *good* **unfolding** *goodEnv-def liftAll-def* **by** *simp*
have *?rhosw11 us u* = *Some*(*?Sw11 X*) **using** *Some* **unfolding** *swapEnv-defs lift-def* **by** *simp*
also have *?Sw11 X* = *?Sw22 X*
using ‹*good X*› **by**(*rule swap-compose*)
also have *Some*(*?Sw22 X*) = *?rhosw22 us u*
using *Some* **unfolding** ∗ *swapEnv-defs lift-def* **by** *simp*
finally show *?thesis* **.**
qed
qed

lemma *swapEnv-commute*:
⟦*goodEnv rho*; {*x*,*y*} ⊆ *var zs*; *zs* ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {}⟧ ⟹
((*rho* &[*x* ∧ *y*]-*zs*) &[*x'* ∧ *y'*]-*zs'*) = ((*rho* &[*x'* ∧ *y'*]-*zs'*) &[*x* ∧ *y*]-*zs*)
using *swapEnv-compose*[*of rho zs' x' y' zs x y*] **by**(*auto simp add*: *sw-def*)

lemma *swapEnv-involutive*[*simp*]:
assumes *goodEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*x* ∧ *y*]-*zs*) = *rho*
using *assms* **unfolding** *swapEnv-defs lift-def*
by (*fastforce simp*: *option.case-eq-if*)

theorem *swapEnv-sym*: (*rho* &[*x* ∧ *y*]-*zs*) = (*rho* &[*y* ∧ *x*]-*zs*)
proof(*intro ext*)
fix *us u*
have ∗: (*u* @*us*[*x* ∧ *y*]-*zs*) = (*u* @*us*[*y* ∧ *x*]-*zs*) **using** *sw-sym* **by** *fastforce*
show (*rho* &[*x* ∧ *y*]-*zs*) *us u* = (*rho* &[*y* ∧ *x*]-*zs*) *us u*
unfolding *swapEnv-defs lift-def* ∗
by(*cases rho us* (*u* @*us*[*y* ∧ *x*]-*zs*)) (*auto simp*: *swap-sym*)
qed

lemma *swapEnv-involutive2*[*simp*]:

assumes *good*: *goodEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*y* ∧ *x*]-*zs*) = *rho*
using *assms* **by**(*simp add*: *swapEnv-sym*)

lemma *swapEnv-preserves-freshEnv*[*simp*]:
assumes *good*: *goodEnv rho*
shows *freshEnv xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*) (*rho* &[*y1* ∧ *y2*]-*ys*) = *freshEnv xs x rho*
proof−
let *?xsw* = *x* @*xs*[*y1* ∧ *y2*]-*ys* **let** *?xswsw* = *?xsw* @*xs*[*y1* ∧ *y2*]-*ys*
let *?rhosw* = *rho* &[*y1* ∧ *y2*]-*ys*
let *?Left* = *freshEnv xs ?xsw ?rhosw*
let *?Right* = *freshEnv xs x rho*
have (*?rhosw xs ?xsw* = *None*) = (*rho xs x* = *None*)
unfolding *freshEnv-def swapEnv-defs*
by(*simp add*: *lift-None sw-involutive*)
moreover
have (∀ *zs z′ Z′*. *?rhosw zs z′* = *Some Z′* ⟶ *fresh xs ?xsw Z′*) =
(∀ *zs z Z*. *rho zs z* = *Some Z* ⟶ *fresh xs x Z*)
proof(*rule iff-allI*, *auto*)
fix *zs z Z* **assume** ∗: ∀ *z′ Z′*. *?rhosw zs z′* = *Some Z′* ⟶ *fresh xs ?xsw Z′*
and ∗∗: *rho zs z* = *Some Z* **let** *?z′* = *z* @*zs*[*y1* ∧ *y2*]-*ys* **let** *?Z′* = *Z* #[*y1* ∧ *y2*]-*ys*
have *?rhosw zs ?z′* = *Some ?Z′* **using** ∗∗ **unfolding** *swapEnv-defs lift-def*
by(*simp add*: *sw-involutive*)
hence *fresh xs ?xsw ?Z′* **using** ∗ **by** *simp*
moreover have *good Z* **using** ∗∗ *good* **unfolding** *goodEnv-def liftAll-def* **by** *simp*
ultimately show *fresh xs x Z* **using** *swap-preserves-fresh* **by** *auto*
next
fix *zs z′ Z′*
assume ∗: ∀ *z Z*. *rho zs z* = *Some Z* ⟶ *fresh xs x Z* **and** ∗∗: *?rhosw zs z′* = *Some Z′*
let *?z* = *z′* @*zs*[*y1* ∧ *y2*]-*ys*
obtain *Z* **where** *rho*: *rho zs ?z* = *Some Z* **and** *Z′*: *Z′* = *Z* #[*y1* ∧ *y2*]-*ys*
using ∗∗ **unfolding** *swapEnv-defs lift-def* **by**(*cases rho zs ?z*, *auto*)
hence *fresh xs x Z* **using** ∗ **by** *simp*
moreover have *good Z* **using** *rho good* **unfolding** *goodEnv-def liftAll-def* **by** *simp*
ultimately show *fresh xs ?xsw Z′* **unfolding** *Z′* **using** *swap-preserves-fresh* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *freshEnv-def swapEnv-defs*
unfolding *liftAll-def* **by** *simp*
qed

lemma *swapEnv-preserves-freshEnv-distinct*:
assumes *goodEnv rho* **and**
xs ≠ *ys* ∨ *x* ∉ {*y1*,*y2*}
shows *freshEnv xs x* (*rho* &[*y1* ∧ *y2*]-*ys*) = *freshEnv xs x rho*

by (*metis assms sw-simps3 swapEnv-preserves-freshEnv*)

lemma *freshEnv-swapEnv-exchange1*:
assumes *goodEnv rho*
shows *freshEnv xs x2* (*rho* &[*x1* ∧ *x2*]-*xs*) = *freshEnv xs x1 rho*
by (*metis assms sw-simps1 swapEnv-preserves-freshEnv*)

lemma *freshEnv-swapEnv-exchange2*:
assumes *goodEnv rho*
shows *freshEnv xs x2* (*rho* &[*x2* ∧ *x1*]-*xs*) = *freshEnv xs x1 rho*
using *assms* **by**(*simp add*: *freshEnv-swapEnv-exchange1 swapEnv-sym*)

lemma *freshEnv-swapEnv-id*[*simp*]:
assumes *good*: *goodEnv rho* **and**
fresh: *freshEnv xs x1 rho* *freshEnv xs x2 rho*
shows (*rho* &[*x1* ∧ *x2*]-*xs*) = *rho*
proof(*intro ext*)
fix *us u*
let *?usw* = *u* @*us*[*x1* ∧ *x2*]-*xs* **let** *?rhosw* = *rho* &[*x1* ∧ *x2*]-*xs*
let *?Sw* = λ *X*. (*X* #[*x1* ∧ *x2*]-*xs*)
show *?rhosw us u* = *rho us u*
proof(*cases rho us u*)
case *None*
hence *rho us ?usw* = *None* **using** *fresh* **unfolding** *freshEnv-def sw-def* **by** *auto*
hence *?rhosw us u* = *None* **unfolding** *swapEnv-defs lift-def* **by** *auto*
with *None* **show** *?thesis* **by** *simp*
next
case (*Some X*)
moreover have *?usw* = *u* **using** *fresh Some* **unfolding** *freshEnv-def sw-def* **by** *auto*
ultimately have *?rhosw us u* = *Some* (*?Sw X*) **unfolding** *swapEnv-defs lift-def* **by** *auto*
moreover
{**have** *good X* **using** *Some good* **unfolding** *goodEnv-def liftAll-def* **by** *auto*
moreover have *fresh xs x1 X* **and** *fresh xs x2 X*
using *Some fresh* **unfolding** *freshEnv-def liftAll-def* **by** *auto*
ultimately have *?Sw X* = *X* **by** *simp*
}
ultimately show *?thesis* **using** *Some* **by** *simp*
qed
qed

lemma *freshEnv-swapEnv-compose*:
assumes *good*: *goodEnv rho* **and**
fresh: *freshEnv xs y rho* *freshEnv xs z rho*
shows ((*rho* &[*y* ∧ *x*]-*xs*) &[*z* ∧ *y*]-*xs*) = (*rho* &[*z* ∧ *x*]-*xs*)
by (*simp add*: *fresh good sw-def swapEnv-compose*)

lemmas *good-swapAll-freshAll-otherSimps* =
swap-ident swap-involutive swap-involutive2 swap-preserves-fresh fresh-swap-id freshAbs-swapAbs-id
swapEnv-ident swapEnv-involutive swapEnv-involutive2 swapEnv-preserves-freshEnv freshEnv-swapEnv-id

5.5.5 Properties of the relation of being swapped

theorem *swap-swapped*: $(X,\ X\ \#[x \wedge y]\text{-}zs) \in swapped$
by(*auto simp add*: *swapped.Refl swapped.Swap*)

lemma *swapped-preserves-good*:
assumes *good X* **and** $(X,Y) \in swapped$
shows *good Y*
using *assms*(*2*,*1*) **by** (*induct rule*: *swapped.induct*) *auto*

lemma *swapped-skel*:
assumes *good X* **and** $(X,Y) \in swapped$
shows *skel Y* = *skel X*
using *assms*(*2*,*1*)
by (*induct rule*: *swapped.induct*) (*auto simp*: *swapped-preserves-good skel-swap*)

lemma *obtain-rep*:
assumes *GOOD*: *good X* **and** *FRESH*: *fresh xs x′ X*
shows $\exists\ X'.\ (X,X') \in swapped \wedge good\ X' \wedge Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$
using *Abs-swap-fresh FRESH GOOD swap-preserves-good swap-swapped* **by** *blast*

5.6 Induction

5.6.1 Induction lifted from quasi-terms

lemma *term-templateInduct*[*case-names rel Var Op Abs*]:
fixes *X*::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*term* **and**
A::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*abs* **and** *phi phiAbs rel*
assumes
rel: $\bigwedge X\ Y.\ [\![good\ X;\ (X,Y) \in rel]\!] \Longrightarrow good\ Y \wedge skel\ Y = skel\ X$ **and**
var: $\bigwedge xs\ x.\ phi\ (Var\ xs\ x)$ **and**
op: $\bigwedge delta\ inp\ binp.\ [\![goodInp\ inp;\ goodBinp\ binp;\ liftAll\ phi\ inp;\ liftAll\ phiAbs\ binp]\!]$
$\Longrightarrow phi\ (Op\ delta\ inp\ binp)$ **and**
abs: $\bigwedge xs\ x\ X.\ [\![good\ X;\ \bigwedge\ Y.\ (X,Y) \in rel \Longrightarrow phi\ Y]\!]$
$\Longrightarrow phiAbs\ (Abs\ xs\ x\ X)$
shows $(good\ X \longrightarrow phi\ X) \wedge (goodAbs\ A \longrightarrow phiAbs\ A)$
proof−
let *?qX* = *pick X* **let** *?qA* = *pick A*
let *?qphi* = *phi o asTerm* **let** *?qphiAbs* = *phiAbs o asAbs*
let *?qrel* = $\{(qY,\ qY')|\ qY\ qY'.\ (asTerm\ qY,\ asTerm\ qY') \in rel\}$

have $(good\ X \longrightarrow qGood\ ?qX) \wedge (goodAbs\ A \longrightarrow qGoodAbs\ ?qA)$
using *good-imp-qGood-pick goodAbs-imp-qGoodAbs-pick* **by** *auto*

moreover
have (*good X* ⟶ (*?qphi ?qX* = *phi X*)) ∧ (*goodAbs A* ⟶ (*?qphiAbs ?qA* = *phiAbs A*))
using *asTerm-pick asAbs-pick* **by** *fastforce*
moreover
have (*qGood ?qX* ⟶ *?qphi ?qX*) ∧ (*qGoodAbs ?qA* ⟶ *?qphiAbs ?qA*)
proof(*induction rule*: *qGood-qTerm-templateInduct*[*of ?qrel*])
case (*Rel qX qY*)
thus *?case* **using** *qGood-iff-good-asTerm pick-asTerm* **unfolding** *skel-def*
using *rel skel-asTerm-qSkel*
by *simp* (*smt* (*verit*) *qGood-iff-good-asTerm skel-asTerm-qSkel*)
next
case (*Var xs x*)
then show *?case* **using** *var* **unfolding** *Var-def* **by** *simp*
next
case (*Op delta qinp qbinp*)
hence *good-qinp*: *qGoodInp qinp* ∧ *qGoodBinp qbinp*
unfolding *qGoodInp-def qGoodBinp-def liftAll-def* **by** *simp*
let *?inp* = *asInp qinp* **let** *?binp* = *asBinp qbinp*
have *good-inp*: *goodInp ?inp* ∧ *goodBinp ?binp*
using *good-qinp qGoodInp-iff-goodInp-asInp qGoodBinp-iff-goodBinp-asBinp* **by** *auto*
have *1*: *Op delta ?inp ?binp* = *asTerm* (*qOp delta qinp qbinp*)
using *good-qinp Op-asInp-asTerm-qOp* **by** *fastforce*
{**fix** *i X*
assume *inp*: *?inp i* = *Some X*
then obtain *qX* **where** *qinp*: *qinp i* = *Some qX* **and** *X*: *X* = *asTerm qX*
unfolding *asInp-def lift-def* **by**(*cases qinp i*, *auto*)
have *qGood qX* ∧ *phi* (*asTerm qX*) **using** *qinp Op.IH* **by** (*simp add*: *liftAll-def*)
hence *good X* ∧ *phi X* **unfolding** *X* **using** *qGood-iff-good-asTerm* **by** *auto*
}
moreover
{**fix** *i A*
assume *binp*: *?binp i* = *Some A*
then obtain *qA* **where** *qbinp*: *qbinp i* = *Some qA* **and** *A*: *A* = *asAbs qA*
unfolding *asBinp-def lift-def* **by**(*cases qbinp i*, *auto*)
have *qGoodAbs qA* ∧ *phiAbs* (*asAbs qA*) **using** *qbinp Op.IH* **by** (*simp add*: *liftAll-def*)
hence *goodAbs A* ∧ *phiAbs A* **unfolding** *A* **using** *qGoodAbs-iff-goodAbs-asAbs* **by** *auto*
}
ultimately show *?case*
using *op*[*of ?inp ?binp delta*] *good-inp* **unfolding** *1 liftAll-def* **by** *simp*
next
case (*Abs xs x qX*)
have *good* (*asTerm qX*) **using** ‹*qGood qX*› *qGood-iff-good-asTerm* **by** *auto*
moreover
{**fix** *Y* **assume** ∗: (*asTerm qX*, *Y*) ∈ *rel*
obtain *qY* **where** *qY*: *qY* = *pick Y* **by** *blast*

have *good* (*asTerm qX*) **using** ‹*qGood qX*› *qGood-iff-good-asTerm* **by** *auto*
hence *good Y* **using** ∗ *rel* **by** *auto*
hence *Y*: *Y* = *asTerm qY* **unfolding** *qY* **using** *asTerm-pick* **by** *auto*
have *phi Y* **using** ∗ *Abs.IH* **unfolding** *Y* **by** *simp*
}
ultimately have *phiAbs* (*Abs xs x* (*asTerm qX*)) **using** *abs* **by** *simp*
thus *?case* **using** ‹*qGood qX*› *Abs-asTerm-asAbs-qAbs* **by** *fastforce*
qed

ultimately show *?thesis* **by** *blast*
qed

lemma *term-rawInduct*[*case-names Var Op Abs*]:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and** *phi phiAbs*
assumes
Var: $\bigwedge$ *xs x*. *phi* (*Var xs x*) **and**
Op: $\bigwedge$ *delta inp binp*. ⟦*goodInp inp*; *goodBinp binp*; *liftAll phi inp*; *liftAll phiAbs binp*⟧
⟹ *phi* (*Op delta inp binp*) **and**
Abs: $\bigwedge$ *xs x X*. ⟦*good X*; *phi X*⟧ ⟹ *phiAbs* (*Abs xs x X*)
shows (*good X* ⟶ *phi X*) ∧ (*goodAbs A* ⟶ *phiAbs A*)
by(*rule term-templateInduct*[*of Id*], *auto simp add*: *assms*)

lemma *term-induct*[*case-names Var Op Abs*]:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and** *phi phiAbs*
assumes
Var: $\bigwedge$ *xs x*. *phi* (*Var xs x*) **and**
Op: $\bigwedge$ *delta inp binp*. ⟦*goodInp inp*; *goodBinp binp*; *liftAll phi inp*; *liftAll phiAbs binp*⟧
⟹ *phi* (*Op delta inp binp*) **and**
Abs: $\bigwedge$ *xs x X*. ⟦*good X*;
$\bigwedge$ *Y*. (*X*,*Y*) ∈ *swapped* ⟹ *phi Y*;
$\bigwedge$ *Y*. ⟦*good Y*; *skel Y* = *skel X*⟧ ⟹ *phi Y*⟧
⟹ *phiAbs* (*Abs xs x X*)
shows (*good X* ⟶ *phi X*) ∧ (*goodAbs A* ⟶ *phiAbs A*)
apply(*induct rule*: *term-templateInduct*[*of swapped* ∪ {(*X*,*Y*). *good Y* ∧ *skel Y* = *skel X*}])
by(*auto simp*: *assms swapped-skel swapped-preserves-good*)

5.6.2 Fresh induction

First a general situation, where parameters are of an unspecified type (that should be given by the user):

lemma *term-fresh-forall-induct*[*case-names PAR Var Op Abs*]:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and** *A*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs*

and *phi* **and** *phiAbs* **and** *varsOf* :: *'param* ⇒ *'varSort* ⇒ *'var set*

assumes
PAR: ⋀ *p xs*. (|*varsOf xs p*| <*o* |*UNIV*::′*var set*|) **and**
var: ⋀ *xs x p*. *phi* (*Var xs x*) *p* **and**
op: ⋀ *delta inp binp p*.
⟦|{*i*. *inp i* ≠ *None*}| <*o* |*UNIV*::′*var set*|; |{*i*. *binp i* ≠ *None*}| <*o* |*UNIV*::′*var set*|;
liftAll (λ *X*. *good X* ∧ (∀ *q*. *phi X p*)) *inp*; *liftAll* (λ *A*. *goodAbs A* ∧ (∀ *q*. *phiAbs A p*)) *binp*⟧
⟹ *phi* (*Op delta inp binp*) *p* **and**
abs: ⋀ *xs x X p*. ⟦*good X*; *x* ∉ *varsOf p xs*; *phi X p*⟧ ⟹ *phiAbs* (*Abs xs x X*) *p*
shows (*good X* ⟶ (∀ *p*. *phi X p*)) ∧ (*goodAbs A* ⟶ (∀ *p*. *phiAbs A p*))
proof(*induction rule*: *term-templateInduct*[*of swapped*])
case (*Abs xs x X*)
show *?case* **proof** *safe*
fix *p*
obtain *x′* **where** *x′-freshP*: *x′* ∉ *varsOf p xs* **and** *x′-fresh-X*: *fresh xs x′ X*
using ‹*good X*› *PAR obtain-fresh*[*of varsOf p xs* {*X*} {} {} *xs*] **by** *auto*
then obtain *X′* **where** *XX′*: (*X*, *X′*) ∈ *swapped* **and** *good-X′*: *good X′* **and**
Abs-eq: *Abs xs x X* = *Abs xs x′ X′*
using ‹*good X*› *x′-freshP x′-fresh-X* **using** *obtain-rep*[*of X xs x′ x*] **by** *auto*
thus *phiAbs* (*Abs xs x X*) *p*
unfolding *Abs-eq* **using** *x′-freshP good-X′ abs Abs.IH* **by** *simp*
qed
qed(*insert assms swapped-preserves-good swapped-skel*,
unfold liftAll-def goodInp-def goodBinp-def, *auto*)

lemma *term-templateInduct-fresh*[*case-names PAR Var Op Abs*]:
fixes *X*::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term* **and**
A::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*abs* **and**
rel **and** *phi* **and** *phiAbs* **and**
vars :: ′*varSort* ⇒ ′*var set* **and**
terms :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term set* **and**
abs :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*abs set* **and**
envs :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env set*
assumes
PAR:
⋀ *xs*.
(|*vars xs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*vars xs*)) ∧
(|*terms*| <*o* |*UNIV* :: ′*var set*| ∨ *finite terms*) ∧ (∀ *X* ∈ *terms*. *good X*) ∧
(|*abs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite abs*) ∧ (∀ *A* ∈ *abs*. *goodAbs A*) ∧
(|*envs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite envs*) ∧ (∀ *rho* ∈ *envs*. *goodEnv rho*) **and**
rel: ⋀ *X Y*. ⟦*good X*; (*X*,*Y*) ∈ *rel*⟧ ⟹ *good Y* ∧ *skel Y* = *skel X* **and**
Var: ⋀ *xs x*. *phi* (*Var xs x*) **and**
Op:
⋀ *delta inp binp*.
⟦*goodInp inp*; *goodBinp binp*;
liftAll phi inp; *liftAll phiAbs binp*⟧
⟹ *phi* (*Op delta inp binp*) **and**

abs:
⋀ *xs x X*.
⟦*good X*;
x ∉ *vars xs*;
⋀ *Y*. *Y* ∈ *terms* ⟹ *fresh xs x Y*;
⋀ *A*. *A* ∈ *abs* ⟹ *freshAbs xs x A*;
⋀ *rho*. *rho* ∈ *envs* ⟹ *freshEnv xs x rho*;
⋀ *Y*. (*X*,*Y*) ∈ *rel* ⟹ *phi Y*⟧
⟹ *phiAbs* (*Abs xs x X*)
shows
(*good X* ⟶ *phi X*) ∧
(*goodAbs A* ⟶ *phiAbs A*)
proof(*induction rule*: *term-templateInduct*[*of swapped O rel*])
case (*Abs xs x X*) **note** *good-X* = ‹*good X*›
have |{*X*} ∪ *terms*| <*o* |*UNIV* :: *'var set*| ∨ *finite* ({*X*} ∪ *terms*)
apply(*cases finite terms*, *auto simp add*: *PAR*)
using *PAR var-infinite-INNER card-of-Un-singl-ordLess-infinite* **by** *force*
then obtain *x'* **where** *x'-not*: *x'* ∉ *vars xs* **and**
x'-fresh-X: *fresh xs x' X* **and**
x'-freshP: (∀ *Y* ∈ *terms*. *fresh xs x' Y*) ∧
(∀ *A* ∈ *abs*. *freshAbs xs x' A*) ∧
(∀ *rho* ∈ *envs*. *freshEnv xs x' rho*)
using *good-X PAR*
using *obtain-fresh*[*of vars xs* {*X*} ∪ *terms abs envs xs*] **by** *auto*
then obtain *X'* **where** *XX'*: (*X*, *X'*) ∈ *swapped* **and** *good-X'*: *good X'* **and**
Abs-eq: *Abs xs x X* = *Abs xs x' X'*
using *good-X x'-not x'-fresh-X* **using** *obtain-rep*[*of X xs x' x*] **by** *auto*
have ⋀*Y*. (*X'*, *Y*) ∈ *rel* ⟹ *phi Y* **using** *XX' Abs.IH* **by** *auto*
thus *?case*
unfolding *Abs-eq* **using** *x'-not x'-freshP good-X' abs* **by** *auto*
qed(*insert Op rel*, *unfold relcomp-unfold liftAll-def*, *simp-all add*: *Var*,
metis rel swapped-preserves-good swapped-skel)

A version of the above not employing any relation for the bound-argument case:

lemma *term-rawInduct-fresh*[*case-names Par Var Op Obs*]:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
vars :: *'varSort* ⇒ *'var set* **and**
terms :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term set* **and**
abs :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs set* **and**
envs :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env set*
assumes
PAR:
⋀ *xs*.
(|*vars xs*| <*o* |*UNIV* :: *'var set*| ∨ *finite* (*vars xs*)) ∧
(|*terms*| <*o* |*UNIV* :: *'var set*| ∨ *finite terms*) ∧ (∀ *X* ∈ *terms*. *good X*) ∧
(|*abs*| <*o* |*UNIV* :: *'var set*| ∨ *finite abs*) ∧ (∀ *A* ∈ *abs*. *goodAbs A*) ∧
(|*envs*| <*o* |*UNIV* :: *'var set*| ∨ *finite envs*) ∧ (∀ *rho* ∈ *envs*. *goodEnv rho*) **and**

Var: $\bigwedge$ *xs x*. *phi* (*Var xs x*) **and**
Op:
$\bigwedge$ *delta inp binp*.
 ⟦*goodInp inp*; *goodBinp binp*;
  liftAll phi inp; *liftAll phiAbs binp*⟧
 $\Longrightarrow$ *phi* (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *xs x X*.
 ⟦*good X*;
  $x \notin$ *vars xs*;
  $\bigwedge$ *Y*. $Y \in$ *terms* $\Longrightarrow$ *fresh xs x Y*;
  $\bigwedge$ *A*. $A \in$ *abs* $\Longrightarrow$ *freshAbs xs x A*;
  $\bigwedge$ *rho*. *rho* $\in$ *envs* $\Longrightarrow$ *freshEnv xs x rho*;
  phi X⟧
 $\Longrightarrow$ *phiAbs* (*Abs xs x X*)
shows
(*good X* $\longrightarrow$ *phi X*) $\wedge$
 (*goodAbs A* $\longrightarrow$ *phiAbs A*)
apply(*induct rule*: *term-templateInduct-fresh*[*of vars terms abs envs Id*])
using *assms* **by** *auto*

The typical raw induction with freshness is one dealing with finitely many variables, terms, abstractions and environments as parameters – we have all these condensed in the notion of a parameter (type constructor "param"):

lemma *term-induct-fresh*[*case-names Par Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
  A :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
  P :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param*
assumes
goodP: *goodPar P* **and**
Var: $\bigwedge$ *xs x*. *phi* (*Var xs x*) **and**
Op:
$\bigwedge$ *delta inp binp*.
 ⟦*goodInp inp*; *goodBinp binp*;
  liftAll phi inp; *liftAll phiAbs binp*⟧
 $\Longrightarrow$ *phi* (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *xs x X*.
 ⟦*good X*;
  $x \notin$ *varsOf P*;
  $\bigwedge$ *Y*. $Y \in$ *termsOf P* $\Longrightarrow$ *fresh xs x Y*;
  $\bigwedge$ *A*. $A \in$ *absOf P* $\Longrightarrow$ *freshAbs xs x A*;
  $\bigwedge$ *rho*. *rho* $\in$ *envsOf P* $\Longrightarrow$ *freshEnv xs x rho*;
  phi X⟧
 $\Longrightarrow$ *phiAbs* (*Abs xs x X*)
shows
(*good X* $\longrightarrow$ *phi X*) $\wedge$
 (*goodAbs A* $\longrightarrow$ *phiAbs A*)
proof(*induct rule*: *term-rawInduct-fresh*

[*of* λ *xs. varsOf P termsOf P absOf P envsOf P*])
case (*Par xs*)
then show *?case* **unfolding** *goodPar-def*
using *goodP* **by**(*cases P*) *simp*
qed(*insert assms*, *auto*)

end

end

6 More on Terms

theory *Terms* **imports** *Transition-QuasiTerms-Terms*
begin

In this section, we continue the study of terms, with stating and proving properties specific to terms (while in the previous section we dealt with lifting properties from quasi-terms). Consequently, in this theory, not only the theorems, but neither the proofs should mention quasi-items at all. Among the properties specific to terms will be the compositionality properties of substitution (while, by contrast, similar properties of swapping also held for quasi-tems).

context *FixVars*
begin

declare *qItem-simps*[*simp del*]
declare *qItem-versus-item-simps*[*simp del*]

6.1 Identity environment versus other operators

theorem *getEnv-updEnv-idEnv*[*simp*]:
(*idEnv* [$x \leftarrow X$]-*xs*) *ys y* = (*if* (*ys* = *xs* $\wedge$ *y* = *x*) *then Some X else None*)
unfolding *idEnv-def updEnv-def* **by** *simp*

theorem *subst-psubst-idEnv*:
(*X* #[*Y* / *y*]-*ys*) = (*X* #[*idEnv* [$y \leftarrow Y$]-*ys*])
unfolding *subst-def idEnv-def updEnv-def psubst-def* **by** *simp*

theorem *vsubst-psubst-idEnv*:
(*X* #[*z* // *y*]-*ys*) = (*X* #[*idEnv* [$y \leftarrow$ *Var ys z*]-*ys*])
unfolding *vsubst-def* **by**(*simp add*: *subst-psubst-idEnv*)

theorem *substEnv-psubstEnv-idEnv*:
(*rho* &[*Y* / *y*]-*ys*) = (*rho* &[*idEnv* [$y \leftarrow Y$]-*ys*])
unfolding *substEnv-def idEnv-def updEnv-def psubstEnv-def* **by** *simp*

theorem *vsubstEnv-psubstEnv-idEnv*:
(*rho* &[*z* // *y*]-*ys*) = (*rho* &[*idEnv* [$y \leftarrow$ *Var ys z*]-*ys*])

unfolding *vsubstEnv-def* **by** (*simp add*: *substEnv-psubstEnv-idEnv*)

theorem *freshEnv-idEnv*: *freshEnv xs x idEnv*
unfolding *idEnv-def freshEnv-def liftAll-def* **by** *simp*

theorem *swapEnv-idEnv*[*simp*]: (*idEnv* &[$x \wedge y$]-*xs*) = *idEnv*
unfolding *idEnv-def swapEnv-def comp-def swapEnvDom-def swapEnvIm-def lift-def*
by *simp*

theorem *psubstEnv-idEnv*[*simp*]: (*idEnv* &[*rho*]) = *rho*
unfolding *idEnv-def psubstEnv-def lift-def* **by** *simp*

theorem *substEnv-idEnv*: (*idEnv* &[*X* / *x*]-*xs*) = (*idEnv* [$x \leftarrow X$]-*xs*)
unfolding *substEnv-def* **using** *psubstEnv-idEnv* **by** *auto*

theorem *vsubstEnv-idEnv*: (*idEnv* &[*y* // *x*]-*xs*) = (*idEnv* [$x \leftarrow$ (*Var xs y*)]-*xs*)
unfolding *vsubstEnv-def* **using** *substEnv-idEnv* **.**

lemma *psubstAll-idEnv*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs*
shows
(*good X* $\longrightarrow$ (*X* #[*idEnv*]) = *X*) $\wedge$
(*goodAbs A* $\longrightarrow$ (*A* $[*idEnv*]) = *A*)
apply(*induct rule*: *term-rawInduct*)
unfolding *psubstInp-def psubstBinp-def*
using *idEnv-preserves-good psubst-Var-simp1*
by (*simp-all del*: *getEnv-idEnv add*:
liftAll-lift-ext lift-ident freshEnv-idEnv psubstBinp-def psubstInp-def)
fastforce+

lemma *psubst-idEnv*[*simp*]:
good X $\Longrightarrow$ (*X* #[*idEnv*]) = *X*
by(*simp add*: *psubstAll-idEnv*)

lemma *psubstEnv-idEnv-id*[*simp*]:
assumes *goodEnv rho*
shows (*rho* &[*idEnv*]) = *rho*
using *assms* **unfolding** *psubstEnv-def lift-def goodEnv-def liftAll-def*
apply(*intro ext*) **subgoal for** *xs x* **by**(*cases rho xs x*) *auto* **.**

6.2 Environment update versus other operators

theorem *updEnv-overwrite*[*simp*]: ((*rho* [$x \leftarrow X$]-*xs*) [$x \leftarrow X'$]-*xs*) = (*rho* [$x \leftarrow X'$]-*xs*)
unfolding *updEnv-def* **by** *fastforce*

theorem *updEnv-commute*:
assumes $xs \neq ys \vee x \neq y$

shows ((*rho* [*x* ← *X*]-*xs*) [*y* ← *Y*]-*ys*) = ((*rho* [*y* ← *Y*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **unfolding** *updEnv-def* **by** *fastforce*

theorem *freshEnv-updEnv-E1*:
assumes *freshEnv xs y* (*rho* [*x* ← *X*]-*xs*)
shows *y* ≠ *x*
using *assms* **unfolding** *freshEnv-def liftAll-def updEnv-def* **by** *auto*

theorem *freshEnv-updEnv-E2*:
assumes *freshEnv ys y* (*rho* [*x* ← *X*]-*xs*)
shows *fresh ys y X*
using *assms* **unfolding** *freshEnv-def liftAll-def updEnv-def*
by (*auto split*: *if-splits*)

theorem *freshEnv-updEnv-E3*:
assumes *freshEnv ys y* (*rho* [*x* ← *X*]-*xs*)
shows *rho ys y* = *None*
using *assms freshEnv-updEnv-E1*[*of ys y*] **unfolding** *freshEnv-def*
by (*metis getEnv-updEnv option.simps*(*3*))

theorem *freshEnv-updEnv-E4*:
assumes *freshEnv ys y* (*rho* [*x* ← *X*]-*xs*)
and *zs* ≠ *xs* ∨ *z* ≠ *x* **and** *rho zs z* = *Some Z*
shows *fresh ys y Z*
using *assms* **unfolding** *freshEnv-def liftAll-def*
by (*metis getEnv-updEnv1*)

theorem *freshEnv-updEnv-I*:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *fresh ys y X* **and** *rho ys y* = *None*
and ⋀ *zs z Z*. ⟦*zs* ≠ *xs* ∨ *z* ≠ *x*; *rho zs z* = *Some Z*⟧ ⟹ *fresh ys y Z*
shows *freshEnv ys y* (*rho* [*x* ← *X*]-*xs*)
unfolding *freshEnv-def liftAll-def*
using *assms* **by** *auto*

theorem *swapEnv-updEnv*:
((*rho* [*x* ← *X*]-*xs*) &[*y1* ∧ *y2*]-*ys*) =
((*rho* &[*y1* ∧ *y2*]-*ys*) [(*x* @*xs*[*y1* ∧ *y2*]-*ys*) ← (*X* #[*y1* ∧ *y2*]-*ys*)]-*xs*)
unfolding *swapEnv-defs sw-def lift-def*
by(*cases xs* = *ys*) *fastforce*+

lemma *swapEnv-updEnv-fresh*:
assumes *ys* ≠ *xs* ∨ *x* ∉ {*y1*,*y2*} **and** *good X*
and *fresh ys y1 X* **and** *fresh ys y2 X*
shows ((*rho* [*x* ← *X*]-*xs*) &[*y1* ∧ *y2*]-*ys*) =
((*rho* &[*y1* ∧ *y2*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **by**(*simp add*: *swapEnv-updEnv*)

theorem *psubstEnv-updEnv*:
((*rho* [*x* ← *X*]-*xs*) &[*rho'*]) = ((*rho* &[*rho'*]) [*x* ← (*X* #[*rho'*])]-*xs*)

unfolding *psubstEnv-def* **by** *fastforce*

theorem *psubstEnv-updEnv-idEnv*:
$((idEnv\ [x \leftarrow X]\text{-}xs)\ \&[rho]) = (rho\ [x \leftarrow (X\ \#[rho])]\text{-}xs)$
by(*simp add*: *psubstEnv-updEnv*)

theorem *substEnv-updEnv*:
$((rho\ [x \leftarrow X]\text{-}xs)\ \&[Y\ /\ y]\text{-}ys) = ((rho\ \&[Y\ /\ y]\text{-}ys)\ [x \leftarrow (X\ \#[Y\ /\ y]\text{-}ys)]\text{-}xs)$
unfolding *substEnv-def subst-def* **by**(*rule psubstEnv-updEnv*)

theorem *vsubstEnv-updEnv*:
$((rho\ [x \leftarrow X]\text{-}xs)\ \&[y1\ //\ y]\text{-}ys) = ((rho\ \&[y1\ //\ y]\text{-}ys)\ [x \leftarrow (X\ \#[y1\ //\ y]\text{-}ys)]\text{-}xs)$
unfolding *vsubstEnv-def vsubst-def* **using** *substEnv-updEnv* **.**

6.3 Environment “get" versus other operators

Currently, “get" is just function application. While the next properties are immediate consequences of the definitions, it is worth stating them because of their abstract character (since later, concrete terms inferred from abstract terms by a presumptive package, “get" will no longer be function application).

theorem *getEnv-ext*:
assumes $\bigwedge xs\ x.\ rho\ xs\ x = rho'\ xs\ x$
shows $rho = rho'$
using *assms* **by**(*simp add*: *ext*)

theorem *freshEnv-getEnv1* [*simp*]:
$[\![freshEnv\ ys\ y\ rho;\ rho\ xs\ x = Some\ X]\!] \Longrightarrow ys \neq xs \lor y \neq x$
unfolding *freshEnv-def* **by** *auto*

theorem *freshEnv-getEnv2* [*simp*]:
$[\![freshEnv\ ys\ y\ rho;\ rho\ xs\ x = Some\ X]\!] \Longrightarrow fresh\ ys\ y\ X$
unfolding *freshEnv-def liftAll-def* **by** *simp*

theorem *freshEnv-getEnv*[*simp*]:
$freshEnv\ ys\ y\ rho \Longrightarrow rho\ ys\ y = None$
unfolding *freshEnv-def* **by** *simp*

theorem *getEnv-swapEnv1* [*simp*]:
assumes $rho\ xs\ (x\ @xs\ [z1 \land z2]\text{-}zs) = None$
shows $(rho\ \&[z1 \land z2]\text{-}zs)\ xs\ x = None$
using *assms* **unfolding** *swapEnv-defs lift-def* **by** *simp*

theorem *getEnv-swapEnv2* [*simp*]:
assumes $rho\ xs\ (x\ @xs\ [z1 \land z2]\text{-}zs) = Some\ X$
shows $(rho\ \&[z1 \land z2]\text{-}zs)\ xs\ x = Some\ (X\ \#[z1 \land z2]\text{-}zs)$
using *assms* **unfolding** *swapEnv-defs lift-def* **by** *simp*

theorem *getEnv-psubstEnv-None*[*simp*]:
assumes *rho xs x* = *None*
shows (*rho* &[*rho'*]) *xs x* = *rho' xs x*
using *assms* **unfolding** *psubstEnv-def* **by** *simp*

theorem *getEnv-psubstEnv-Some*[*simp*]:
assumes *rho xs x* = *Some X*
shows (*rho* &[*rho'*]) *xs x* = *Some* (*X* #[*rho'*])
using *assms* **unfolding** *psubstEnv-def* **by** *simp*

theorem *getEnv-substEnv1*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *None*
shows (*rho* &[*Y* / *y*]-*ys*) *xs x* = *None*
using *assms* **unfolding** *substEnv-def2* **by** *auto*

theorem *getEnv-substEnv2*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *Some X*
shows (*rho* &[*Y* / *y*]-*ys*) *xs x* = *Some* (*X* #[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substEnv-def2* **by** *auto*

theorem *getEnv-substEnv3*[*simp*]:
⟦*ys* ≠ *xs* ∨ *y* ≠ *x*; *freshEnv xs x rho*⟧
⟹ (*rho* &[*Y* / *y*]-*ys*) *xs x* = *None*
using *getEnv-substEnv1* **by** *auto*

theorem *getEnv-substEnv4*[*simp*]:
freshEnv ys y rho ⟹ (*rho* &[*Y* / *y*]-*ys*) *ys y* = *Some Y*
unfolding *substEnv-psubstEnv-idEnv* **by** *simp*

theorem *getEnv-vsubstEnv1*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *None*
shows (*rho* &[*y1* // *y*]-*ys*) *xs x* = *None*
using *assms* **unfolding** *vsubstEnv-def* **by** *auto*

theorem *getEnv-vsubstEnv2*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *Some X*
shows (*rho* &[*y1* // *y*]-*ys*) *xs x* = *Some* (*X* #[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubstEnv-def vsubst-def* **by** *auto*

theorem *getEnv-vsubstEnv3*[*simp*]:
⟦*ys* ≠ *xs* ∨ *y* ≠ *x*; *freshEnv xs x rho*⟧
⟹ (*rho* &[*z* // *y*]-*ys*) *xs x* = *None*
using *getEnv-vsubstEnv1* **by** *auto*

theorem *getEnv-vsubstEnv4*[*simp*]:
freshEnv ys y rho ⟹ (*rho* &[*z* // *y*]-*ys*) *ys y* = *Some* (*Var ys z*)
unfolding *vsubstEnv-psubstEnv-idEnv* **by** *simp*

6.4 Substitution versus other operators

definition *freshImEnvAt* ::
'varSort ⇒ *'var* ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env* ⇒ *'varSort* ⇒ *'var*
⇒ *bool*
where
freshImEnvAt xs x rho ys y ==
 rho ys y = *None* ∧ (*ys* ≠ *xs* ∨ *y* ≠ *x*) ∨
 (∃ *Y*. *rho ys y* = *Some Y* ∧ *fresh xs x Y*)

lemma *freshAll-psubstAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
  A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
  P::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param* **and** *x*
assumes *goodP*: *goodPar P*
shows
(*good X* ⟶ *z* ∈ *varsOf P* ⟶
 (∀ *rho* ∈ *envsOf P*.
  fresh zs z (*X* #[*rho*]) =
  (∀ *ys*. ∀ *y*. *fresh ys y X* ∨ *freshImEnvAt zs z rho ys y*)))
 ∧
 (*goodAbs A* ⟶ *z* ∈ *varsOf P* ⟶
 (∀ *rho* ∈ *envsOf P*.
  freshAbs zs z (*A* $[*rho*]) =
  (∀ *ys*. ∀ *y*. *freshAbs ys y A* ∨ *freshImEnvAt zs z rho ys y*)))
proof(*induction rule*: *term-induct-fresh*[*of P*])
 case *Par*
 then show *?case* **using** *goodP* **by** *simp*
next
 case (*Var ys y*)
 thus *?case* **proof** *clarify*
  fix *rho*
  assume *r*: *rho* ∈ *envsOf P*
  hence *g*: *goodEnv rho* **using** *goodP* **by** *simp*
  thus *fresh zs z* (*psubst rho* (*Var ys y*)) =
  (∀ *ysa ya*. *fresh ysa ya* (*Var ys y*) ∨ *freshImEnvAt zs z rho ysa ya*)
  unfolding *freshImEnvAt-def*
  by(*cases ys* = *zs* ∧ *y* = *z*, (*cases rho ys y*, *auto*)+)
 qed
next
 case (*Op delta inp binp*)
 show *?case* **proof** *clarify*
  fix *rho*
  assume *P*: *z* ∈ *varsOf P rho* ∈ *envsOf P*
  let *?L1* = *liftAll* (*fresh zs z* ∘ *psubst rho*) *inp*
  let *?L2* = *liftAll* (*freshAbs zs z* ∘ *psubstAbs rho*) *binp*
  let *?R1* = %*ys y*. *liftAll* (*fresh ys y*) *inp*
  let *?R2* = %*ys y*. *liftAll* (*freshAbs ys y*) *binp*
  let *?R3* = %*ys y*. *freshImEnvAt zs z rho ys y*
  have (*?L1* ∧ *?L2*) = (∀ *ys y*. *?R1 ys y* ∧ *?R2 ys y* ∨ *?R3 ys y*)

using *Op.IH P* **unfolding** *liftAll-def* **by** *simp blast*
thus *fresh zs z* ((*Op delta inp binp*) #[*rho*]) =
(∀ *ys y. fresh ys y* (*Op delta inp binp*) ∨ *freshImEnvAt zs z rho ys y*)
by (*metis* (*no-types*, *lifting*) *Op.hyps*(*1*) *Op.hyps*(*2*) *P*(*2*) *envsOf-preserves-good freshBinp-def freshInp-def fresh-Op-simp goodP liftAll-lift-comp psubstBinp-def psubstBinp-preserves-good*
psubstInp-def psubstInp-preserves-good psubst-Op-simp)
qed
next
case (*Abs xs x X*)
thus *?case*
using *goodP* **by** *simp* (*metis* (*full-types*) *freshEnv-def freshImEnvAt-def*)
qed

corollary *fresh-psubst*:
assumes *good X* **and** *goodEnv rho*
shows
fresh zs z (*X* #[*rho*]) =
(∀ *ys y. fresh ys y X* ∨ *freshImEnvAt zs z rho ys y*)
using *assms freshAll-psubstAll*[*of Par* [*z*] [] [] [*rho*]]
unfolding *goodPar-def* **by** *simp*

corollary *fresh-psubst-E1*:
assumes *good X* **and** *goodEnv rho*
and *rho ys y* = *None* **and** *fresh zs z* (*X* #[*rho*])
shows *fresh ys y X* ∨ (*ys* ≠ *zs* ∨ *y* ≠ *z*)
using *assms fresh-psubst* **unfolding** *freshImEnvAt-def* **by** *fastforce*

corollary *fresh-psubst-E2*:
assumes *good X* **and** *goodEnv rho*
and *rho ys y* = *Some Y* **and** *fresh zs z* (*X* #[*rho*])
shows *fresh ys y X* ∨ *fresh zs z Y*
using *assms fresh-psubst*[*of X rho*] **unfolding** *freshImEnvAt-def* **by** *fastforce*

corollary *fresh-psubst-I1*:
assumes *good X* **and** *goodEnv rho*
and *fresh zs z X* **and** *freshEnv zs z rho*
shows *fresh zs z* (*X* #[*rho*])
using *assms* **apply**(*simp add*: *fresh-psubst*)
unfolding *freshEnv-def liftAll-def freshImEnvAt-def* **by** *auto*

corollary *psubstEnv-preserves-freshEnv*:
assumes *good*: *goodEnv rho goodEnv rho′*
and *fresh*: *freshEnv zs z rho freshEnv zs z rho′*
shows *freshEnv zs z* (*rho* &[*rho′*])
using *assms* **unfolding** *freshEnv-def liftAll-def*
by (*smt* (*verit*, *del-insts*) *fresh*(*2*) *fresh-psubst-I1 getEnv-preserves-good getEnv-psubstEnv-None getEnv-psubstEnv-Some not-None-eq option.inject*)

corollary *fresh-psubst-I*:
assumes *good X* **and** *goodEnv rho*
and *rho zs z = None* $\Longrightarrow$ *fresh zs z X* **and**
$\bigwedge$ *ys y Y*. *rho ys y = Some Y* $\Longrightarrow$ *fresh ys y X* $\vee$ *fresh zs z Y*
shows *fresh zs z* (*X* #[*rho*])
using *assms* **unfolding** *freshImEnvAt-def*
by (*simp add*: *fresh-psubst*) (*metis freshImEnvAt-def not-None-eq*)

lemma *fresh-subst*:
assumes *good X* **and** *good Y*
shows *fresh zs z* (*X* #[*Y* / *y*]-*ys*) =
(((*zs* = *ys* $\wedge$ *z* = *y*) $\vee$ *fresh zs z X*) $\wedge$ (*fresh ys y X* $\vee$ *fresh zs z Y*))
using *assms* **unfolding** *subst-def freshImEnvAt-def*
by (*simp add*: *fresh-psubst*)
(*metis* (*no-types*, *lifting*) *freshImEnvAt-def fresh-psubst fresh-psubst-E2 getEnv-updEnv-idEnv idEnv-preserves-good option.simps*(*3*) *updEnv-preserves-good*)

lemma *fresh-vsubst*:
assumes *good X*
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*) =
(((*zs* = *ys* $\wedge$ *z* = *y*) $\vee$ *fresh zs z X*) $\wedge$ (*fresh ys y X* $\vee$ (*zs* $\neq$ *ys* $\vee$ *z* $\neq$ *y1*)))
unfolding *vsubst-def* **using** *assms* **by**(*auto simp*: *fresh-subst*)

lemma *subst-preserves-fresh*:
assumes *good X* **and** *good Y*
and *fresh zs z X* **and** *fresh zs z Y*
shows *fresh zs z* (*X* #[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *fresh-subst*)

lemma *substEnv-preserves-freshEnv-aux*:
assumes *rho*: *goodEnv rho* **and** *Y*: *good Y*
and *fresh-rho*: *freshEnv zs z rho* **and** *fresh-Y*: *fresh zs z Y* **and** *diff*: *zs* $\neq$ *ys* $\vee$ *z* $\neq$ *y*
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
using *assms* **unfolding** *freshEnv-def liftAll-def*
by (*simp add*: *option.case-eq-if substEnv-def2 subst-preserves-fresh*)

lemma *substEnv-preserves-freshEnv*:
assumes *rho*: *goodEnv rho* **and** *Y*: *good Y*
and *fresh-rho*: *freshEnv zs z rho* **and** *fresh-Y*: *fresh zs z Y* **and** *diff*: *zs* $\neq$ *ys* $\vee$ *z* $\neq$ *y*
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-preserves-freshEnv-aux*)

lemma *vsubst-preserves-fresh*:
assumes *good X*
and *fresh zs z X* **and** *zs* $\neq$ *ys* $\vee$ *z* $\neq$ *y1*
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *fresh-vsubst*)

lemma *vsubstEnv-preserves-freshEnv*:
assumes *rho*: *goodEnv rho*
and *fresh-rho*: *freshEnv zs z rho* **and** *diff*: $zs \neq ys \vee z \notin \{y,y1\}$
shows *freshEnv zs z* (*rho* &[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubstEnv-def*
by(*simp add*: *substEnv-preserves-freshEnv*)

lemma *fresh-fresh-subst*[*simp*]:
assumes *good Y* **and** *good X*
and *fresh ys y Y*
shows *fresh ys y* (*X* #[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *fresh-subst*)

lemma *diff-fresh-vsubst*[*simp*]:
assumes *good X*
and $y \neq y1$
shows *fresh ys y* (*X* #[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *fresh-vsubst*)

lemma *fresh-subst-E1*:
assumes *good X* **and** *good Y*
and *fresh zs z* (*X* #[*Y* / *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *fresh zs z X*
using *assms* **by**(*auto simp add*: *fresh-subst*)

lemma *fresh-vsubst-E1*:
assumes *good X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *fresh zs z X*
using *assms* **by**(*auto simp add*: *fresh-vsubst*)

lemma *fresh-subst-E2*:
assumes *good X* **and** *good Y*
and *fresh zs z* (*X* #[*Y* / *y*]-*ys*)
shows *fresh ys y X* ∨ *fresh zs z Y*
using *assms* **by**(*simp add*: *fresh-subst*)

lemma *fresh-vsubst-E2*:
assumes *good X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
shows *fresh ys y X* ∨ $zs \neq ys \vee z \neq y1$
using *assms* **by**(*simp add*: *fresh-vsubst*)

lemma *psubstAll-cong*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
P::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param*
assumes *goodP*: *goodPar P*

shows
(*good X* $\longrightarrow$
  ($\forall$ *rho rho'*. {*rho*, *rho'*} $\subseteq$ *envsOf P* $\longrightarrow$
   ($\forall$ *ys*. $\forall$ *y*. *fresh ys y X* $\vee$ *rho ys y* = *rho' ys y*) $\longrightarrow$
            (*X* #[*rho*]) = (*X* #[*rho'*])))
$\wedge$
 (*goodAbs A* $\longrightarrow$
  ($\forall$ *rho rho'*. {*rho*, *rho'*} $\subseteq$ *envsOf P* $\longrightarrow$
   ($\forall$ *ys*. $\forall$ *y*. *freshAbs ys y A* $\vee$ *rho ys y* = *rho' ys y*) $\longrightarrow$
            (*A* $[*rho*]) = (*A* $[*rho'*])))
proof(*induction rule*: *term-induct-fresh*[*of P*])
  case *Par*
  then show *?case* **using** *assms* **.**
next
  case (*Var xs x*)
  then show *?case* **using** *goodP* **by** (*auto simp*: *psubst-Var*)
next
  case (*Op delta inp binp*)
  show *?case* **proof** *clarify*
    fix *rho rho'*
    assume *envs*: {*rho*, *rho'*} $\subseteq$ *envsOf P*
    hence *goodEnv*: *goodEnv rho* $\wedge$ *goodEnv rho'* **using** *goodP* **by** *simp*
    assume $\forall$ *ys y*. *fresh ys y* (*Op delta inp binp*) $\vee$ *rho ys y* = *rho' ys y*
    hence *1*: *liftAll* (λ *X*. $\forall$ *ys y*. *fresh ys y X* $\vee$ *rho ys y* = *rho' ys y*) *inp* $\wedge$
          liftAll (λ *A*. $\forall$ *ys y*. *freshAbs ys y A* $\vee$ *rho ys y* = *rho' ys y*) *binp*
    using *Op* **by** *simp* (*smt* (*verit*) *freshBinp-def freshInp-def liftAll-def*)
    have *liftAll* (λ *X*. (*X* #[*rho*]) = (*X* #[*rho'*])) *inp* $\wedge$
        liftAll (λ *A*. (*A* $[*rho*]) = (*A* $[*rho'*])) *binp*
    using *Op.IH 1 envs* **by** (*auto simp*: *liftAll-def*)
    thus (*Op delta inp binp*) #[*rho*] = (*Op delta inp binp*) #[*rho'*]
    using *Op.IH 1*
    by (*simp add*: *Op.hyps goodEnv psubstBinp-def psubstInp-def liftAll-lift-ext*)
  qed
next
  case (*Abs xs x X*)
  thus *?case*
  using *Abs goodP* **unfolding** *freshEnv-def liftAll-def*
  by *simp* (*metis Abs.hyps*(*5*) *envsOf-preserves-good psubstAbs-simp*)
qed

corollary *psubst-cong*[*fundef-cong*]:
assumes *good X* **and** *goodEnv rho* **and** *goodEnv rho'*
and $\bigwedge$ *ys y*. *fresh ys y X* $\vee$ *rho ys y* = *rho' ys y*
shows (*X* #[*rho*]) = (*X* #[*rho'*])
using *assms psubstAll-cong*[*of Par* [] [] [] [*rho*,*rho'*]]
unfolding *goodPar-def* **by** *simp*

lemma *fresh-psubst-updEnv*:
assumes *good X* **and** *good Y* **and** *goodEnv rho*
and *fresh xs x Y*
shows (*Y* #[*rho* [*x* ← *X*]-*xs*]) = (*Y* #[*rho*])
using *assms* **by** (*auto cong*: *psubst-cong*)

lemma *psubstAll-ident*:
fixes *X* :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* **and**
 A :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* **and**
 P :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*) *Transition-QuasiTerms-Terms.param*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
 (∀ *rho* ∈ *envsOf P*.
 (∀ *zs z*. *freshEnv zs z rho* ∨ *fresh zs z X*)
 ⟶ (*X* #[*rho*]) = *X*))
 ∧
 (*goodAbs A* ⟶
 (∀ *rho* ∈ *envsOf P*.
 (∀ *zs z*. *freshEnv zs z rho* ∨ *freshAbs zs z A*)
 ⟶ (*A* $[*rho*]) = *A*))
proof(*induction rule*: *term-induct-fresh*)
 case (*Var xs x*)
 then show *?case*
 by (*meson assms freshEnv-def fresh-Var-simp goodPar-def psubst-Var-simp1*)
next
 case (*Op delta inp binp*)
 then show *?case*
 by (*metis* (*no-types*,*lifting*) *Op-preserves-good assms envsOf-preserves-good freshEnv-getEnv idEnv-def idEnv-preserves-good psubst-cong psubst-idEnv*)
qed(*insert P*, *fastforce*+)

corollary *freshEnv-psubst-ident*[*simp*]:
fixes *X* :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term*
assumes *good X* **and** *goodEnv rho*
and ⋀ *zs z*. *freshEnv zs z rho* ∨ *fresh zs z X*
shows (*X* #[*rho*]) = *X*
using *assms psubstAll-ident*[*of Par* [] [] [] [*rho*]]
unfolding *goodPar-def* **by** *simp*

lemma *fresh-subst-ident*[*simp*]:
assumes *good X* **and** *good Y* **and** *fresh xs x Y*
shows (*Y* #[*X* / *x*]-*xs*) = *Y*
by (*simp add*: *assms fresh-psubst-updEnv subst-def*)

corollary *substEnv-updEnv-fresh*:
assumes *good X* **and** *good Y* **and** *fresh ys y X*
shows ((*rho* [*x* ← *X*]-*xs*) &[*Y* / *y*]-*ys*) = ((*rho* &[*Y* / *y*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **by**(*simp add*: *substEnv-updEnv*)

lemma *fresh-substEnv-updEnv*[*simp*]:
assumes *rho*: *goodEnv rho* **and** *Y*: *good Y*
and ∗: *freshEnv ys y rho*
shows (*rho* &[*Y* / *y*]-*ys*) = (*rho* [*y* ← *Y*]-*ys*)
apply (*rule getEnv-ext*)
subgoal for *xs x* **using** *assms* **by** (*cases rho xs x*) *auto* **.**

lemma *fresh-vsubst-ident*[*simp*]:
assumes *good Y* **and** *fresh xs x Y*
shows (*Y* #[*x1* // *x*]-*xs*) = *Y*
using *assms* **unfolding** *vsubst-def* **by** *simp*

corollary *vsubstEnv-updEnv-fresh*:
assumes *good X* **and** *fresh ys y X*
shows ((*rho* [*x* ← *X*]-*xs*) &[*y1* // *y*]-*ys*) = ((*rho* &[*y1* // *y*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **by**(*simp add*: *vsubstEnv-updEnv*)

lemma *fresh-vsubstEnv-updEnv*[*simp*]:
assumes *rho*: *goodEnv rho*
and ∗: *freshEnv ys y rho*
shows (*rho* &[*y1* // *y*]-*ys*) = (*rho* [*y* ← *Var ys y1*]-*ys*)
using *assms* **unfolding** *vsubstEnv-def* **by** *simp*

lemma *swapAll-psubstAll*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
P::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
(∀ *rho z1 z2*. *rho* ∈ *envsOf P* ∧ {*z1*,*z2*} ⊆ *varsOf P* ⟶
((*X* #[*rho*]) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[*rho* &[*z1* ∧
z2]-*zs*])))
∧
(*goodAbs A* ⟶
(∀ *rho z1 z2*. *rho* ∈ *envsOf P* ∧ {*z1*,*z2*} ⊆ *varsOf P* ⟶
((*A* $[*rho*]) $[*z1* ∧ *z2*]-*zs*) = ((*A* $[*z1* ∧ *z2*]-*zs*) $[*rho* &[*z1* ∧ *z2*]-*zs*])))
proof(*induction rule*: *term-induct-fresh*[*of P*])
case (*Var xs x*)
then show *?case* **using** *assms*
by *simp* (*smt* (*verit*) *Var-preserves-good envsOf-preserves-good getEnv-swapEnv1 getEnv-swapEnv2 option.case-eq-if option.exhaust-sel psubst-Var psubst-Var-simp2 swapEnv-preserves-good*
swap-Var-simp swap-involutive2 swap-sym)
next
case (*Op delta inp binp*)
then show *?case*
using *assms*

unfolding *psubstInp-def swapInp-def psubstBinp-def swapBinp-def lift-comp*
unfolding *liftAll-def lift-def*
by *simp* (*auto simp*: *lift-def psubstInp-def swapInp-def*
psubstBinp-def swapBinp-def split: *option.splits*)
qed(*insert assms*, *auto*)

lemma *swap-psubst*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[*rho* &[*z1* ∧ *z2*]-*zs*])
using *assms swapAll-psubstAll*[*of Par* [*z1*,*z2*] [] [] [*rho*]]
unfolding *goodPar-def* **by** *auto*

lemma *swap-subst*:
assumes *good X* **and** *good Y*
shows ((*X* #[*Y* / *y*]-*ys*) #[*z1* ∧ *z2*]-*zs*) =
((*X* #[*z1* ∧ *z2*]-*zs*) #[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
proof−
have *1*: (*idEnv* [(*y* @*ys*[*z1* ∧ *z2*]-*zs*) ← (*Y* #[*z1* ∧ *z2*]-*zs*)]-*ys*) =
((*idEnv* [*y* ← *Y*]-*ys*) &[*z1* ∧ *z2*]-*zs*)
by(*simp add*: *swapEnv-updEnv*)
show *?thesis*
using *assms* **unfolding** *subst-def 1* **by** (*intro swap-psubst*) *auto*
qed

lemma *swap-vsubst*:
assumes *good X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*z1* ∧ *z2*]-*zs*) =
((*X* #[*z1* ∧ *z2*]-*zs*) #[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **unfolding** *vsubst-def*
by(*simp add*: *swap-subst*)

lemma *swapEnv-psubstEnv*:
assumes *goodEnv rho* **and** *goodEnv rho'*
shows ((*rho* &[*rho'*]) &[*z1* ∧ *z2*]-*zs*) = ((*rho* &[*z1* ∧ *z2*]-*zs*) &[*rho'* &[*z1* ∧ *z2*]-*zs*])

using *assms* **apply**(*intro ext*)
subgoal for *xs x*
by (*cases rho xs* (*x* @*xs*[*z1* ∧ *z2*]-*zs*))
(*auto simp*: *lift-def swapEnv-defs swap-psubst*) **.**

lemma *swapEnv-substEnv*:
assumes *good Y* **and** *goodEnv rho*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*z1* ∧ *z2*]-*zs*) =
((*rho* &[*z1* ∧ *z2*]-*zs*) &[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
proof−
have *1*: (*idEnv* [(*y* @*ys*[*z1* ∧ *z2*]-*zs*) ← (*Y* #[*z1* ∧ *z2*]-*zs*)]-*ys*) =
((*idEnv* [*y* ← *Y*]-*ys*) &[*z1* ∧ *z2*]-*zs*)
by(*simp add*: *swapEnv-updEnv*)
show *?thesis*

unfolding *substEnv-def 1*
using *assms* **by** (*intro swapEnv-psubstEnv*) *auto*
qed

lemma *swapEnv-vsubstEnv*:
assumes *goodEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*z1* ∧ *z2*]-*zs*) =
((*rho* &[*z1* ∧ *z2*]-*zs*) &[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **unfolding** *vsubstEnv-def* **by**(*simp add*: *swapEnv-substEnv*)

lemma *psubstAll-compose*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
P::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
(∀ *rho rho'*. {*rho*,*rho'*} ⊆ *envsOf P* ⟶ ((*X* #[*rho*]) #[*rho'*]) = (*X* #[(*rho* &[*rho'*])])))
∧
(*goodAbs A* ⟶
(∀ *rho rho'*. {*rho*,*rho'*} ⊆ *envsOf P* ⟶ ((*A* $[*rho*]) $[*rho'*]) = (*A* $[(*rho* &[*rho'*])])))
proof(*induction rule*: *term-induct-fresh*[*of P*])
case (*Var xs x*)
then show *?case* **using** *assms*
by *simp* (*smt* (*verit*, *del-insts*) *Var-preserves-good case-optionE envsOf-preserves-good option.case-distrib option.simps*(*4*) *option.simps*(*5*)
psubstEnv-def psubstEnv-preserves-good psubst-Var psubst-preserves-good)
next
case (*Op delta inp binp*)
then show *?case*
using *assms*
unfolding *psubstInp-def swapInp-def psubstBinp-def swapBinp-def lift-comp*
unfolding *liftAll-def lift-def*
by *simp* (*auto simp*: *lift-def psubstInp-def swapInp-def*
psubstBinp-def swapBinp-def split: *option.splits*)
qed(*insert assms*, *simp-all add*: *psubstEnv-preserves-freshEnv*)

corollary *psubst-compose*:
assumes *good X* **and** *goodEnv rho* **and** *goodEnv rho'*
shows ((*X* #[*rho*]) #[*rho'*]) = (*X* #[(*rho* &[*rho'*])])
using *assms psubstAll-compose*[*of Par* [] [] [] [*rho*, *rho'*]]
unfolding *goodPar-def* **by** *auto*

lemma *psubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho'* **and** *goodEnv rho''*
shows ((*rho* &[*rho'*]) &[*rho''*]) = (*rho* &[(*rho'* &[*rho''*])])
using *assms* **apply**(*intro ext*)
subgoal for *xs x*

by (*cases rho xs x*) (*auto simp*: *lift-def psubstEnv-def psubst-compose*) **.**

lemma *psubst-subst-compose*:
assumes *good X* **and** *good Y* **and** *goodEnv rho*
shows ((*X* #[*Y* / *y*]-*ys*) #[*rho*]) = (*X* #[(*rho* [*y* ← (*Y* #[*rho*])]-*ys*)])
by (*simp add*: *assms psubstEnv-updEnv-idEnv psubst-compose subst-psubst-idEnv*)

lemma *psubstEnv-substEnv-compose*:
assumes *goodEnv rho* **and** *good Y* **and** *goodEnv rho′*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*rho′*]) = (*rho* &[(*rho′* [*y* ← (*Y* #[*rho′*])]-*ys*)])
by (*simp add*: *assms psubstEnv-compose psubstEnv-updEnv-idEnv substEnv-def*)

lemma *psubst-vsubst-compose*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*y1* // *y*]-*ys*) #[*rho*]) = (*X* #[(*rho* [*y* ← ((*Var ys y1*) #[*rho*])]-*ys*)])
using *assms* **unfolding** *vsubst-def* **by**(*simp add*: *psubst-subst-compose*)

lemma *psubstEnv-vsubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho′*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*rho′*]) = (*rho* &[(*rho′* [*y* ← ((*Var ys y1*) #[*rho′*])]-*ys*)])
using *assms* **unfolding** *vsubstEnv-def* **by**(*simp add*: *psubstEnv-substEnv-compose*)

lemma *subst-psubst-compose*:
assumes *good X* **and** *good Y* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*Y* / *y*]-*ys*) = (*X* #[(*rho* &[*Y* / *y*]-*ys*)])
unfolding *subst-def substEnv-def* **using** *assms* **by**(*simp add*: *psubst-compose*)

lemma *substEnv-psubstEnv-compose*:
assumes *goodEnv rho* **and** *good Y* **and** *goodEnv rho′*
shows ((*rho* &[*rho′*]) &[*Y* / *y*]-*ys*) = (*rho* &[(*rho′* &[*Y* / *y*]-*ys*)])
unfolding *substEnv-def* **using** *assms* **by**(*simp add*: *psubstEnv-compose*)

lemma *psubst-subst-compose-freshEnv*:
assumes *goodEnv rho* **and** *good X* **and** *good Y*
assumes *freshEnv ys y rho*
shows ((*X* #[*Y* / *y*]-*ys*) #[*rho*]) = ((*X* #[*rho*]) #[(*Y* #[*rho*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *subst-psubst-compose psubst-subst-compose*)

lemma *psubstEnv-substEnv-compose-freshEnv*:
assumes *goodEnv rho* **and** *goodEnv rho′* **and** *good Y*
assumes *freshEnv ys y rho′*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*rho′*]) = ((*rho* &[*rho′*]) &[(*Y* #[*rho′*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *substEnv-psubstEnv-compose psubstEnv-substEnv-compose*)

lemma *vsubst-psubst-compose*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*y1* // *y*]-*ys*) = (*X* #[(*rho* &[*y1* // *y*]-*ys*)])
unfolding *vsubst-def vsubstEnv-def* **using** *assms* **by**(*simp add*: *subst-psubst-compose*)

lemma *vsubstEnv-psubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho'*
shows ((*rho* &[*rho'*]) &[*y1* // *y*]-*ys*) = (*rho* &[(*rho'* &[*y1* // *y*]-*ys*)])
unfolding *vsubstEnv-def* **using** *assms* **by**(*simp add*: *substEnv-psubstEnv-compose*)

lemma *subst-compose1*:
assumes *good X* **and** *good Y1* **and** *good Y2*
shows ((*X* #[*Y1* / *y*]-*ys*) #[*Y2* / *y*]-*ys*) = (*X* #[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
proof−
 have *goodEnv* (*idEnv* [*y* ← *Y1*]-*ys*) ∧ *goodEnv* (*idEnv* [*y* ← *Y2*]-*ys*) **using** *assms*
by *simp*
 thus *?thesis* **using** ‹*good X*› **unfolding** *subst-def substEnv-def*
 by(*simp add*: *psubst-compose psubstEnv-updEnv*)
qed

lemma *substEnv-compose1*:
assumes *goodEnv rho* **and** *good Y1* **and** *good Y2*
shows ((*rho* &[*Y1* / *y*]-*ys*) &[*Y2* / *y*]-*ys*) = (*rho* &[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
by (*simp add*: *assms psubstEnv-compose psubstEnv-updEnv-idEnv substEnv-def subst-psubst-idEnv*)

lemma *subst-vsubst-compose1*:
assumes *good X* **and** *good Y* **and** *y* ≠ *y1*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y*]-*ys*) = (*X* #[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def* **by**(*simp add*: *subst-compose1*)

lemma *substEnv-vsubstEnv-compose1*:
assumes *goodEnv rho* **and** *good Y* **and** *y* ≠ *y1*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Y* / *y*]-*ys*) = (*rho* &[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstEnv-def* **by**(*simp add*: *substEnv-compose1*)

lemma *vsubst-subst-compose1*:
assumes *good X* **and** *good Y*
shows ((*X* #[*Y* / *y*]-*ys*) #[*y1* // *y*]-*ys*) = (*X* #[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **unfolding** *vsubst-def* **by**(*simp add*: *subst-compose1*)

lemma *vsubstEnv-substEnv-compose1*:
assumes *goodEnv rho* **and** *good Y*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*y1* // *y*]-*ys*) = (*rho* &[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstEnv-def* **by**(*simp add*: *substEnv-compose1*)

lemma *vsubst-compose1*:
assumes *good X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y*]-*ys*) = (*X* #[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def*
by(*cases y* = *y1*) (*auto simp*: *subst-compose1*)

lemma *vsubstEnv-compose1*:
assumes *goodEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*y2* // *y*]-*ys*) = (*rho* &[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)

using *assms* **unfolding** *vsubstEnv-def*
by(*cases y = y1*) (*auto simp*: *substEnv-compose1*)

lemma *subst-compose2*:
assumes *good X* **and** *good Y* **and** *good Z*
and *ys ≠ zs ∨ y ≠ z* **and** *fresh*: *fresh ys y Z*
shows ((*X* #[*Y* / *y*]-*ys*) #[*Z* / *z*]-*zs*) = ((*X* #[*Z* / *z*]-*zs*) #[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
by (*metis assms fresh freshEnv-getEnv freshEnv-getEnv2 freshEnv-idEnv freshEnv-updEnv-I idEnv-preserves-good psubst-subst-compose-freshEnv subst-psubst-idEnv updEnv-preserves-good*)

lemma *substEnv-compose2*:
assumes *goodEnv rho* **and** *good Y* **and** *good Z*
and *ys ≠ zs ∨ y ≠ z* **and** *fresh*: *fresh ys y Z*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*Z* / *z*]-*zs*) = ((*rho* &[*Z* / *z*]-*zs*) &[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
by (*metis assms fresh freshEnv-updEnv-I getEnv-idEnv idEnv-preserves-good option.discI psubstEnv-substEnv-compose-freshEnv substEnv-def subst-psubst-idEnv updEnv-preserves-good*)

lemma *subst-vsubst-compose2*:
assumes *good X* **and** *good Z*
and *ys ≠ zs ∨ y ≠ z* **and** *fresh*: *fresh ys y Z*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Z* / *z*]-*zs*) = ((*X* #[*Z* / *z*]-*zs*) #[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubst-def* **by**(*simp add*: *subst-compose2*)

lemma *substEnv-vsubstEnv-compose2*:
assumes *goodEnv rho* **and** *good Z*
and *ys ≠ zs ∨ y ≠ z* **and** *fresh*: *fresh ys y Z*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Z* / *z*]-*zs*) = ((*rho* &[*Z* / *z*]-*zs*) &[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubstEnv-def* **by**(*simp add*: *substEnv-compose2*)

lemma *vsubst-subst-compose2*:
assumes *good X* **and** *good Y*
and *ys ≠ zs ∨ y ∉ {z,z1}*
shows ((*X* #[*Y* / *y*]-*ys*) #[*z1* // *z*]-*zs*) = ((*X* #[*z1* // *z*]-*zs*) #[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubst-def* **by**(*simp add*: *subst-compose2*)

lemma *vsubstEnv-substEnv-compose2*:
assumes *goodEnv rho* **and** *good Y*
and *ys ≠ zs ∨ y ∉ {z,z1}*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*z1* // *z*]-*zs*) = ((*rho* &[*z1* // *z*]-*zs*) &[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstEnv-def* **by**(*simp add*: *substEnv-compose2*)

lemma *vsubst-compose2*:
assumes *good X*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*X* #[*y1* // *y*]-*ys*) #[*z1* // *z*]-*zs*) =
((*X* #[*z1* // *z*]-*zs*) #[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
by (*metis vsubst-def Var-preserves-good assms vsubst-Var-simp vsubst-def vsubst-subst-compose2*)

lemma *vsubstEnv-compose2*:
assumes *goodEnv rho*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*rho* &[*y1* // *y*]-*ys*) &[*z1* // *z*]-*zs*) =
((*rho* &[*z1* // *z*]-*zs*) &[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
by (*metis Var-preserves-good assms*
vsubstEnv-def vsubstEnv-substEnv-compose2 vsubst-Var-simp)

6.5 Properties specific to variable-for-variable substitution

lemma *vsubstAll-ident*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and**
P::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*param* **and** *zs*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
(∀ *z*. *z* ∈ *varsOf P* ⟶ (*X* #[*z* // *z*]-*zs*) = *X*))
∧
(*goodAbs A* ⟶
(∀ *z*. *z* ∈ *varsOf P* ⟶ (*A* $[*z* // *z*]-*zs*) = *A*))
proof(*induct rule*: *term-induct-fresh*[*of P*])
case (*Op delta inp binp*)
then show *?case*
using *assms*
unfolding *vsubst-def vsubstAbs-def liftAll-def lift-def*
by *simp* (*auto simp*: *lift-def substInp-def2 substBinp-def2 vsubstInp-def2 split*: *option.splits*)
next
case (*Abs xs x X*)
then show *?case*
by (*metis empty-iff insert-iff vsubstAbs-simp*)
qed(*insert assms*, *simp-all*)

corollary *vsubst-ident*[*simp*]:
assumes *good X*
shows (*X* #[*z* // *z*]-*zs*) = *X*
using *assms vsubstAll-ident*[*of Par* [*z*] [] [] [] *X*]
unfolding *goodPar-def* **by** *simp*

corollary *subst-ident*[*simp*]:

assumes *good X*
shows (*X* #[(*Var zs z*) / *z*]-*zs*) = *X*
using *assms vsubst-ident* **unfolding** *vsubst-def* **by** *auto*

lemma *vsubstAll-swapAll*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* **and**
P::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*param* **and** *ys*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
(∀ *y1 y2*. {*y1*,*y2*} ⊆ *varsOf P* ∧ *fresh ys y1 X* ⟶
(*X* #[*y1* // *y2*]-*ys*) = (*X* #[*y1* ∧ *y2*]-*ys*)))
∧
(*goodAbs A* ⟶
(∀ *y1 y2*. {*y1*,*y2*} ⊆ *varsOf P* ∧ *freshAbs ys y1 A* ⟶
(*A* $[*y1* // *y2*]-*ys*) = (*A* $[*y1* ∧ *y2*]-*ys*)))
apply(*induction rule*: *term-induct-fresh*[*OF P*])
subgoal by (*force simp add*: *sw-def*)
subgoal by *simp* (*auto*
simp: *vsubstInp-def substInp-def2 vsubst-def swapInp-def*
vsubstBinp-def substBinp-def2 vsubstAbs-def swapBinp-def
freshInp-def freshBinp-def lift-def liftAll-def
split: *option.splits*)
subgoal by *simp* (*metis Var-preserves-good fresh-Var-simp substAbs-simp sw-def*
vsubstAbs-def vsubst-def) **.**

corollary *vsubst-eq-swap*:
assumes *good X* **and** *y1* = *y2* ∨ *fresh ys y1 X*
shows (*X* #[*y1* // *y2*]-*ys*) = (*X* #[*y1* ∧ *y2*]-*ys*)
apply(*cases y1* = *y2*)
using *assms vsubstAll-swapAll*[*of Par* [*y1*, *y2*] [] [] [] *X*]
unfolding *goodPar-def* **by** *auto*

lemma *skelAll-vsubstAll*:
fixes *X*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* **and**
P::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*param* **and** *ys*
assumes *P*: *goodPar P*
shows
(*good X* ⟶
(∀ *y1 y2*. {*y1*,*y2*} ⊆ *varsOf P* ⟶
skel (*X* #[*y1* // *y2*]-*ys*) = *skel X*))
∧
(*goodAbs A* ⟶
(∀ *y1 y2*. {*y1*,*y2*} ⊆ *varsOf P* ⟶
skelAbs (*A* $[*y1* // *y2*]-*ys*) = *skelAbs A*))
proof(*induction rule*: *term-induct-fresh*[*of P*])
case (*Op delta inp binp*)

then show *?case*
by (*simp add*: *skelInp-def2 skelBinp-def2*)
(*auto simp*: *vsubst-def vsubstInp-def substInp-def2*
vsubstAbs-def vsubstBinp-def substBinp-def2 lift-def liftAll-def
split: *option.splits*)
next
case (*Abs xs x X*)
then show *?case* **using** *assms*
by *simp* (*metis not-equals-and-not-equals-not-in*
skelAbs-simp vsubstAbs-simp vsubst-preserves-good)
qed(*insert assms*, *simp-all*)

corollary *skel-vsubst*:
assumes *good X*
shows *skel* (*X* #[*y1* // *y2*]-*ys*) = *skel X*
using *assms skelAll-vsubstAll*[*of Par* [*y1*, *y2*] [] [] [] *X*]
unfolding *goodPar-def* **by** *simp*

lemma *subst-vsubst-trans*:
assumes *good X* **and** *good Y* **and** *fresh ys y1 X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y1*]-*ys*) = (*X* #[*Y* / *y*]-*ys*)
using *assms* **unfolding** *subst-def vsubst-def*
by (*cases y1* = *y*) (*simp-all add*: *fresh-psubst-updEnv psubstEnv-updEnv-idEnv*
psubst-compose updEnv-commute)

lemma *vsubst-trans*:
assumes *good X* **and** *fresh ys y1 X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y1*]-*ys*) = (*X* #[*y2* // *y*]-*ys*)
unfolding *vsubst-def*[*of - y2 y1*] *vsubst-def*[*of - y2 y*]
using *assms* **by**(*simp add*: *subst-vsubst-trans*)

lemma *vsubst-commute*:
assumes *X*: *good X*
and $xs \neq xs' \vee \{x,y\} \cap \{x',y'\} = \{\}$ **and** *fresh xs x X* **and** *fresh xs' x' X*
shows ((*X* #[*x* // *y*]-*xs*) #[*x'* // *y'*]-*xs'*) = ((*X* #[*x'* // *y'*]-*xs'*) #[*x* // *y*]-*xs*)
proof−
have *fresh xs' x'* (*X* #[*x* // *y*]-*xs*)
using *assms* **by** (*intro vsubst-preserves-fresh*) *auto*
moreover have *fresh xs x* (*X* #[*x'* // *y'*]-*xs'*)
using *assms* **by** (*intro vsubst-preserves-fresh*) *auto*
ultimately show *?thesis* **using** *assms*
by (*auto simp*: *vsubst-eq-swap intro*!: *swap-commute*)
qed

6.6 Abstraction versions of the properties

Environment identity and update versus other operators:

lemma *psubstAbs-idEnv*[*simp*]:
goodAbs A $\Longrightarrow$ (*A* $[*idEnv*]) = *A*

by(*simp add*: *psubstAll-idEnv*)

Substitution versus other operators:

corollary *freshAbs-psubstAbs*:
assumes *goodAbs A* **and** *goodEnv rho*
shows
freshAbs zs z (*A* $[*rho*]) =
(∀ *ys y*. *freshAbs ys y A* ∨ *freshImEnvAt zs z rho ys y*)
using *assms freshAll-psubstAll*[*of Par* [*z*] [] [] [*rho*]]
unfolding *goodPar-def* **by** *simp*

corollary *freshAbs-psubstAbs-E1*:
assumes *goodAbs A* **and** *goodEnv rho*
and *rho ys y* = *None* **and** *freshAbs zs z* (*A* $[*rho*])
shows *freshAbs ys y A* ∨ (*ys* ≠ *zs* ∨ *y* ≠ *z*)
using *assms freshAbs-psubstAbs* **unfolding** *freshImEnvAt-def* **by** *fastforce*

corollary *freshAbs-psubstAbs-E2*:
assumes *goodAbs A* **and** *goodEnv rho*
and *rho ys y* = *Some Y* **and** *freshAbs zs z* (*A* $[*rho*])
shows *freshAbs ys y A* ∨ *fresh zs z Y*
using *assms freshAbs-psubstAbs*[*of A rho*] **unfolding** *freshImEnvAt-def* **by** *fast-force*

corollary *freshAbs-psubstAbs-I1*:
assumes *goodAbs A* **and** *goodEnv rho*
and *freshAbs zs z A* **and** *freshEnv zs z rho*
shows *freshAbs zs z* (*A* $[*rho*])
using *assms* **apply**(*simp add*: *freshAbs-psubstAbs*)
unfolding *freshEnv-def liftAll-def freshImEnvAt-def* **by** *auto*

corollary *freshAbs-psubstAbs-I*:
assumes *goodAbs A* **and** *goodEnv rho*
and *rho zs z* = *None* ⟹ *freshAbs zs z A* **and**
⋀ *ys y Y*. *rho ys y* = *Some Y* ⟹ *freshAbs ys y A* ∨ *fresh zs z Y*
shows *freshAbs zs z* (*A* $[*rho*])
using *assms* **using** *option.exhaust-sel*
by (*simp add*: *freshAbs-psubstAbs freshImEnvAt-def*) *blast*

lemma *freshAbs-substAbs*:
assumes *goodAbs A* **and** *good Y*
shows *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*) =
(((*zs* = *ys* ∧ *z* = *y*) ∨ *freshAbs zs z A*) ∧ (*freshAbs ys y A* ∨ *fresh zs z Y*))
unfolding *substAbs-def* **using** *assms*
by (*auto simp*: *freshAbs-psubstAbs freshImEnvAt-def*)

lemma *freshAbs-vsubstAbs*:
assumes *goodAbs A*
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*) =

(((*zs* = *ys* ∧ *z* = *y*) ∨ *freshAbs zs z A*) ∧
(*freshAbs ys y A* ∨ (*zs* ≠ *ys* ∨ *z* ≠ *y1*)))
unfolding *vsubstAbs-def* **using** *assms* **by**(*auto simp*: *freshAbs-substAbs*)

lemma *substAbs-preserves-freshAbs*:
assumes *goodAbs A* **and** *good Y*
and *freshAbs zs z A* **and** *fresh zs z Y*
shows *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *freshAbs-substAbs*)

lemma *vsubstAbs-preserves-freshAbs*:
assumes *goodAbs A*
and *freshAbs zs z A* **and** *zs* ≠ *ys* ∨ *z* ≠ *y1*
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *freshAbs-vsubstAbs*)

lemma *fresh-freshAbs-substAbs*[*simp*]:
assumes *good Y* **and** *goodAbs A*
and *fresh ys y Y*
shows *freshAbs ys y* (*A* $[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *freshAbs-substAbs*)

lemma *diff-freshAbs-vsubstAbs*[*simp*]:
assumes *goodAbs A*
and *y* ≠ *y1*
shows *freshAbs ys y* (*A* $[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *freshAbs-vsubstAbs*)

lemma *freshAbs-substAbs-E1*:
assumes *goodAbs A* **and** *good Y*
and *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*) **and** *zs* ≠ *ys* ∨ *z* ≠ *y*
shows *freshAbs zs z A*
using *assms* **by**(*auto simp*: *freshAbs-substAbs*)

lemma *freshAbs-vsubstAbs-E1*:
assumes *goodAbs A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*) **and** *zs* ≠ *ys* ∨ *z* ≠ *y*
shows *freshAbs zs z A*
using *assms* **by**(*auto simp*: *freshAbs-vsubstAbs*)

lemma *freshAbs-substAbs-E2*:
assumes *goodAbs A* **and** *good Y*
and *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*)
shows *freshAbs ys y A* ∨ *fresh zs z Y*
using *assms* **by**(*simp add*: *freshAbs-substAbs*)

lemma *freshAbs-vsubstAbs-E2*:
assumes *goodAbs A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)

shows *freshAbs ys y A* ∨ *zs* ≠ *ys* ∨ *z* ≠ *y1*
using *assms* **by**(*simp add*: *freshAbs-vsubstAbs*)

corollary *psubstAbs-cong*[*fundef-cong*]:
assumes *goodAbs A* **and** *goodEnv rho* **and** *goodEnv rho'*
and ⋀ *ys y*. *freshAbs ys y A* ∨ *rho ys y* = *rho' ys y*
shows (*A* $[*rho*]) = (*A* $[*rho'*])
using *assms psubstAll-cong*[*of Par* [] [] [] [*rho*,*rho'*]]
unfolding *goodPar-def* **by** *simp*

lemma *freshAbs-psubstAbs-updEnv*:
assumes *good X* **and** *goodAbs A* **and** *goodEnv rho*
and *freshAbs xs x A*
shows (*A* $[*rho* [*x* ← *X*]-*xs*]) = (*A* $[*rho*])
using *assms* **by** (*intro psubstAbs-cong*) *auto*

corollary *freshEnv-psubstAbs-ident*[*simp*]:
fixes *A* :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs*
assumes *goodAbs A* **and** *goodEnv rho*
and ⋀ *zs z*. *freshEnv zs z rho* ∨ *freshAbs zs z A*
shows (*A* $[*rho*]) = *A*
using *assms psubstAll-ident*[*of Par* [] [] [] [*rho*]]
unfolding *goodPar-def* **by** *simp*

lemma *freshAbs-substAbs-ident*[*simp*]:
assumes *good X* **and** *goodAbs A* **and** *freshAbs xs x A*
shows (*A* $[*X* / *x*]-*xs*) = *A*
by (*simp add*: *assms freshAbs-psubstAbs-updEnv substAbs-def*)

corollary *substAbs-Abs*[*simp*]:
assumes *good X* **and** *good Y*
shows ((*Abs xs x X*) $[*Y* / *x*]-*xs*) = *Abs xs x X*
using *assms* **by** *simp*

lemma *freshAbs-vsubstAbs-ident*[*simp*]:
assumes *goodAbs A* **and** *freshAbs xs x A*
shows (*A* $[*x1* // *x*]-*xs*) = *A*
using *assms* **unfolding** *vsubstAbs-def* **by**(*auto simp*: *freshAbs-substAbs-ident*)

lemma *swapAbs-psubstAbs*:
assumes *goodAbs A* **and** *goodEnv rho*
shows ((*A* $[*rho*]) $[*z1* ∧ *z2*]-*zs*) = ((*A* $[*z1* ∧ *z2*]-*zs*) $[*rho* &[*z1* ∧ *z2*]-*zs*])
using *assms swapAll-psubstAll*[*of Par* [*z1*,*z2*] [] [] [*rho*]]
unfolding *goodPar-def* **by** *auto*

lemma *swapAbs-substAbs*:
assumes *goodAbs A* **and** *good Y*
shows ((*A* $[*Y* / *y*]-*ys*) $[*z1* ∧ *z2*]-*zs*) =
((*A* $[*z1* ∧ *z2*]-*zs*) $[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)

proof −
  have *1*: (*idEnv* [(*y* @*ys*[*z1* ∧ *z2*]-*zs*) ← (*Y* #[*z1* ∧ *z2*]-*zs*)]-*ys*) =
          ((*idEnv* [*y* ← *Y*]-*ys*) &[*z1* ∧ *z2*]-*zs*)
  by(*simp add*: *swapEnv-updEnv*)
  show *?thesis*
  unfolding *substAbs-def 1* **using** *assms* **by** (*intro swapAbs-psubstAbs*) *auto*
qed

lemma *swapAbs-vsubstAbs*:
assumes *goodAbs A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*z1* ∧ *z2*]-*zs*) =
       ((*A* $[*z1* ∧ *z2*]-*zs*) $[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **unfolding** *vsubstAbs-def*
by(*simp add*: *swapAbs-substAbs*)

lemma *psubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho* **and** *goodEnv rho′*
shows ((*A* $[*rho*]) $[*rho′*]) = (*A* $[(*rho* &[*rho′*])])
using *assms psubstAll-compose*[*of Par* [] [] [] [*rho*, *rho′*]]
unfolding *goodPar-def* **by** *auto*

lemma *psubstAbs-substAbs-compose*:
assumes *goodAbs A* **and** *good Y* **and** *goodEnv rho*
shows ((*A* $[*Y* / *y*]-*ys*) $[*rho*]) = (*A* $[(*rho* [*y* ← (*Y* #[*rho*])]-*ys*)])
by (*simp add*: *assms psubstAbs-compose psubstEnv-updEnv-idEnv substAbs-def*)

lemma *psubstAbs-vsubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho*
shows ((*A* $[*y1* // *y*]-*ys*) $[*rho*]) = (*A* $[(*rho* [*y* ← ((*Var ys y1*) #[*rho*])]-*ys*)])
using *assms* **unfolding** *vsubstAbs-def* **by**(*simp add*: *psubstAbs-substAbs-compose*)

lemma *substAbs-psubstAbs-compose*:
assumes *goodAbs A* **and** *good Y* **and** *goodEnv rho*
shows ((*A* $[*rho*]) $[*Y* / *y*]-*ys*) = (*A* $[(*rho* &[*Y* / *y*]-*ys*)])
unfolding *substAbs-def substEnv-def* **using** *assms* **by**(*simp add*: *psubstAbs-compose*)

lemma *psubstAbs-substAbs-compose-freshEnv*:
assumes *goodAbs A* **and** *goodEnv rho* **and** *good Y*
assumes *freshEnv ys y rho*
shows ((*A* $[*Y* / *y*]-*ys*) $[*rho*]) = ((*A* $[*rho*]) $[(*Y* #[*rho*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *substAbs-psubstAbs-compose psubstAbs-substAbs-compose*)

lemma *vsubstAbs-psubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho*
shows ((*A* $[*rho*]) $[*y1* // *y*]-*ys*) = (*A* $[(*rho* &[*y1* // *y*]-*ys*)])
unfolding *vsubstAbs-def vsubstEnv-def* **using** *assms*
by(*simp add*: *substAbs-psubstAbs-compose*)

lemma *substAbs-compose1*:

assumes *goodAbs A* **and** *good Y1* **and** *good Y2*
shows ((*A* $[*Y1* / *y*]-*ys*) $[*Y2* / *y*]-*ys*) = (*A* $[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
by (*metis assms idEnv-preserves-good psubstAbs-substAbs-compose substAbs-def subst-psubst-idEnv updEnv-overwrite updEnv-preserves-good*)

lemma *substAbs-vsubstAbs-compose1*:
assumes *goodAbs A* **and** *good Y* **and** $y \neq y1$
shows ((*A* $[*y1* // *y*]-*ys*) $[*Y* / *y*]-*ys*) = (*A* $[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubstAbs-def* **by**(*simp add*: *substAbs-compose1*)

lemma *vsubstAbs-substAbs-compose1*:
assumes *goodAbs A* **and** *good Y*
shows ((*A* $[*Y* / *y*]-*ys*) $[*y1* // *y*]-*ys*) = (*A* $[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **unfolding** *vsubstAbs-def vsubst-def* **by**(*simp add*: *substAbs-compose1*)

lemma *vsubstAbs-compose1*:
assumes *goodAbs A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*y2* // *y*]-*ys*) = (*A* $[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
using *assms* **unfolding** *vsubstAbs-def*
by(*cases y* = *y1*) (*auto simp*: *substAbs-compose1*)

lemma *substAbs-compose2*:
assumes *goodAbs A* **and** *good Y* **and** *good Z*
and $ys \neq zs \lor y \neq z$ **and** *fresh*: *fresh ys y Z*
shows ((*A* $[*Y* / *y*]-*ys*) $[*Z* / *z*]-*zs*) = ((*A* $[*Z* / *z*]-*zs*) $[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
by (*metis assms fresh freshEnv-idEnv idEnv-preserves-good psubstAbs-substAbs-compose-freshEnv substAbs-def substEnv-idEnv substEnv-preserves-freshEnv-aux subst-psubst-idEnv updEnv-preserves-good*)

lemma *substAbs-vsubstAbs-compose2*:
assumes *goodAbs A* **and** *good Z*
and $ys \neq zs \lor y \neq z$ **and** *fresh*: *fresh ys y Z*
shows ((*A* $[*y1* // *y*]-*ys*) $[*Z* / *z*]-*zs*) = ((*A* $[*Z* / *z*]-*zs*) $[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubstAbs-def* **by**(*simp add*: *substAbs-compose2*)

lemma *vsubstAbs-substAbs-compose2*:
assumes *goodAbs A* **and** *good Y*
and $ys \neq zs \lor y \notin \{z,z1\}$
shows ((*A* $[*Y* / *y*]-*ys*) $[*z1* // *z*]-*zs*) = ((*A* $[*z1* // *z*]-*zs*) $[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
using *assms* **unfolding** *vsubstAbs-def vsubst-def* **by**(*simp add*: *substAbs-compose2*)

lemma *vsubstAbs-compose2*:
assumes *goodAbs A*
and $ys \neq zs \lor y \notin \{z,z1\}$
shows ((*A* $[*y1* // *y*]-*ys*) $[*z1* // *z*]-*zs*) =
((*A* $[*z1* // *z*]-*zs*) $[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)

unfolding *vsubstAbs-def*
by (*smt* (*verit*) *Var-preserves-good assms fresh-Var-simp insertCI substAbs-compose2 vsubst-Var-simp vsubst-def*)

Properties specific to variable-for-variable substitution:

corollary *vsubstAbs-ident*[*simp*]:
assumes *goodAbs A*
shows (*A* $[*z* // *z*]-*zs*) = *A*
using *assms vsubstAll-ident*[*of Par* [*z*] [] [] [] - - *A*]
unfolding *goodPar-def* **by** *simp*

corollary *substAbs-ident*[*simp*]:
assumes *goodAbs A*
shows (*A* $[(*Var zs z*) / *z*]-*zs*) = *A*
using *assms vsubstAbs-ident* **unfolding** *vsubstAbs-def* **by** *auto*

corollary *vsubstAbs-eq-swapAbs*:
assumes *goodAbs A* **and** *freshAbs ys y1 A*
shows (*A* $[*y1* // *y2*]-*ys*) = (*A* $[*y1* ∧ *y2*]-*ys*)
using *assms vsubstAll-swapAll*[*of Par* [*y1*, *y2*] [] [] [] - - *A*]
unfolding *goodPar-def* **by** *simp*

corollary *skelAbs-vsubstAbs*:
assumes *goodAbs A*
shows *skelAbs* (*A* $[*y1* // *y2*]-*ys*) = *skelAbs A*
using *assms skelAll-vsubstAll*[*of Par* [*y1*, *y2*] [] [] [] - - *A*]
unfolding *goodPar-def* **by** *simp*

lemma *substAbs-vsubstAbs-trans*:
assumes *goodAbs A* **and** *good Y* **and** *freshAbs ys y1 A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*Y* / *y1*]-*ys*) = (*A* $[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substAbs-def vsubstAbs-def*
by (*cases y1* = *y*) (*auto simp*: *freshAbs-psubstAbs-updEnv psubstAbs-compose psubstEnv-updEnv-idEnv updEnv-commute*)

lemma *vsubstAbs-trans*:
assumes *goodAbs A* **and** *freshAbs ys y1 A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*y2* // *y1*]-*ys*) = (*A* $[*y2* // *y*]-*ys*)
unfolding *vsubstAbs-def*[*of - y2 y1*] *vsubstAbs-def*[*of - y2 y*]
using *assms* **by**(*simp add*: *substAbs-vsubstAbs-trans*)

lemmas *good-psubstAll-freshAll-otherSimps* =
psubst-idEnv psubstEnv-idEnv-id psubstAbs-idEnv
freshEnv-psubst-ident freshEnv-psubstAbs-ident

lemmas *good-substAll-freshAll-otherSimps* =
fresh-fresh-subst fresh-subst-ident fresh-substEnv-updEnv subst-ident
fresh-freshAbs-substAbs freshAbs-substAbs-ident substAbs-ident

lemmas *good-vsubstAll-freshAll-otherSimps* =
diff-fresh-vsubst fresh-vsubst-ident fresh-vsubstEnv-updEnv vsubst-ident
diff-freshAbs-vsubstAbs freshAbs-vsubstAbs-ident vsubstAbs-ident

lemmas *good-allOpers-otherSimps* =
good-swapAll-freshAll-otherSimps
good-psubstAll-freshAll-otherSimps
good-substAll-freshAll-otherSimps
good-vsubstAll-freshAll-otherSimps

lemmas *good-item-simps* =
param-simps
all-preserve-good
good-freeCons
good-allOpers-simps
good-allOpers-otherSimps

end

end

7 Binding Signatures and well-sorted terms

theory *Well-Sorted-Terms*
imports *Terms*
begin

This section introduces binding signatures and well-sorted terms for them. All the properties we proved for good terms are then lifted to well-sorted terms.

7.1 Binding signatures

A *(binding) signature* consists of:
- an indication of which sorts of variables can be injected in which sorts of terms;
- for any operation symbol, dwelling in a type "opSym", an indication of its result sort, its (nonbinding) arity, and its binding arity.

In addition, we have a predicate, "wlsOpSym", that specifies which operations symbols are well-sorted (or well-structured) [1] – only these operation symbols will be considered in forming terms. In other words, the relevant collection of operation symbols is given not by the whole type "opSym", but by the predicate "wlsOpSym". This bit of extra flexibility will be useful when (pre)instantiating the signature to concrete syntaxes. (Note that the

[1]We shall use "wls" in many contexts as a prefix indicating well-sortedness or well-structuredness.

"wlsOpSym" condition will be required for well-sorted terms as part of the notion of well-sorted (free and bound) input, "wlsInp" and "wlsBinp".)

record *('index,'bindex,'varSort,'sort,'opSym)signature* =
varSortAsSort :: *'varSort* ⇒ *'sort*
wlsOpSym :: *'opSym* ⇒ *bool*
sortOf :: *'opSym* ⇒ *'sort*
arityOf :: *'opSym* ⇒ (*'index*, *'sort*)*input*
barityOf :: *'opSym* ⇒ (*'bindex*, *'varSort* ∗ *'sort*)*input*

7.2 The Binding Syntax locale

For our signatures, we shall make some assumptions:
- For each sort of term, there is at most one sort of variables injectable in terms of that sort (i.e., "varSortAsSort" is injective");
- The domains of arities (sets of indexes) are smaller than the set of variables of each sort;
- The type of sorts is smaller than the set of variables of each sort.

These are satisfiable assumptions, and in particular they are trivially satisfied by any finitary syntax with bindings.

definition *varSortAsSort-inj* **where**
varSortAsSort-inj Delta ==
inj (*varSortAsSort Delta*)

definition *arityOf-lt-var* **where**
arityOf-lt-var (- :: *'var*) *Delta* ==
∀ *delta*.
wlsOpSym Delta delta ⟶ |{*i*. *arityOf Delta delta i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|

definition *barityOf-lt-var* **where**
barityOf-lt-var (- :: *'var*) *Delta* ==
∀ *delta*.
wlsOpSym Delta delta ⟶ |{*i*. *barityOf Delta delta i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|

definition *sort-lt-var* **where**
sort-lt-var (- :: *'sort*) (- :: *'var*) ==
|*UNIV* :: *'sort set*| <*o* |*UNIV* :: *'var set*|

locale *FixSyn* =
fixes *dummyV* :: *'var*
and *Delta* :: (*'index,'bindex,'varSort,'sort,'opSym*)*signature*
assumes

FixSyn-var-infinite: *var-infinite* (*undefined* :: *'var*)
and *FixSyn-var-regular*: *var-regular* (*undefined* :: *'var*)

and *varSortAsSort-inj*: *varSortAsSort-inj Delta*
and *arityOf-lt-var*: *arityOf-lt-var* (*undefined* :: *'var*) *Delta*
and *barityOf-lt-var*: *barityOf-lt-var* (*undefined* :: *'var*) *Delta*
and *sort-lt-var*: *sort-lt-var* (*undefined* :: *'sort*) (*undefined* :: *'var*)

context *FixSyn*
begin
lemmas *FixSyn-assms* =
FixSyn-var-infinite FixSyn-var-regular
varSortAsSort-inj arityOf-lt-var barityOf-lt-var
sort-lt-var
end

7.3 Definitions and basic properties of well-sortedness

7.3.1 Notations and definitions

datatype (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* =
ParS 'varSort ⇒ *'var list*
'sort ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term list*
(*'varSort* * *'sort*) ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs list*
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env list*

fun *varsOfS* ::
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* ⇒ *'varSort* ⇒ *'var set*
where *varsOfS* (*ParS xLF* - - -) *xs* = *set* (*xLF xs*)

fun *termsOfS* ::
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* ⇒
'sort ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term set*
where *termsOfS* (*ParS* - *XLF* - -) *s* = *set* (*XLF s*)

fun *absOfS* ::
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* ⇒
(*'varSort* * *'sort*) ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs set*
where *absOfS* (*ParS* - - *ALF* -) (*xs*,*s*) = *set* (*ALF* (*xs*,*s*))

fun *envsOfS* ::
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env set*
where *envsOfS* (*ParS* - - - *rhoL*) = *set rhoL*

7.3.2 Sublocale of "FixVars"

lemma *sort-lt-var-imp-varSort-lt-var*:
assumes
**: *varSortAsSort-inj* (*Delta* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*)*signature*)
and ***: *sort-lt-var* (*undefined* :: *'sort*) (*undefined* :: *'var*)
shows *varSort-lt-var* (*undefined* :: *'varSort*) (*undefined* :: *'var*)
proof−

have |*UNIV*::*'varSort set*| ≤*o* |*UNIV*::*'sort set*|
using *card-of-ordLeq* ∗∗ **unfolding** *varSortAsSort-inj-def* **by** *auto*
thus *?thesis*
using *ordLeq-ordLess-trans assms*
unfolding *sort-lt-var-def varSort-lt-var-def* **by** *blast*
qed

sublocale *FixSyn* < *FixVars*
where *dummyV* = *dummyV* **and** *dummyVS* = *undefined*::*'varSort*
using *FixSyn-assms*
by *unfold-locales* (*auto simp add*: *sort-lt-var-imp-varSort-lt-var*)

7.3.3 Abbreviations

context *FixSyn*
begin

abbreviation *asSort* **where** *asSort* == *varSortAsSort Delta*
abbreviation *wlsOpS* **where** *wlsOpS* == *wlsOpSym Delta*
abbreviation *stOf* **where** *stOf* == *sortOf Delta*
abbreviation *arOf* **where** *arOf* == *arityOf Delta*
abbreviation *barOf* **where** *barOf* == *barityOf Delta*

abbreviation *empInp* ::
(*'index*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term*)*input*
where *empInp* == λ*i*. *None*

abbreviation *empAr* :: (*'index*,*'sort*)*input*
where *empAr* == λ*i*. *None*

abbreviation *empBinp* :: (*'bindex*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs*)*input*
where *empBinp* == λ*i*. *None*

abbreviation *empBar* :: (*'bindex*,*'varSort* ∗ *'sort*)*input*
where *empBar* == λ*i*. *None*

lemma *freshInp-empInp*[*simp*]:
freshInp xs x empInp
unfolding *freshInp-def liftAll-def* **by** *simp*

lemma *swapInp-empInp*[*simp*]:
(*empInp* %[*x1* ∧ *x2*]-*xs*) = *empInp*
unfolding *swapInp-def lift-def* **by** *simp*

lemma *psubstInp-empInp*[*simp*]:
(*empInp* %[*rho*]) = *empInp*
unfolding *psubstInp-def lift-def* **by** *simp*

lemma *substInp-empInp*[*simp*]:

(*empInp* %[*Y* / *y*]-*ys*) = *empInp*
unfolding *substInp-def* **by** *simp*

lemma *vsubstInp-empInp*[*simp*]:
(*empInp* %[*y1* // *y*]-*ys*) = *empInp*
unfolding *vsubstInp-def* **by** *simp*

lemma *freshBinp-empBinp*[*simp*]:
freshBinp xs x empBinp
unfolding *freshBinp-def liftAll-def* **by** *simp*

lemma *swapBinp-empBinp*[*simp*]:
(*empBinp* %%[*x1* ∧ *x2*]-*xs*) = *empBinp*
unfolding *swapBinp-def lift-def* **by** *simp*

lemma *psubstBinp-empBinp*[*simp*]:
(*empBinp* %%[*rho*]) = *empBinp*
unfolding *psubstBinp-def lift-def* **by** *simp*

lemma *substBinp-empBinp*[*simp*]:
(*empBinp* %%[*Y* / *y*]-*ys*) = *empBinp*
unfolding *substBinp-def* **by** *simp*

lemma *vsubstBinp-empBinp*[*simp*]:
(*empBinp* %%[*y1* // *y*]-*ys*) = *empBinp*
unfolding *vsubstBinp-def* **by** *simp*

lemmas *empInp-simps* =
freshInp-empInp swapInp-empInp psubstInp-empInp substInp-empInp vsubstInp-empInp freshBinp-empBinp swapBinp-empBinp psubstBinp-empBinp substBinp-empBinp vsubstBinp-empBinp

7.3.4 Inner versions of the locale assumptions

lemma *varSortAsSort-inj-INNER*: *inj asSort*
using *varSortAsSort-inj*
unfolding *varSortAsSort-inj-def* **by** *simp*

lemma *asSort-inj*[*simp*]:
(*asSort xs* = *asSort ys*) = (*xs* = *ys*)
using *varSortAsSort-inj-INNER* **unfolding** *inj-on-def* **by** *auto*

lemma *arityOf-lt-var-INNER*:
assumes *wlsOpS delta*
shows |{*i*. *arityOf Delta delta i* ≠ *None*}| <*o* |*UNIV* :: *'var set*|
using *assms arityOf-lt-var* **unfolding** *arityOf-lt-var-def* **by** *simp*

lemma *barityOf-lt-var-INNER*:
assumes *wlsOpS delta*

shows $|\{i.\ barityOf\ Delta\ delta\ i \neq None\}|$ $<o$ $|UNIV :: 'var\ set|$
using *assms barityOf-lt-var* **unfolding** *barityOf-lt-var-def* **by** *simp*

lemma *sort-lt-var-INNER*:
$|UNIV :: 'sort\ set|$ $<o$ $|UNIV :: 'var\ set|$
using *sort-lt-var* **unfolding** *sort-lt-var-def* **by** *simp*

lemma *sort-le-var*:
$|UNIV :: 'sort\ set| \leq o\ |UNIV :: 'var\ set|$
using *sort-lt-var-INNER ordLess-imp-ordLeq* **by** *auto*

lemma *varSort-sort-lt-var*:
$|UNIV :: ('varSort * 'sort)\ set|$ $<o$ $|UNIV :: 'var\ set|$
unfolding *UNIV-Times-UNIV*[*symmetric*]
using *var-infinite-INNER varSort-lt-var-INNER sort-lt-var-INNER*
by(*rule card-of-Times-ordLess-infinite*)

lemma *varSort-sort-le-var*:
$|UNIV :: ('varSort * 'sort)\ set| \leq o\ |UNIV :: 'var\ set|$
using *varSort-sort-lt-var ordLess-imp-ordLeq* **by** *auto*

7.3.5 Definitions of well-sorted items

We shall only be interested in abstractions that pertain to some bound arities:

definition *isInBar* **where**
isInBar xs-s ==
$\exists$ *delta i. wlsOpS delta* $\wedge$ *barOf delta i* = *Some xs-s*

Well-sorted terms (according to the signature) are defined as expected (mutually inductively together with well-sorted abstractions and inputs):

inductive
wls :: *'sort* ⇒ (*'index,'bindex,'varSort,'var,'opSym*)*term* ⇒ *bool*
and
wlsAbs :: *'varSort* ∗ *'sort* ⇒ (*'index,'bindex,'varSort,'var,'opSym*)*abs* ⇒ *bool*
and
wlsInp :: *'opSym* ⇒ (*'index*,(*'index,'bindex,'varSort,'var,'opSym*)*term*)*input* ⇒ *bool*
and
wlsBinp :: *'opSym* ⇒ (*'bindex*,(*'index,'bindex,'varSort,'var,'opSym*)*abs*)*input* ⇒ *bool*
where
Var: *wls* (*asSort xs*) (*Var xs x*)
|
Op: ⟦*wlsInp delta inp*; *wlsBinp delta binp*⟧ ⟹ *wls* (*stOf delta*) (*Op delta inp binp*)
|
Inp:
⟦*wlsOpS delta*;
$\bigwedge$ *i*. (*arOf delta i* = *None* $\wedge$ *inp i* = *None*) $\vee$

(∃ *s X*. *arOf delta i* = *Some s* ∧ *inp i* = *Some X* ∧ *wls s X*)⟧
⟹ *wlsInp delta inp*
|
Binp:
⟦*wlsOpS delta*;
⋀ *i*. (*barOf delta i* = *None* ∧ *binp i* = *None*) ∨
(∃ *us s A*. *barOf delta i* = *Some* (*us*,*s*) ∧ *binp i* = *Some A* ∧ *wlsAbs* (*us*,*s*)
A)⟧
⟹ *wlsBinp delta binp*
|
Abs: ⟦*isInBar* (*xs*,*s*); *wls s X*⟧ ⟹ *wlsAbs* (*xs*,*s*) (*Abs xs x X*)

lemmas *Var-preserves-wls* = *wls-wlsAbs-wlsInp-wlsBinp.Var*
lemmas *Op-preserves-wls* = *wls-wlsAbs-wlsInp-wlsBinp.Op*
lemmas *Abs-preserves-wls* = *wls-wlsAbs-wlsInp-wlsBinp.Abs*

lemma *barOf-isInBar*[*simp*]:
assumes *wlsOpS delta* **and** *barOf delta i* = *Some* (*us*,*s*)
shows *isInBar* (*us*,*s*)
unfolding *isInBar-def* **using** *assms* **by** *blast*

lemmas *Cons-preserve-wls* =
barOf-isInBar
Var-preserves-wls Op-preserves-wls
Abs-preserves-wls

declare *Cons-preserve-wls* [*simp*]

definition *wlsEnv* :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env* ⇒ *bool*
where
wlsEnv rho ==
(∀ *ys*. *liftAll* (*wls* (*asSort ys*)) (*rho ys*)) ∧
(∀ *ys*. |{*y*. *rho ys y* ≠ *None*}| <*o* |*UNIV* :: ′*var set*|)

definition *wlsPar* :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*,′*sort*)*paramS* ⇒ *bool*
where
wlsPar P ==
(∀ *s*. ∀ *X* ∈ *termsOfS P s*. *wls s X*) ∧
(∀ *xs s*. ∀ *A* ∈ *absOfS P* (*xs*,*s*). *wlsAbs* (*xs*,*s*) *A*) ∧
(∀ *rho* ∈ *envsOfS P*. *wlsEnv rho*)

lemma *ParS-preserves-wls*[*simp*]:
assumes ⋀ *s X*. *X* ∈ *set* (*XLF s*) ⟹ *wls s X*
and ⋀ *xs s A*. *A* ∈ *set* (*ALF* (*xs*,*s*)) ⟹ *wlsAbs* (*xs*,*s*) *A*
and ⋀ *rho*. *rho* ∈ *set rhoF* ⟹ *wlsEnv rho*
shows *wlsPar* (*ParS xLF XLF ALF rhoF*)
using *assms* **unfolding** *wlsPar-def* **by** *auto*

lemma *termsOfS-preserves-wls*[*simp*]:

assumes *wlsPar P* **and** *X* : *termsOfS P s*
shows *wls s X*
using *assms* **unfolding** *wlsPar-def* **by** *auto*

lemma *absOfS-preserves-wls*[*simp*]:
assumes *wlsPar P* **and** *A* : *absOfS P* (*us*,*s*)
shows *wlsAbs* (*us*,*s*) *A*
using *assms* **unfolding** *wlsPar-def* **by** *auto*

lemma *envsOfS-preserves-wls*[*simp*]:
assumes *wlsPar P* **and** *rho* : *envsOfS P*
shows *wlsEnv rho*
using *assms* **unfolding** *wlsPar-def* **by** *blast*

lemma *not-isInBar-absOfS-empty*[*simp*]:
assumes ∗: ¬ *isInBar* (*us*,*s*) **and** ∗∗: *wlsPar P*
shows *absOfS P* (*us*,*s*) = {}
proof−
  {**fix** *A* **assume** *A* : *absOfS P* (*us*,*s*)
   hence *wlsAbs* (*us*,*s*) *A* **using** ∗∗ **by** *simp*
   hence *False* **using** ∗ **using** *wlsAbs.cases* **by** *auto*
  }
  thus *?thesis* **by** *auto*
qed

lemmas *paramS-simps* =
varsOfS.simps termsOfS.simps absOfS.simps envsOfS.simps
ParS-preserves-wls
termsOfS-preserves-wls absOfS-preserves-wls envsOfS-preserves-wls
not-isInBar-absOfS-empty

7.3.6 Well-sorted exists

lemma *wlsInp-iff*:
wlsInp delta inp =
(*wlsOpS delta* ∧ *sameDom* (*arOf delta*) *inp* ∧ *liftAll2 wls* (*arOf delta*) *inp*)
by (*simp add*: *wlsInp.simps wls-wlsAbs-wlsInp-wlsBinp.Inp sameDom-and-liftAll2-iff*)

lemma *wlsBinp-iff*:
wlsBinp delta binp =
(*wlsOpS delta* ∧ *sameDom* (*barOf delta*) *binp* ∧ *liftAll2 wlsAbs* (*barOf delta*) *binp*)
by (*simp add*: *wlsBinp.simps wls-wlsAbs-wlsInp-wlsBinp.Inp sameDom-and-liftAll2-iff*)

lemma *exists-asSort-wls*:
∃ *X*. *wls* (*asSort xs*) *X*
by (*intro exI*[*of - Var xs undefined*]) *simp*

lemma *exists-wls-imp-exists-wlsAbs*:
assumes $*$: *isInBar* (*us*,*s*) **and** $**$: $\exists$ *X*. *wls s X*
shows $\exists$ *A*. *wlsAbs* (*us*,*s*) *A*
proof−
  obtain *X* **where** *wls s X* **using** $**$ **by** *blast*
  hence *wlsAbs* (*us*,*s*) (*Abs us undefined X*) **using** $*$ **by** *simp*
  thus *?thesis* **by** *blast*
qed

lemma *exists-asSort-wlsAbs*:
assumes *isInBar* (*us*,*asSort xs*)
shows $\exists$ *A*. *wlsAbs* (*us*,*asSort xs*) *A*
proof−
  obtain *X* **where** *wls* (*asSort xs*) *X* **using** *exists-asSort-wls* **by** *auto*
  thus *?thesis* **using** *assms exists-wls-imp-exists-wlsAbs* **by** *auto*
qed

Standard criterion for the non-emptiness of the sets of well-sorted terms for each sort, by a well-founded relation and a function picking, for sorts not corresponding to varSorts, an operation symbol as an "inductive" witness for non-emptyness. "witOpS" stands for "witness operation symbol".

definition *witOpS* **where**
witOpS s delta R ==
 wlsOpS delta $\wedge$ *stOf delta* = *s* $\wedge$
 liftAll ($\lambda s'$. (s',*s*) : *R*) (*arOf delta*) $\wedge$
 liftAll (λ(*us*,s'). (s',*s*) : *R*) (*barOf delta*)

lemma *wf-exists-wls*:
assumes *wf*: *wf R* **and** $*$: $\bigwedge s$. ($\exists$ *xs*. *s* = *asSort xs*) $\vee$ *witOpS s* (*f s*) *R*
shows $\exists$ *X*. *wls s X*
proof(*induction rule*: *wf-induct*[*of R*])
  case (*2 s*)
  show *?case*
  proof(*cases* $\exists$ *xs*. *s* = *asSort xs*)
    case *True*
    thus *?thesis* **using** *exists-asSort-wls* **by** *auto*
  next
    let *?delta* = *f s*
    case *False*
    hence *delta*: *wlsOpS ?delta* **and** *st*: *stOf ?delta* = *s*
    and *ar*: *liftAll* ($\lambda s'$. (s',*s*) : *R*) (*arOf ?delta*)
    and *bar*: *liftAll* (λ(*us*,s'). (s',*s*) : *R*) (*barOf ?delta*)
    using $*$ **unfolding** *witOpS-def* **by** *auto*

    have *1*: $\forall$ *i* s'. *arOf ?delta i* = *Some* s' $\longrightarrow$ ($\exists$ *X*. *wls* s' *X*)
    using *ar 2* **unfolding** *liftAll-def* **by** *simp*
    let *?chi* = λi s' *X*. *arOf ?delta i* = *Some* s' $\longrightarrow$ *wls* s' *X*
    define *inp* **where**

inp ≡ (λ*i*. (*if arOf ?delta i* = *None*
then None
else Some (*SOME X*. ∀ *s*′. *?chi i s*′ *X*)))
have *inp*: *wlsInp ?delta inp*
unfolding *wlsInp-iff sameDom-def liftAll2-def* **using** *delta*
by (*auto simp*: *inp-def 1 someI2-ex split*: *if-splits*)

have *1*: ∀ *i us s*′. *barOf ?delta i* = *Some* (*us*,*s*′) ⟶ (∃ *A*. *wlsAbs* (*us*,*s*′) *A*)
using *bar 2* **unfolding** *liftAll-def* **using** *delta exists-wls-imp-exists-wlsAbs* **by** *simp*
let *?chi* = λ*i us s*′ *A*. *barOf ?delta i* = *Some* (*us*,*s*′) ⟶ *wlsAbs* (*us*,*s*′) *A*
define *binp* **where**
binp ≡ (λ*i*. (*if barOf ?delta i* = *None*
then None
else Some (*SOME A*. ∀ *us s*′. *?chi i us s*′ *A*)))
have *binp*: *wlsBinp ?delta binp*
unfolding *wlsBinp-iff sameDom-def liftAll2-def* **using** *delta*
by (*auto simp*: *binp-def 1 someI2-ex split*: *if-splits*)

have *wls s* (*Op ?delta inp binp*)
using *inp binp st* **using** *Op-preserves-wls*[*of ?delta inp binp*] **by** *simp*
thus *?thesis* **by** *blast*
qed
qed(*insert assms*, *auto*)

lemma *wf-exists-wlsAbs*:
assumes *isInBar* (*us*,*s*)
and *wf R* **and** ⋀*s*. (∃ *xs*. *s* = *asSort xs*) ∨ *witOpS s* (*f s*) *R*
shows ∃ *A*. *wlsAbs* (*us*,*s*) *A*
using *assms* **by** (*auto intro*: *exists-wls-imp-exists-wlsAbs wf-exists-wls*)

7.3.7 Well-sorted implies Good

lemma *wlsInp-empAr-empInp*[*simp*]:
assumes *wlsOpS delta* **and** *arOf delta* = *empAr*
shows *wlsInp delta empInp*
using *assms*
unfolding *wlsInp-iff sameDom-def liftAll2-def* **by** *auto*

lemma *wlsBinp-empBar-empBinp*[*simp*]:
assumes *wlsOpS delta* **and** *barOf delta* = *empBar*
shows *wlsBinp delta empBinp*
using *assms* **unfolding** *wlsBinp-iff sameDom-def liftAll2-def* **by** *auto*

lemmas *empInp-otherSimps* =
wlsInp-empAr-empInp wlsBinp-empBar-empBinp

lemma *wlsAll-implies-goodAll*:
(*wls s X* ⟶ *good X*) ∧

(*wlsAbs* (*xs*,*s'*) *A* ⟶ *goodAbs A*) ∧
(*wlsInp delta inp* ⟶ *goodInp inp*) ∧
(*wlsBinp delta binp* ⟶ *goodBinp binp*)
apply(*induct rule*: *wls-wlsAbs-wlsInp-wlsBinp.induct*)
subgoal by *auto*
subgoal by *auto*
subgoal unfolding *goodInp-def liftAll-def*
by *simp* (*smt* (*verit*) *Collect-cong arityOf-lt-var-INNER option.distinct*(*1*) *option.sel*)
subgoal unfolding *goodBinp-def liftAll-def*
by *simp* (*smt* (*verit*) *Collect-cong barityOf-lt-var-INNER option.distinct*(*1*) *option.sel*)
subgoal by *auto* **.**

corollary *wls-imp-good*[*simp*]: *wls s X* ⟹ *good X*
by(*simp add*: *wlsAll-implies-goodAll*)

corollary *wlsAbs-imp-goodAbs*[*simp*]: *wlsAbs* (*xs*,*s*) *A* ⟹ *goodAbs A*
by(*simp add*: *wlsAll-implies-goodAll*)

corollary *wlsInp-imp-goodInp*[*simp*]: *wlsInp delta inp* ⟹ *goodInp inp*
by(*simp add*: *wlsAll-implies-goodAll*)

corollary *wlsBinp-imp-goodBinp*[*simp*]: *wlsBinp delta binp* ⟹ *goodBinp binp*
by(*simp add*: *wlsAll-implies-goodAll*)

lemma *wlsEnv-imp-goodEnv*[*simp*]: *wlsEnv rho* ⟹ *goodEnv rho*
unfolding *wlsEnv-def goodEnv-def liftAll-def*
by *simp* (*insert wls-imp-good*, *blast*)

lemmas *wlsAll-imp-goodAll* =
wls-imp-good wlsAbs-imp-goodAbs
wlsInp-imp-goodInp wlsBinp-imp-goodBinp
wlsEnv-imp-goodEnv

7.3.8 Swapping preserves well-sortedness

lemma *swapAll-pres-wlsAll*:
(*wls s X* ⟶ *wls s* (*X* #[*z1* ∧ *z2*]-*zs*)) ∧
(*wlsAbs* (*xs*,*s'*) *A* ⟶ *wlsAbs* (*xs*,*s'*) (*A* $[*z1* ∧ *z2*]-*zs*)) ∧
(*wlsInp delta inp* ⟶ *wlsInp delta* (*inp* %[*z1* ∧ *z2*]-*zs*)) ∧
(*wlsBinp delta binp* ⟶ *wlsBinp delta* (*binp* %%[*z1* ∧ *z2*]-*zs*))
proof(*induct rule*: *wls-wlsAbs-wlsInp-wlsBinp.induct*)
  case (*Inp delta inp*)
  then show *?case*
  unfolding *wlsInp-iff sameDom-def liftAll2-def lift-def swapInp-def*
  using *option.sel* **by** (*fastforce simp add*: *split*: *option.splits*)
next
  case (*Binp delta binp*)
  then show *?case*

unfolding *wlsBinp-iff sameDom-def liftAll2-def lift-def swapBinp-def*
using *option.sel* **by** (*fastforce simp add*: *split*: *option.splits*)
qed(*insert Cons-preserve-wls*, *simp-all*)

lemma *swap-preserves-wls*[*simp*]:
wls s X $\Longrightarrow$ *wls s* (*X* #[*z1* $\wedge$ *z2*]-*zs*)
by(*simp add*: *swapAll-pres-wlsAll*)

lemma *swap-preserves-wls2*[*simp*]:
assumes *good X*
shows *wls s* (*X* #[*z1* $\wedge$ *z2*]-*zs*) = *wls s X*
using *assms swap-preserves-wls*[*of s X* #[*z1* $\wedge$ *z2*]-*zs zs z1 z2*] **by** *auto*

lemma *swap-preserves-wls3*:
assumes *good X* **and** *good Y*
and (*X* #[*x1* $\wedge$ *x2*]-*xs*) = (*Y* #[*y1* $\wedge$ *y2*]-*ys*)
shows *wls s X* = *wls s Y*
by (*metis assms swap-preserves-wls2*)

lemma *swapAbs-preserves-wls*[*simp*]:
wlsAbs (*xs*,*x*) *A* $\Longrightarrow$ *wlsAbs* (*xs*,*x*) (*A* $[*z1* $\wedge$ *z2*]-*zs*)
by(*simp add*: *swapAll-pres-wlsAll*)

lemma *swapInp-preserves-wls*[*simp*]:
wlsInp delta inp $\Longrightarrow$ *wlsInp delta* (*inp* %[*z1* $\wedge$ *z2*]-*zs*)
by(*simp add*: *swapAll-pres-wlsAll*)

lemma *swapBinp-preserves-wls*[*simp*]:
wlsBinp delta binp $\Longrightarrow$ *wlsBinp delta* (*binp* %%[*z1* $\wedge$ *z2*]-*zs*)
by(*simp add*: *swapAll-pres-wlsAll*)

lemma *swapEnvDom-preserves-wls*:
assumes *wlsEnv rho*
shows *wlsEnv* (*swapEnvDom xs x y rho*)
proof−
 {**fix** *xsa ys* **let** *?Left* = {*ya*. *swapEnvDom xs x y rho ys ya* $\neq$ *None*}
  have |{*y*} $\cup$ {*ya*. *rho ys ya* $\neq$ *None*}| <*o* |*UNIV* :: ′*var set*|
  using *assms var-infinite-INNER card-of-Un-singl-ordLess-infinite*
  unfolding *wlsEnv-def* **by** *fastforce*
  hence |{*x*,*y*} $\cup$ {*ya*. *rho ys ya* $\neq$ *None*}| <*o* |*UNIV* :: ′*var set*|
  using *var-infinite-INNER card-of-Un-singl-ordLess-infinite* **by** *fastforce*
  moreover
  {**have** *?Left* $\subseteq$ {*x*,*y*} $\cup$ {*ya*. *rho ys ya* $\neq$ *None*}
   unfolding *swapEnvDom-def sw-def*[*abs-def*] **by** *auto*
   hence |*?Left*| $\leq$*o* |{*x*,*y*} $\cup$ {*ya*. *rho ys ya* $\neq$ *None*}|
   using *card-of-mono1* **by** *auto*
  }
  ultimately have |*?Left*| <*o* |*UNIV* :: ′*var set*|
  using *ordLeq-ordLess-trans* **by** *blast*

}
thus *?thesis* **using** *assms* **unfolding** *wlsEnv-def liftAll-def*
by (*auto simp add*: *swapEnvDom-def*)
qed

lemma *swapEnvIm-preserves-wls*:
assumes *wlsEnv rho*
shows *wlsEnv* (*swapEnvIm xs x y rho*)
using *assms* **unfolding** *wlsEnv-def swapEnvIm-def liftAll-def lift-def*
by (*auto split*: *option.splits*)

lemma *swapEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho*
shows *wlsEnv* (*rho* &[*z1* ∧ *z2*]-*zs*)
unfolding *swapEnv-def comp-def*
using *assms* **by**(*auto simp*: *swapEnvDom-preserves-wls swapEnvIm-preserves-wls*)

lemmas *swapAll-preserve-wls* =
swap-preserves-wls swapAbs-preserves-wls
swapInp-preserves-wls swapBinp-preserves-wls
swapEnv-preserves-wls

lemma *swapped-preserves-wls*:
assumes *wls s X* **and** $(X,Y) \in$ *swapped*
shows *wls s Y*
proof−
have $(X,Y) \in$ *swapped* ⟹ *wls s X* ⟶ *wls s Y*
by (*induct rule*: *swapped.induct*) *auto*
thus *?thesis* **using** *assms* **by** *simp*
qed

7.3.9 Inversion rules for well-sortedness

lemma *wlsAll-inversion*:
(*wls s X* ⟶
(∀ *xs x*. *X* = *Var xs x* ⟶ *s* = *asSort xs*) ∧
(∀ *delta inp binp*. *goodInp inp* ∧ *goodBinp binp* ∧ *X* = *Op delta inp binp* ⟶
stOf delta = *s* ∧ *wlsInp delta inp* ∧ *wlsBinp delta binp*))
∧
(*wlsAbs xs-s A* ⟶
isInBar xs-s ∧
(∀ *x X*. *good X* ∧ *A* = *Abs* (*fst xs-s*) *x X* ⟶
wls (*snd xs-s*) *X*))
∧
(*wlsInp delta inp* ⟶ *True*)
∧
(*wlsBinp delta binp* ⟶ *True*)
proof(*induct rule*: *wls-wlsAbs-wlsInp-wlsBinp.induct*)
case (*Abs xs s X x*)

then show *?case* **using** *swap-preserves-wls3 wls-imp-good*
by (*metis FixVars.Abs-ainj-ex FixVars-axioms snd-conv*)
qed (*auto simp*: *Abs-ainj-ex*)

lemma *conjLeft*: ⟦*phi1* ∧ *phi2*; *phi1* ⟹ *chi*⟧ ⟹ *chi*
by *blast*

lemma *conjRight*: ⟦*phi1* ∧ *phi2*; *phi2* ⟹ *chi*⟧ ⟹ *chi*
by *blast*

lemma *wls-inversion*[*rule-format*]:
wls s X ⟶
(∀ *xs x*. *X* = *Var xs x* ⟶ *s* = *asSort xs*) ∧
(∀ *delta inp binp*. *goodInp inp* ∧ *goodBinp binp* ∧ *X* = *Op delta inp binp* ⟶
stOf delta = *s* ∧ *wlsInp delta inp* ∧ *wlsBinp delta binp*)
using *wlsAll-inversion*
[*of s X undefined undefined undefined undefined undefined*]
by (*rule conjLeft*)

lemma *wlsAbs-inversion*[*rule-format*]:
wlsAbs (*xs*,*s*) *A* ⟶
isInBar (*xs*,*s*) ∧
(∀ *x X*. *good X* ∧ *A* = *Abs xs x X* ⟶ *wls s X*)
using *wlsAll-inversion*
[*of undefined undefined* (*xs*,*s*) *A undefined undefined undefined*]
by *auto*

lemma *wls-Var-simp*[*simp*]:
wls s (*Var xs x*) = (*s* = *asSort xs*)
using *wls-inversion* **by** *auto*

lemma *wls-Op-simp*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows
wls s (*Op delta inp binp*) =
(*stOf delta* = *s* ∧ *wlsInp delta inp* ∧ *wlsBinp delta binp*)
using *Op assms wls-inversion* **by** *blast*

lemma *wls-Abs-simp*[*simp*]:
assumes *good X*
shows *wlsAbs* (*xs*,*s*) (*Abs xs x X*) = (*isInBar* (*xs*,*s*) ∧ *wls s X*)
using *Abs assms wlsAbs-inversion* **by** *blast*

lemma *wlsAll-inversion2*:
(*wls s X* ⟶ *True*)
∧
(*wlsAbs xs-s A* ⟶

isInBar xs-s ∧
(∃ *x X*. *wls* (*snd xs-s*) *X* ∧ *A* = *Abs* (*fst xs-s*) *x X*))
∧
(*wlsInp delta inp* ⟶ *True*)
∧
(*wlsBinp delta binp* ⟶ *True*)
by (*induct rule*: *wls-wlsAbs-wlsInp-wlsBinp.induct*)
(*auto simp add*: *Abs-ainj-ex simp del*: *not-None-eq*)

lemma *wlsAbs-inversion2*[*rule-format*]:
wlsAbs (*xs*,*s*) *A* ⟶
isInBar (*xs*,*s*) ∧ (∃ *x X*. *wls s X* ∧ *A* = *Abs xs x X*)
using *wlsAll-inversion2* **by** *auto*

corollary *wlsAbs-Abs-varSort*:
assumes *X*: *good X* **and** *wlsAbs*: *wlsAbs* (*xs*,*s*) (*Abs xs′ x X*)
shows *xs* = *xs′*
by (*metis Abs-ainj-all X wlsAbs wlsAbs-inversion2 wls-imp-good*)

lemma *wlsAbs*:
wlsAbs (*xs*,*s*) *A* ⟷
isInBar (*xs*,*s*) ∧ (∃ *x X*. *wls s X* ∧ *A* = *Abs xs x X*)
using *Abs wlsAbs-inversion2* **by** *blast*

lemma *wlsAbs-Abs*[*simp*]:
assumes *X*: *good X*
shows *wlsAbs* (*xs′*,*s*) (*Abs xs x X*) = (*isInBar* (*xs′*,*s*) ∧ *xs* = *xs′* ∧ *wls s X*)
using *assms wlsAbs-Abs-varSort* **by** *fastforce*

lemmas *Cons-wls-simps* =
wls-Var-simp wls-Op-simp wls-Abs-simp wlsAbs-Abs

7.4 Induction principles for well-sorted terms

7.4.1 Regular induction

theorem *wls-templateInduct*[*case-names rel Var Op Abs*]:
assumes
rel: ⋀ *s X Y*. ⟦*wls s X*; (*X*,*Y*) ∈ *rel s*⟧ ⟹ *wls s Y* ∧ *skel Y* = *skel X* **and**
Var: ⋀ *xs x*. *phi* (*asSort xs*) (*Var xs x*) **and**
Op:
⋀ *delta inp binp*.
⟦*wlsInp delta inp*; *wlsBinp delta binp*;
liftAll2 phi (*arOf delta*) *inp*; *liftAll2 phiAbs* (*barOf delta*) *binp*⟧
⟹ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
⋀ *s xs x X*.
⟦*isInBar* (*xs*,*s*); *wls s X*; ⋀ *Y*. (*X*,*Y*) ∈ *rel s* ⟹ *phi s Y*⟧
⟹ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows

(*wls s X* ⟶ *phi s X*) ∧
(*wlsAbs* (*xs*,*s*′) *A* ⟶ *phiAbs* (*xs*,*s*′) *A*)
proof−
  have (*good X* ⟶ (∀ *s*. *wls s X* ⟶ *phi s X*)) ∧
      (*goodAbs A* ⟶ (∀ *xs s*. *wlsAbs* (*xs*,*s*) *A* ⟶ *phiAbs* (*xs*,*s*) *A*))
  apply(*induct rule*: *term-templateInduct*[*of* {(*X*, *Y*). ∃ *s*. *wls s X* ∧ (*X*, *Y*) ∈ *rel s*}])
  subgoal using *rel wls-imp-good* **by** *blast*
  subgoal using *Var* **by** *auto*
  subgoal by (*auto intro*!: *Op simp*: *wlsInp-iff wlsBinp-iff liftAll-def liftAll2-def*)
  subgoal using *Abs rel* **by** *simp blast* **.**
  thus *?thesis* **by** *auto*
qed

theorem *wls-rawInduct*[*case-names Var Op Abs*]:
assumes
Var: ⋀ *xs x*. *phi* (*asSort xs*) (*Var xs x*) **and**
Op:
⋀ *delta inp binp*.
  ⟦*wlsInp delta inp*; *wlsBinp delta binp*;
   liftAll2 phi (*arOf delta*) *inp*; *liftAll2 phiAbs* (*barOf delta*) *binp*⟧
  ⟹ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs: ⋀ *s xs x X*. ⟦*isInBar* (*xs*,*s*); *wls s X*; *phi s X*⟧ ⟹ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows
(*wls s X* ⟶ *phi s X*) ∧
(*wlsAbs* (*xs*,*s*′) *A* ⟶ *phiAbs* (*xs*,*s*′) *A*)
by (*induct rule*: *wls-templateInduct*[*of* λ*s*. *Id*]) (*simp-all add*: *assms*)

7.4.2 Fresh induction

First for an unspecified notion of parameter:

theorem *wls-templateInduct-fresh*[*case-names Par Rel Var Op Abs*]:
fixes *s X xs s*′ *A phi phiAbs rel*
and *vars* :: ′*varSort* ⇒ ′*var set*
and *terms* :: ′*sort* ⇒ (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term set*
and *abs* :: (′*varSort* ∗ ′*sort*) ⇒ (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*abs set*
and *envs* :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env set*
assumes
PAR:
⋀ *xs us s*.
  (|*vars xs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*vars xs*)) ∧
  (|*terms s*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*terms s*)) ∧
  (|*abs* (*us*,*s*)| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*abs* (*us*,*s*))) ∧
  (∀ *X* ∈ *terms s*. *wls s X*) ∧
  (∀ *A* ∈ *abs* (*us*,*s*). *wlsAbs* (*us*,*s*) *A*) ∧
  (|*envs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*envs*)) ∧
  (∀ *rho* ∈ *envs*. *wlsEnv rho*) **and**
rel: ⋀ *s X Y*. ⟦*wls s X*; (*X*, *Y*) ∈ *rel s*⟧ ⟹ *wls s Y* ∧ *skel Y* = *skel X* **and**
Var: ⋀ *xs x*. *phi* (*asSort xs*) (*Var xs x*) **and**

Op:
$\bigwedge$ *delta inp binp.*
$[\![$*wlsInp delta inp*; *wlsBinp delta binp*;
liftAll2 (λ*s X. phi s X*) (*arOf delta*) *inp*;
liftAll2 (λ(*us*,*s*) *A. phiAbs* (*us*,*s*) *A*) (*barOf delta*) *binp*$]\!]$
$\Longrightarrow$ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *s xs x X.*
$[\![$*isInBar* (*xs*,*s*); *wls s X*;
$x \notin$ *vars xs*;
$\bigwedge$ *s′ Y. Y* $\in$ *terms s′* $\Longrightarrow$ *fresh xs x Y*;
$\bigwedge$ *xs′ s′ A. A* $\in$ *abs* (*xs′*,*s′*) $\Longrightarrow$ *freshAbs xs x A*;
$\bigwedge$ *rho. rho* $\in$ *envs* $\Longrightarrow$ *freshEnv xs x rho*;
$\bigwedge$ *Y.* $(X,Y) \in$ *rel s* $\Longrightarrow$ *phi s Y*$]\!]$
$\Longrightarrow$ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows
(*wls s X* $\longrightarrow$ *phi s X*) $\wedge$
(*wlsAbs* (*xs*,*s′*) *A* $\longrightarrow$ *phiAbs* (*xs*,*s′*) *A*)
proof−
let *?terms* = $\bigcup$ *s. terms s*
let *?abs* = $\bigcup$ *xs s. abs* (*xs*,*s*)
have $\forall$ *s.* |*terms s*| <*o* |*UNIV* :: *′var set*|
using *PAR var-infinite-INNER finite-ordLess-infinite2* **by** *blast*
hence *1*:|$\bigcup$*s. terms s*| <*o* |*UNIV* :: *′var set*|
using *sort-lt-var-INNER var-regular-INNER regular-UNION* **by** *blast*
have $\forall$ *us s.* |*abs* (*us*,*s*)| <*o* |*UNIV* :: *′var set*|
using *PAR var-infinite-INNER finite-ordLess-infinite2* **by** *blast*
hence $\forall$ *us.* |$\bigcup$*s. abs* (*us*,*s*)| <*o* |*UNIV* :: *′var set*|
by(*auto simp add*: *sort-lt-var-INNER var-regular-INNER regular-UNION*)
hence *2*: |$\bigcup$ *us s. abs* (*us*,*s*)| <*o* |*UNIV* :: *′var set*|
using *varSort-lt-var-INNER var-regular-INNER* **by**(*auto simp add*: *regular-UNION*)

have (*good X* $\longrightarrow$ ($\forall$ *s. wls s X* $\longrightarrow$ *phi s X*)) $\wedge$
(*goodAbs A* $\longrightarrow$ ($\forall$ *xs s. wlsAbs* (*xs*,*s*) *A* $\longrightarrow$ *phiAbs* (*xs*,*s*) *A*))
apply(*induct rule*: *term-templateInduct-fresh*
[*of vars ?terms ?abs envs*
$\{(X,Y).\ \exists$ *s. wls s X* $\wedge$ $(X,Y) \in$ *rel s*$\}$])
subgoal for *xs*
using *PAR 1 2* **apply** *simp-all* **using** *wls-imp-good wlsAbs-imp-goodAbs* **by** *blast+*
subgoal using *assms* **by** *simp* (*meson wls-imp-good*)
subgoal using *assms* **by** *simp*
subgoal using *assms*
by (*smt* (*verit*, *ccfv-threshold*) *case-prodI2′ liftAll2-def liftAll-def wlsBinp-iff wlsInp-iff wls-Op-simp*)
subgoal using *assms* **by** *simp metis*
done
thus *?thesis* **by** *auto*
qed

A version of the above not employing any relation for the abstraction case:

theorem *wls-rawInduct-fresh*[*case-names Par Var Op Abs*]:
fixes *s X xs s' A phi phiAbs*
and *vars* :: *'varSort* ⇒ *'var set*
and *terms* :: *'sort* ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term set*
and *abs* :: (*'varSort* ∗ *'sort*) ⇒ (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs set*
and *envs* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*env set*
assumes
PAR:
⋀ *xs us s.*
(|*vars xs*| <*o* |*UNIV* :: *'var set*| ∨ *finite* (*vars xs*)) ∧
(|*terms s*| <*o* |*UNIV* :: *'var set*| ∨ *finite* (*terms s*)) ∧
(∀ *X* ∈ *terms s. wls s X*) ∧
(|*abs* (*us*,*s*)| <*o* |*UNIV* :: *'var set*| ∨ *finite* (*abs* (*us*,*s*))) ∧
(∀ *A* ∈ *abs* (*us*,*s*). *wlsAbs* (*us*,*s*) *A*) ∧
(|*envs*| <*o* |*UNIV* :: *'var set*| ∨ *finite* (*envs*)) ∧
(∀ *rho* ∈ *envs. wlsEnv rho*) **and**
Var: ⋀ *xs x. phi* (*asSort xs*) (*Var xs x*) **and**
Op:
⋀ *delta inp binp.*
⟦*wlsInp delta inp*; *wlsBinp delta binp*;
liftAll2 (λ*s X. phi s X*) (*arOf delta*) *inp*;
liftAll2 (λ(*us*,*s*) *A. phiAbs* (*us*,*s*) *A*) (*barOf delta*) *binp*⟧
⟹ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
⋀ *s xs x X.*
⟦*isInBar* (*xs*,*s*); *wls s X*;
x ∉ *vars xs*;
⋀ *s' Y. Y* ∈ *terms s'* ⟹ *fresh xs x Y*;
⋀ *us s' A. A* ∈ *abs* (*us*,*s'*) ⟹ *freshAbs xs x A*;
⋀ *rho. rho* ∈ *envs* ⟹ *freshEnv xs x rho*;
phi s X⟧
⟹ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows
(*wls s X* ⟶ *phi s X*) ∧
(*wlsAbs* (*xs*,*s'*) *A* ⟶ *phiAbs* (*xs*,*s'*) *A*)
apply(*induct rule*: *wls-templateInduct-fresh*[*of vars terms abs envs* λ*s. Id*])
using *assms* **by** *auto*

Then for our notion of sorted parameter:

theorem *wls-induct-fresh*[*case-names Par Var Op Abs*]:
fixes *X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and** *s* **and**
A :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* **and** *xs s'* **and**
P :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*,*'sort*)*paramS* **and** *phi phiAbs*
assumes
P: *wlsPar P* **and**
Var: ⋀ *xs x. phi* (*asSort xs*) (*Var xs x*) **and**
Op:
⋀ *delta inp binp.*

$[\![$*wlsInp delta inp*; *wlsBinp delta binp*;
liftAll2 (λ*s X*. *phi s X*) (*arOf delta*) *inp*;
liftAll2 (λ(*us*,*s*) *A*. *phiAbs* (*us*,*s*) *A*) (*barOf delta*) *binp*$]\!]$
$\Longrightarrow$ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *s xs x X*.
$[\![$*isInBar* (*xs*,*s*); *wls s X*;
x $\notin$ *varsOfS P xs*;
$\bigwedge$ *s′ Y*. *Y* $\in$ *termsOfS P s′* $\Longrightarrow$ *fresh xs x Y*;
$\bigwedge$ *us s′ A*. *A* $\in$ *absOfS P* (*us*,*s′*) $\Longrightarrow$ *freshAbs xs x A*;
$\bigwedge$ *rho*. *rho* $\in$ *envsOfS P* $\Longrightarrow$ *freshEnv xs x rho*;
phi s X$]\!]$
$\Longrightarrow$ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows
(*wls s X* $\longrightarrow$ *phi s X*) $\wedge$
(*wlsAbs* (*xs*,*s′*) *A* $\longrightarrow$ *phiAbs* (*xs*,*s′*) *A*)
proof(*induct rule*: *wls-rawInduct-fresh*
[*of varsOfS P termsOfS P absOfS P envsOfS P - - s X xs s′ A*])
case (*Par xs us s*)
then show *?case* **using** *assms* **by**(*cases P*) *simp*
qed(*insert assms*, *simp-all*)

7.4.3 The syntactic constructs are almost free (on well-sorted terms)

theorem *wls-Op-inj*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
and *wlsInp delta′ inp′* **and** *wlsBinp delta′ binp′*
shows
(*Op delta inp binp* = *Op delta′ inp′ binp′*) =
(*delta* = *delta′* $\wedge$ *inp* = *inp′* $\wedge$ *binp* = *binp′*)
using *assms* **by** *simp*

lemma *wls-Abs-ainj-all*:
assumes *wls s X* **and** *wls s′ X′*
shows
(*Abs xs x X* = *Abs xs′ x′ X′*) =
(*xs* = *xs′* $\wedge$
($\forall$ *y*. (*y* = *x* $\vee$ *fresh xs y X*) $\wedge$ (*y* = *x′* $\vee$ *fresh xs y X′*) $\longrightarrow$
(*X* #[*y* $\wedge$ *x*]-*xs*) = (*X′* #[*y* $\wedge$ *x′*]-*xs*)))
using *assms* **by**(*simp add*: *Abs-ainj-all*)

theorem *wls-Abs-swap-all*:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs x′ X′*) =
($\forall$ *y*. (*y* = *x* $\vee$ *fresh xs y X*) $\wedge$ (*y* = *x′* $\vee$ *fresh xs y X′*) $\longrightarrow$
(*X* #[*y* $\wedge$ *x*]-*xs*) = (*X′* #[*y* $\wedge$ *x′*]-*xs*))
using *assms* **by**(*simp add*: *wls-Abs-ainj-all*)

lemma *wls-Abs-ainj-ex*:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs′ x′ X′*) =
(*xs* = *xs′* ∧
(∃ *y*. *y* ∉ {*x*,*x′*} ∧ *fresh xs y X* ∧ *fresh xs y X′* ∧
(*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)))
using *assms* **by**(*simp add*: *Abs-ainj-ex*)

theorem *wls-Abs-swap-ex*:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs x′ X′*) =
(∃ *y*. *y* ∉ {*x*,*x′*} ∧ *fresh xs y X* ∧ *fresh xs y X′* ∧
(*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*))
using *assms* **by**(*simp add*: *wls-Abs-ainj-ex*)

theorem *wls-Abs-inj*[*simp*]:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs x X′*) =
(*X* = *X′*)
using *assms* **by** (*auto simp*: *wls-Abs-swap-all*)

theorem *wls-Abs-swap-cong*[*fundef-cong*]:
assumes *wls s X* **and** *wls s X′*
and *fresh xs y X* **and** *fresh xs y X′* **and** (*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)
shows *Abs xs x X* = *Abs xs x′ X′*
using *assms* **by** (*intro Abs-cong*) *auto*

theorem *wls-Abs-swap-fresh*[*simp*]:
assumes *wls s X* **and** *fresh xs x′ X*
shows *Abs xs x′* (*X* #[*x′* ∧ *x*]-*xs*) = *Abs xs x X*
using *assms* **by**(*simp add*: *Abs-swap-fresh*)

theorem *wls-Var-diff-Op*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows *Var xs x* ≠ *Op delta inp binp*
using *assms* **by** *auto*

theorem *wls-Op-diff-Var*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows *Op delta inp binp* ≠ *Var xs x*
using *assms* **by** *auto*

theorem *wls-nchotomy*:
assumes *wls s X*
shows

$(\exists\ xs\ x.\ asSort\ xs = s \wedge X = Var\ xs\ x) \vee$
$(\exists\ delta\ inp\ binp.\ stOf\ delta = s \wedge wlsInp\ delta\ inp \wedge wlsBinp\ delta\ binp$
$\wedge X = Op\ delta\ inp\ binp)$
using *assms wls.simps* **by** *force*

lemmas *wls-cases* = *wls-wlsAbs-wlsInp-wlsBinp.inducts*(*1*)

lemmas *wlsAbs-nchotomy* = *wlsAbs-inversion2*

theorem *wlsAbs-cases*:
assumes *wlsAbs* (*xs*,*s*) *A*
and $\bigwedge x\ X.\ [\![isInBar\ (xs,s);\ wls\ s\ X]\!] \Longrightarrow phiAbs\ (xs,s)\ (Abs\ xs\ x\ X)$
shows *phiAbs* (*xs*,*s*) *A*
using *assms wlsAbs-nchotomy* **by** *blast*

lemma *wls-disjoint*:
assumes *wls s X* **and** *wls s′ X*
shows $s = s'$
using *assms term-nchotomy wls-imp-good* **by** *fastforce*

lemma *wlsAbs-disjoint*:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wlsAbs* (*xs′*,*s′*) *A*
shows $xs = xs' \wedge s = s'$
using *assms abs-nchotomy wlsAbs-imp-goodAbs wls-disjoint* **by** *fastforce*

lemmas *wls-freeCons* =
Var-inj wls-Op-inj wls-Var-diff-Op wls-Op-diff-Var wls-Abs-swap-fresh

7.5 The non-construct operators preserve well-sortedness

lemma *idEnv-preserves-wls*[*simp*]:
wlsEnv idEnv
proof−
  have *goodEnv idEnv* **by** *simp*
  thus *?thesis* **unfolding** *wlsEnv-def goodEnv-def liftAll-def idEnv-def* **by** *auto*
qed

lemma *updEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wls* (*asSort xs*) *X*
shows *wlsEnv* (*rho* [$x \leftarrow X$]-*xs*)
proof−
  {**fix** *ys*
   let $?L = \{y.\ rho\ ys\ y \neq None\}$
   let $?R = \{y.\ (rho\ [x \leftarrow X]\text{-}xs)\ ys\ y \neq None\}$
   have $?R \leq ?L\ Un\ \{x\}$ **by** *auto*
   hence $|?R| \leq o\ |?L\ Un\ \{x\}|$ **by** *simp*
   moreover
   {**have** $|?L| < o\ |UNIV :: 'var\ set|$
    using *assms* **unfolding** *wlsEnv-def* **by** *simp*

moreover have $|\{x\}| <o\ |UNIV :: 'var\ set|$
using *var-infinite-INNER finite-ordLess-infinite* **by** *auto*
ultimately have $|?L\ Un\ \{x\}| <o\ |UNIV :: 'var\ set|$
using *var-infinite-INNER card-of-Un-ordLess-infinite* **by** *blast*
}
ultimately have $|?R| <o\ |UNIV :: 'var\ set|$
using *ordLeq-ordLess-trans* **by** *blast*
} note *0* = *this*
have *1*: *goodEnv* (*rho* [*x* ← *X*]-*xs*) **using** *assms* **by** *simp*
show *?thesis* **unfolding** *wlsEnv-def goodEnv-def*
using *0 1 assms* **unfolding** *wlsEnv-def liftAll-def* **by** *auto*
qed

lemma *getEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *rho xs x* = *Some X*
shows *wls* (*asSort xs*) *X*
using *assms* **unfolding** *wlsEnv-def liftAll-def* **by** *simp*

lemmas *envOps-preserve-wls* =
idEnv-preserves-wls updEnv-preserves-wls
getEnv-preserves-wls

lemma *psubstAll-preserves-wlsAll*:
assumes *P*: *wlsPar P*
shows
(*wls s X* ⟶ (∀ *rho* ∈ *envsOfS P*. *wls s* (*X* #[*rho*]))) ∧
(*wlsAbs* (*xs*,*s'*) *A* ⟶ (∀ *rho* ∈ *envsOfS P*. *wlsAbs* (*xs*,*s'*) (*A* $[*rho*])))
proof(*induct rule*: *wls-induct-fresh*[*of P*])
case (*Var xs x*)
show *?case*
using *assms* **apply** *safe* **subgoal for** *rho*
apply(*cases rho xs x*) **apply** *simp-all*
using *getEnv-preserves-wls wlsPar-def* **by** *blast+* **.**
next
case (*Op delta inp binp*)
then show *?case* **using** *assms*
by (*auto simp*:
wlsInp-iff psubstInp-def wlsBinp-iff psubstBinp-def liftAll2-def lift-def
sameDom-def intro!: *Op-preserves-wls split*: *option.splits*)
qed(*insert assms*, *auto*)

lemma *psubst-preserves-wls*[*simp*]:
⟦*wls s X*; *wlsEnv rho*⟧ ⟹ *wls s* (*X* #[*rho*])
using *psubstAll-preserves-wlsAll*[*of ParS* (λ-. []) (λ-. []) (λ-. []) [*rho*]]
unfolding *wlsPar-def* **by** *auto*

lemma *psubstAbs-preserves-wls*[*simp*]:
⟦*wlsAbs* (*xs*,*s*) *A*; *wlsEnv rho*⟧ ⟹ *wlsAbs* (*xs*,*s*) (*A* $[*rho*])
using *psubstAll-preserves-wlsAll*[*of ParS* (λ-. []) (λ-. []) (λ-. []) [*rho*]]

unfolding *wlsPar-def* **by** *auto*

lemma *psubstInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsEnv rho*
shows *wlsInp delta* (*inp* %[*rho*])
using *assms* **by** (*auto simp*: *wlsInp-iff psubstInp-def liftAll2-def lift-def*
sameDom-def intro!: *Op-preserves-wls split*: *option.splits*)

lemma *psubstBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp* **and** *wlsEnv rho*
shows *wlsBinp delta* (*binp* %%[*rho*])
using *assms* **by** (*auto simp*: *wlsBinp-iff psubstBinp-def liftAll2-def lift-def*
sameDom-def intro!: *Op-preserves-wls split*: *option.splits*)

lemma *psubstEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
shows *wlsEnv* (*rho* &[*rho'*])
proof−
{**fix** *ys y Y*
assume (*rho* &[*rho'*]) *ys y* = *Some Y*
hence *wls* (*asSort ys*) *Y*
using *assms* **unfolding** *psubstEnv-def wlsEnv-def liftAll-def*
by (*cases rho ys y*) (*auto simp add*: *assms*)
}
moreover have *goodEnv* (*rho* &[*rho'*]) **using** *assms* **by** *simp*
ultimately show *?thesis*
unfolding *goodEnv-def wlsEnv-def psubstEnv-def wlsEnv-def liftAll-def*
by (*auto simp add*: *assms*)
qed

lemmas *psubstAll-preserve-wls* =
psubst-preserves-wls psubstAbs-preserves-wls
psubstInp-preserves-wls psubstBinp-preserves-wls
psubstEnv-preserves-wls

lemma *subst-preserves-wls*[*simp*]:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
shows *wls s* (*X* #[*Y* / *y*]-*ys*)
using *assms* **unfolding** *subst-def* **by** *simp*

lemma *substAbs-preserves-wls*[*simp*]:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
shows *wlsAbs* (*xs*,*s*) (*A* $[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substAbs-def* **by** *simp*

lemma *substInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp* **and** *wls* (*asSort ys*) *Y*
shows *wlsInp delta* (*inp* %[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substInp-def* **by** *simp*

lemma *substBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp* **and** *wls* (*asSort ys*) *Y*
shows *wlsBinp delta* (*binp* %%[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substBinp-def* **by** *simp*

lemma *substEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
shows *wlsEnv* (*rho* &[*Y* / *y*]-*ys*)
using *assms* **unfolding** *substEnv-def* **by** *simp*

lemmas *substAll-preserve-wls* =
subst-preserves-wls substAbs-preserves-wls
substInp-preserves-wls substBinp-preserves-wls
substEnv-preserves-wls

lemma *vsubst-preserves-wls*[*simp*]:
assumes *wls s Y*
shows *wls s* (*Y* #[*x1* // *x*]-*xs*)
using *assms* **unfolding** *vsubst-def* **by** *simp*

lemma *vsubstAbs-preserves-wls*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows *wlsAbs* (*us*,*s*) (*A* $[*x1* // *x*]-*xs*)
using *assms* **unfolding** *vsubstAbs-def* **by** *simp*

lemma *vsubstInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp*
shows *wlsInp delta* (*inp* %[*x1* // *x*]-*xs*)
using *assms* **unfolding** *vsubstInp-def* **by** *simp*

lemma *vsubstBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp*
shows *wlsBinp delta* (*binp* %%[*x1* // *x*]-*xs*)
using *assms* **unfolding** *vsubstBinp-def* **by** *simp*

lemma *vsubstEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho*
shows *wlsEnv* (*rho* &[*x1* // *x*]-*xs*)
using *assms* **unfolding** *vsubstEnv-def* **by** *simp*

lemmas *vsubstAll-preserve-wls* = *vsubst-preserves-wls vsubstAbs-preserves-wls*
vsubstInp-preserves-wls vsubstBinp-preserves-wls vsubstEnv-preserves-wls

lemmas *all-preserve-wls* = *Cons-preserve-wls swapAll-preserve-wls psubstAll-preserve-wls*
envOps-preserve-wls
substAll-preserve-wls vsubstAll-preserve-wls

7.6 Simplification rules for swapping, substitution, freshness and skeleton

theorem *wls-swap-Op-simp*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows
((*Op delta inp binp*) #[*x1* ∧ *x2*]-*xs*) =
Op delta (*inp* %[*x1* ∧ *x2*]-*xs*) (*binp* %%[*x1* ∧ *x2*]-*xs*)
using *assms* **by** *simp*

theorem *wls-swapAbs-simp*[*simp*]:
assumes *wls s X*
shows ((*Abs xs x X*) $[*y1* ∧ *y2*]-*ys*) = *Abs xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*) (*X* #[*y1* ∧ *y2*]-*ys*)
using *assms* **by** *simp*

lemmas *wls-swapAll-simps* =
swap-Var-simp wls-swap-Op-simp wls-swapAbs-simp

theorem *wls-fresh-Op-simp*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows
fresh xs x (*Op delta inp binp*) =
(*freshInp xs x inp* ∧ *freshBinp xs x binp*)
using *assms* **by** *simp*

theorem *wls-freshAbs-simp*[*simp*]:
assumes *wls s X*
shows *freshAbs ys y* (*Abs xs x X*) = (*ys* = *xs* ∧ *y* = *x* ∨ *fresh ys y X*)
using *assms* **by** *simp*

lemmas *wls-freshAll-simps* =
fresh-Var-simp wls-fresh-Op-simp wls-freshAbs-simp

theorem *wls-skel-Op-simp*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows
skel (*Op delta inp binp*) = *Branch* (*skelInp inp*) (*skelBinp binp*)
using *assms* **by** *simp*

lemma *wls-skelInp-def2*:
assumes *wlsInp delta inp*
shows *skelInp inp* = *lift skel inp*
using *assms* **by**(*simp add*: *skelInp-def2*)

lemma *wls-skelBinp-def2*:
assumes *wlsBinp delta binp*
shows *skelBinp binp* = *lift skelAbs binp*
using *assms* **by**(*simp add*: *skelBinp-def2*)

theorem *wls-skelAbs-simp*[*simp*]:
assumes *wls s X*
shows *skelAbs* (*Abs xs x X*) = *Branch* (λi. *Some* (*skel X*)) *Map.empty*
using *assms* **by** *simp*

lemmas *wls-skelAll-simps* =
skel-Var-simp wls-skel-Op-simp wls-skelAbs-simp

theorem *wls-psubst-Var-simp1*[*simp*]:
assumes *wlsEnv rho* **and** *rho xs x* = *None*
shows ((*Var xs x*) #[*rho*]) = *Var xs x*
using *assms* **by** *simp*

theorem *wls-psubst-Var-simp2*[*simp*]:
assumes *wlsEnv rho* **and** *rho xs x* = *Some X*
shows ((*Var xs x*) #[*rho*]) = *X*
using *assms* **by** *simp*

theorem *wls-psubst-Op-simp*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp* **and** *wlsEnv rho*
shows
((*Op delta inp binp*) #[*rho*]) = *Op delta* (*inp* %[*rho*]) (*binp* %%[*rho*])
using *assms* **by** *simp*

theorem *wls-psubstAbs-simp*[*simp*]:
assumes *wls s X* **and** *wlsEnv rho* **and** *freshEnv xs x rho*
shows ((*Abs xs x X*) $[*rho*]) = *Abs xs x* (*X* #[*rho*])
using *assms* **by** *simp*

lemmas *wls-psubstAll-simps* =
wls-psubst-Var-simp1 wls-psubst-Var-simp2 wls-psubst-Op-simp wls-psubstAbs-simp

lemmas *wls-envOps-simps* =
getEnv-idEnv getEnv-updEnv1 getEnv-updEnv2

theorem *wls-subst-Var-simp1*[*simp*]:
assumes *wls* (*asSort ys*) *Y*
and *ys* $\neq$ *xs* $\vee$ *y* $\neq$ *x*
shows ((*Var xs x*) #[*Y* / *y*]-*ys*) = *Var xs x*
using *assms* **unfolding** *subst-def* **by** *auto*

theorem *wls-subst-Var-simp2*[*simp*]:
assumes *wls* (*asSort xs*) *Y*
shows ((*Var xs x*) #[*Y* / *x*]-*xs*) = *Y*
using *assms* **unfolding** *subst-def* **by** *auto*

theorem *wls-subst-Op-simp*[*simp*]:
assumes *wls* (*asSort ys*) *Y*
and *wlsInp delta inp* **and** *wlsBinp delta binp*
shows
((*Op delta inp binp*) #[*Y* / *y*]-*ys*) =
Op delta (*inp* %[*Y* / *y*]-*ys*) (*binp* %%[*Y* / *y*]-*ys*)
using *assms* **unfolding** *subst-def substInp-def*
substAbs-def substBinp-def **by** *auto*

theorem *wls-substAbs-simp*[*simp*]:
assumes *wls* (*asSort ys*) *Y*
and *wls s X* **and** $xs \neq ys \vee x \neq y$ **and** *fresh xs x Y*
shows ((*Abs xs x X*) $[*Y* / *y*]-*ys*) = *Abs xs x* (*X* #[*Y* / *y*]-*ys*)
proof−
have *freshEnv xs x* (*idEnv* [$y \leftarrow Y$]-*ys*) **unfolding** *freshEnv-def liftAll-def*
using *assms* **by** *simp*
thus *?thesis* **using** *assms* **unfolding** *subst-def substAbs-def* **by** *auto*
qed

lemmas *wls-substAll-simps* =
wls-subst-Var-simp1 wls-subst-Var-simp2 wls-subst-Op-simp wls-substAbs-simp

theorem *wls-vsubst-Op-simp*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows
((*Op delta inp binp*) #[*y1* // *y*]-*ys*) =
Op delta (*inp* %[*y1* // *y*]-*ys*) (*binp* %%[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstInp-def*
vsubstAbs-def vsubstBinp-def **by** *simp*

theorem *wls-vsubstAbs-simp*[*simp*]:
assumes *wls s X* **and**
$xs \neq ys \vee x \notin \{y,y1\}$
shows ((*Abs xs x X*) $[*y1* // *y*]-*ys*) = *Abs xs x* (*X* #[*y1* // *y*]-*ys*)
using *assms* **unfolding** *vsubst-def vsubstAbs-def* **by** *simp*

lemmas *wls-vsubstAll-simps* =
vsubst-Var-simp wls-vsubst-Op-simp wls-vsubstAbs-simp

theorem *wls-swapped-skel*:

assumes *wls s X* **and** $(X, Y) \in$ *swapped*
shows *skel Y = skel X*
apply(*rule swapped-skel*) **using** *assms* **by** *auto*

theorem *wls-obtain-rep*:
assumes *wls s X* **and** *FRESH*: *fresh xs x′ X*
shows ∃ *X′*. *skel X′ = skel X* ∧ (*X*,*X′*) ∈ *swapped* ∧ *wls s X′* ∧ *Abs xs x X* = *Abs xs x′ X′*
proof−
  have *0*: *skel* (*X* #[*x′* ∧ *x*]-*xs*) = *skel X* **using** *assms* **by**(*simp add*: *skel-swap*)
  have *1*: *wls s* (*X* #[*x′* ∧ *x*]-*xs*) **using** *assms swap-preserves-wls* **by** *auto*
  have *2*: (*X*, *X* #[*x′* ∧ *x*]-*xs*) ∈ *swapped* **using** *Var swap-swapped* **by** *auto*
  show *?thesis* **using** *assms 0 1 2* **by** *fastforce*
qed

lemmas *wls-allOpers-simps* =
wls-swapAll-simps
wls-freshAll-simps
wls-skelAll-simps
wls-envOps-simps
wls-psubstAll-simps
wls-substAll-simps
wls-vsubstAll-simps

7.7 The ability to pick fresh variables

theorem *wls-single-non-fresh-ordLess-var*:
wls s X ⟹ |{*x*. ¬ *fresh xs x X*}| <*o* |*UNIV* :: ′*var set*|
by(*simp add*: *single-non-fresh-ordLess-var*)

theorem *wls-single-non-freshAbs-ordLess-var*:
wlsAbs (*us*,*s*) *A* ⟹ |{*x*. ¬ *freshAbs xs x A*}| <*o* |*UNIV* :: ′*var set*|
by(*simp add*: *single-non-freshAbs-ordLess-var*)

theorem *wls-obtain-fresh*:
fixes *V*::′*varSort* ⇒ ′*var set* **and**
    XS::′*sort* ⇒ (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term set* **and**
    AS::′*varSort* ⇒ ′*sort* ⇒ (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*abs set* **and**
    Rho::(′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*env set* **and** *zs*
assumes *VVar*: ∀ *xs*. |*V xs*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*V xs*)
and *XSVar*: ∀ *s*. |*XS s*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*XS s*)
and *ASVar*: ∀ *xs s*. |*AS xs s*| <*o* |*UNIV* :: ′*var set*| ∨ *finite* (*AS xs s*)
and *XSwls*: ∀ *s*. ∀ *X* ∈ *XS s*. *wls s X*
and *ASwls*: ∀ *xs s*. ∀ *A* ∈ *AS xs s*. *wlsAbs* (*xs*,*s*) *A*
and *RhoVar*: |*Rho*| <*o* |*UNIV* :: ′*var set*| ∨ *finite Rho*
and *Rhowls*: ∀ *rho* ∈ *Rho*. *wlsEnv rho*
shows
∃ *z*. (∀ *xs*. *z* ∉ *V xs*) ∧
    (∀ *s*. ∀ *X* ∈ *XS s*. *fresh zs z X*) ∧

(∀ *xs s*. ∀ *A* ∈ *AS xs s*. *freshAbs zs z A*) ∧
(∀ *rho* ∈ *Rho*. *freshEnv zs z rho*)
proof−
let *?VG* = ⋃ *xs*. *V xs* **let** *?XSG* = ⋃ *s*. *XS s* **let** *?ASG* = ⋃ *xs s*. *AS xs s*
have ∀ *xs*. |*V xs*| <*o* |*UNIV* :: *'var set*| **using** *VVar finite-ordLess-var* **by** *auto*
hence *1*: |*?VG*| <*o* |*UNIV* :: *'var set*|
using *var-regular-INNER varSort-lt-var-INNER regular-UNION* **by** *blast*
have ∀ *s*. |*XS s*| <*o* |*UNIV* :: *'var set*| **using** *XSVar finite-ordLess-var* **by** *auto*
hence *2*: |*?XSG*| <*o* |*UNIV* :: *'var set*|
using *var-regular-INNER sort-lt-var-INNER regular-UNION* **by** *blast*
have ∀ *xs s*. |*AS xs s*| <*o* |*UNIV* :: *'var set*| **using** *ASVar finite-ordLess-var* **by** *auto*
hence ∀ *xs*. |⋃ *s*. *AS xs s*| <*o* |*UNIV* :: *'var set*|
using *var-regular-INNER sort-lt-var-INNER regular-UNION* **by** *blast*
hence *3*: |*?ASG*| <*o* |*UNIV* :: *'var set*|
using *var-regular-INNER varSort-lt-var-INNER* **by** (*auto simp add*: *regular-UNION*)
have ∃ *z*. *z* ∉ *?VG* ∧
(∀ *X* ∈ *?XSG*. *fresh zs z X*) ∧
(∀ *A* ∈ *?ASG*. *freshAbs zs z A*) ∧
(∀ *rho* ∈ *Rho*. *freshEnv zs z rho*)
using *assms 1 2 3* **by** (*intro obtain-fresh*) *fastforce+*
thus *?thesis* **by** *auto*
qed

theorem *wls-obtain-fresh-paramS*:
assumes *wlsPar P*
shows
∃ *z*.
(∀ *xs*. *z* ∉ *varsOfS P xs*) ∧
(∀ *s*. ∀ *X* ∈ *termsOfS P s*. *fresh zs z X*) ∧
(∀ *us s*. ∀ *A* ∈ *absOfS P* (*us*,*s*). *freshAbs zs z A*) ∧
(∀ *rho* ∈ *envsOfS P*. *freshEnv zs z rho*)
using *assms* **by**(*cases P*) (*auto intro*: *wls-obtain-fresh*)

lemma *wlsAbs-freshAbs-nchotomy*:
assumes *A*: *wlsAbs* (*xs*,*s*) *A* **and** *fresh*: *freshAbs xs x A*
shows ∃ *X*. *wls s X* ∧ *A* = *Abs xs x X*
proof−
{**assume** *wlsAbs* (*xs*,*s*) *A*
hence *freshAbs xs x A* ⟶ (∃ *X*. *wls s X* ∧ *A* = *Abs xs x X*)
using *fresh wls-obtain-rep*[*of s - xs x*] **by** (*fastforce elim*!: *wlsAbs-cases*)
}
thus *?thesis* **using** *assms* **by** *auto*
qed

theorem *wlsAbs-fresh-nchotomy*:
assumes *A*: *wlsAbs* (*xs*,*s*) *A* **and** *P*: *wlsPar P*
shows ∃ *x X*. *A* = *Abs xs x X* ∧
wls s X ∧

(∀ *ys*. $x \notin$ *varsOfS P ys*) ∧
(∀ *s'*. ∀ *Y* ∈ *termsOfS P s'*. *fresh xs x Y*) ∧
(∀ *us s'*. ∀ *B* ∈ *absOfS P* (*us*,*s'*). *freshAbs xs x B*) ∧
(∀ *rho* ∈ *envsOfS P*. *freshEnv xs x rho*)
proof−
let *?chi* =
λ *x*. (∀ *xs*. $x \notin$ *varsOfS P xs*) ∧
(∀ *s'*. ∀ *Y* ∈ *termsOfS P s'*. *fresh xs x Y*) ∧
(∀ *us s'*.∀ *B* ∈ (*if us* = *xs* ∧ *s'* = *s then* {*A*} *else* {}) ∪ *absOfS P* (*us*,*s'*).
freshAbs xs x B) ∧
(∀ *rho* ∈ *envsOfS P*. *freshEnv xs x rho*)
have ∃ *x*. *?chi x*
using *A P* **by** (*intro wls-obtain-fresh*) (*cases P*, *auto*)+
then obtain *x* **where** *1*: *?chi x* **by** *blast*
hence *freshAbs xs x A* **by** *fastforce*
then obtain *X* **where** *X*: *wls s X* **and** *2*: *A* = *Abs xs x X*
using *A 1 wlsAbs-freshAbs-nchotomy*[*of xs s A x*] **by** *auto*
thus *?thesis* **using** *1* **by** *blast*
qed

theorem *wlsAbs-fresh-cases*:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wlsPar P*
and ⋀ *x X*.
⟦*wls s X*;
⋀ *ys*. $x \notin$ *varsOfS P ys*;
⋀ *s' Y*. *Y* ∈ *termsOfS P s'* ⟹ *fresh xs x Y*;
⋀ *us s' B*. *B* ∈ *absOfS P* (*us*,*s'*) ⟹ *freshAbs xs x B*;
⋀ *rho*. *rho* ∈ *envsOfS P* ⟹ *freshEnv xs x rho*⟧
⟹ *phi* (*xs*,*s*) (*Abs xs x X*) *P*
shows *phi* (*xs*,*s*) *A P*
by (*metis assms wlsAbs-fresh-nchotomy*)

7.8 Compositionality properties of freshness and swapping

7.8.1 W.r.t. terms

theorem *wls-swap-ident*[*simp*]:
assumes *wls s X*
shows (*X* #[*x* ∧ *x*]-*xs*) = *X*
using *assms* **by** *simp*

theorem *wls-swap-compose*:
assumes *wls s X*
shows ((*X* #[*x* ∧ *y*]-*zs*) #[*x'* ∧ *y'*]-*zs'*) =
((*X* #[*x'* ∧ *y'*]-*zs'*) #[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]-*zs*)
using *assms* **by** (*intro swap-compose*) *auto*

theorem *wls-swap-commute*:
⟦*wls s X*; *zs* ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {}⟧ ⟹
((*X* #[*x* ∧ *y*]-*zs*) #[*x'* ∧ *y'*]-*zs'*) = ((*X* #[*x'* ∧ *y'*]-*zs'*) #[*x* ∧ *y*]-*zs*)

by (*intro swap-commute*) *auto*

theorem *wls-swap-involutive*[*simp*]:
assumes *wls s X*
shows $((X \ \#[x \wedge y]\text{-}zs) \ \#[x \wedge y]\text{-}zs) = X$
using *assms* **by** *simp*

theorem *wls-swap-inj*[*simp*]:
assumes *wls s X* **and** *wls s X'*
shows
$((X \ \#[x \wedge y]\text{-}zs) = (X' \ \#[x \wedge y]\text{-}zs)) =$
$(X = X')$
using *assms* **by** (*metis wls-swap-involutive*)

theorem *wls-swap-involutive2*[*simp*]:
assumes *wls s X*
shows $((X \ \#[x \wedge y]\text{-}zs) \ \#[y \wedge x]\text{-}zs) = X$
using *assms* **by** (*simp add*: *swap-sym*)

theorem *wls-swap-preserves-fresh*[*simp*]:
assumes *wls s X*
shows *fresh xs* $(x \ @xs[y1 \wedge y2]\text{-}ys) \ (X \ \#[y1 \wedge y2]\text{-}ys)$ = *fresh xs x X*
using *assms* **by** *simp*

theorem *wls-swap-preserves-fresh-distinct*:
assumes *wls s X* **and**
$xs \neq ys \vee x \notin \{y1, y2\}$
shows *fresh xs x* $(X \ \#[y1 \wedge y2]\text{-}ys)$ = *fresh xs x X*
using *assms* **by**(*intro swap-preserves-fresh-distinct*) *auto*

theorem *wls-fresh-swap-exchange1*:
assumes *wls s X*
shows *fresh xs x2* $(X \ \#[x1 \wedge x2]\text{-}xs)$ = *fresh xs x1 X*
using *assms* **by** (*intro fresh-swap-exchange1*) *auto*

theorem *wls-fresh-swap-exchange2*:
assumes *wls s X*
shows *fresh xs x2* $(X \ \#[x2 \wedge x1]\text{-}xs)$ = *fresh xs x1 X*
using *assms* **by** (*intro fresh-swap-exchange2*) *fastforce+*

theorem *wls-fresh-swap-id*[*simp*]:
assumes *wls s X* **and** *fresh xs x1 X* **and** *fresh xs x2 X*
shows $(X \ \#[x1 \wedge x2]\text{-}xs) = X$
using *assms* **by** *simp*

theorem *wls-fresh-swap-compose*:
assumes *wls s X* **and** *fresh xs y X* **and** *fresh xs z X*

shows $((X \#[y \wedge x]\text{-}xs) \#[z \wedge y]\text{-}xs) = (X \#[z \wedge x]\text{-}xs)$
using *assms* **by** (*intro fresh-swap-compose*) *auto*

theorem *wls-skel-swap*:
assumes *wls s X*
shows *skel* $(X \#[x1 \wedge x2]\text{-}xs)$ = *skel X*
using *assms* **by** (*intro skel-swap*) *auto*

7.8.2 W.r.t. environments

theorem *wls-swapEnv-ident*[*simp*]:
assumes *wlsEnv rho*
shows $(rho \,\&[x \wedge x]\text{-}xs) = rho$
using *assms* **by** *simp*

theorem *wls-swapEnv-compose*:
assumes *wlsEnv rho*
shows $((rho \,\&[x \wedge y]\text{-}zs) \,\&[x' \wedge y']\text{-}zs') =$
$((rho \,\&[x' \wedge y']\text{-}zs') \,\&[(x \,@zs[x' \wedge y']\text{-}zs') \wedge (y \,@zs[x' \wedge y']\text{-}zs')]\text{-}zs)$
using *assms* **by** (*intro swapEnv-compose*) *auto*

theorem *wls-swapEnv-commute*:
$[\![wlsEnv\ rho;\ zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\}]\!] \Longrightarrow$
$((rho \,\&[x \wedge y]\text{-}zs) \,\&[x' \wedge y']\text{-}zs') = ((rho \,\&[x' \wedge y']\text{-}zs') \,\&[x \wedge y]\text{-}zs)$
by (*intro swapEnv-commute*) *fastforce+*

theorem *wls-swapEnv-involutive*[*simp*]:
assumes *wlsEnv rho*
shows $((rho \,\&[x \wedge y]\text{-}zs) \,\&[x \wedge y]\text{-}zs) = rho$
using *assms* **by** *simp*

theorem *wls-swapEnv-inj*[*simp*]:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
shows
$((rho \,\&[x \wedge y]\text{-}zs) = (rho' \,\&[x \wedge y]\text{-}zs)) =$
$(rho = rho')$
by (*metis assms wls-swapEnv-involutive*)

theorem *wls-swapEnv-involutive2*[*simp*]:
assumes *wlsEnv rho*
shows $((rho \,\&[x \wedge y]\text{-}zs) \,\&[y \wedge x]\text{-}zs) = rho$
using *assms* **by**(*simp add*: *swapEnv-sym*)

theorem *wls-swapEnv-preserves-freshEnv*[*simp*]:
assumes *wlsEnv rho*
shows *freshEnv xs* $(x \,@xs[y1 \wedge y2]\text{-}ys)$ $(rho \,\&[y1 \wedge y2]\text{-}ys)$ = *freshEnv xs x rho*
using *assms* **by** *simp*

theorem *wls-swapEnv-preserves-freshEnv-distinct*:
assumes *wlsEnv rho*
xs ≠ *ys* ∨ *x* ∉ {*y1*,*y2*}
shows *freshEnv xs x* (*rho* &[*y1* ∧ *y2*]-*ys*) = *freshEnv xs x rho*
using *assms* **by** (*intro swapEnv-preserves-freshEnv-distinct*) *auto*

theorem *wls-freshEnv-swapEnv-exchange1*:
assumes *wlsEnv rho*
shows *freshEnv xs x2* (*rho* &[*x1* ∧ *x2*]-*xs*) = *freshEnv xs x1 rho*
using *assms* **by** (*intro freshEnv-swapEnv-exchange1*) *auto*

theorem *wls-freshEnv-swapEnv-exchange2*:
assumes *wlsEnv rho*
shows *freshEnv xs x2* (*rho* &[*x2* ∧ *x1*]-*xs*) = *freshEnv xs x1 rho*
using *assms* **by** (*intro freshEnv-swapEnv-exchange2*) *auto*

theorem *wls-freshEnv-swapEnv-id*[*simp*]:
assumes *wlsEnv rho* **and** *freshEnv xs x1 rho* **and** *freshEnv xs x2 rho*
shows (*rho* &[*x1* ∧ *x2*]-*xs*) = *rho*
using *assms* **by** *simp*

theorem *wls-freshEnv-swapEnv-compose*:
assumes *wlsEnv rho* **and** *freshEnv xs y rho* **and** *freshEnv xs z rho*
shows ((*rho* &[*y* ∧ *x*]-*xs*) &[*z* ∧ *y*]-*xs*) = (*rho* &[*z* ∧ *x*]-*xs*)
using *assms* **by** (*intro freshEnv-swapEnv-compose*) *auto*

7.8.3 W.r.t. abstractions

theorem *wls-swapAbs-ident*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ (*A* $[*x* ∧ *x*]-*xs*) = *A*
by (*elim wlsAbs-cases*) *auto*

theorem *wls-swapAbs-compose*:
wlsAbs (*us*,*s*) *A* ⟹
((*A* $[*x* ∧ *y*]-*zs*) $[*x'* ∧ *y'*]-*zs'*) =
((*A* $[*x'* ∧ *y'*]-*zs'*) $[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]-*zs*)
by (*erule wlsAbs-cases*) (*simp*, *metis sw-compose wls-swap-compose*)

theorem *wls-swapAbs-commute*:
assumes *zs* ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {}
shows
wlsAbs (*us*,*s*) *A* ⟹
((*A* $[*x* ∧ *y*]-*zs*) $[*x'* ∧ *y'*]-*zs'*) = ((*A* $[*x'* ∧ *y'*]-*zs'*) $[*x* ∧ *y*]-*zs*)
using *assms* **by** (*elim wlsAbs-cases*) (*simp add*: *sw-commute wls-swap-commute*)

theorem *wls-swapAbs-involutive*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ ((*A* $[*x* ∧ *y*]-*zs*) $[*x* ∧ *y*]-*zs*) = *A*
by (*erule wlsAbs-cases*) *simp-all*

theorem *wls-swapAbs-sym*:
wlsAbs (*us*,*s*) *A* $\Longrightarrow$ (*A* $[*x* $\wedge$ *y*]-*zs*) = (*A* $[*y* $\wedge$ *x*]-*zs*)
by (*erule wlsAbs-cases*) (*auto simp add*: *swap-sym sw-sym*)

theorem *wls-swapAbs-inj*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsAbs* (*us*,*s*) *A′*
shows
((*A* $[*x* $\wedge$ *y*]-*zs*) = (*A′* $[*x* $\wedge$ *y*]-*zs*)) =
(*A* = *A′*)
by (*metis assms wls-swapAbs-involutive*)

theorem *wls-swapAbs-involutive2*[*simp*]:
wlsAbs (*us*,*s*) *A* $\Longrightarrow$ ((*A* $[*x* $\wedge$ *y*]-*zs*) $[*y* $\wedge$ *x*]-*zs*) = *A*
using *wls-swapAbs-sym*[*of us s A zs x y*] **by** *auto*

theorem *wls-swapAbs-preserves-freshAbs*[*simp*]:
wlsAbs (*us*,*s*) *A*
$\Longrightarrow$ *freshAbs xs* (*x* @*xs*[*y1* $\wedge$ *y2*]-*ys*) (*A* $[*y1* $\wedge$ *y2*]-*ys*) = *freshAbs xs x A*
by (*erule wlsAbs-cases*)
(*simp-all add*: *sw-def wls-fresh-swap-exchange1 wls-fresh-swap-exchange2*
wls-swap-preserves-fresh-distinct)

theorem *wls-swapAbs-preserves-freshAbs-distinct*:
⟦*wlsAbs* (*us*,*s*) *A*; *xs* $\neq$ *ys* $\vee$ *x* $\notin$ {*y1*,*y2*}⟧
$\Longrightarrow$ *freshAbs xs x* (*A* $[*y1* $\wedge$ *y2*]-*ys*) = *freshAbs xs x A*
apply(*erule wlsAbs-cases*) **apply** *simp-all*
unfolding *sw-def* **by** (*auto simp*: *wls-swap-preserves-fresh-distinct*)

theorem *wls-freshAbs-swapAbs-exchange1*:
wlsAbs (*us*,*s*) *A*
$\Longrightarrow$ *freshAbs xs x2* (*A* $[*x1* $\wedge$ *x2*]-*xs*) = *freshAbs xs x1 A*
apply(*erule wlsAbs-cases*) **apply** *simp-all*
unfolding *sw-def* **by** (*auto simp add*: *wls-fresh-swap-exchange1*)

theorem *wls-freshAbs-swapAbs-exchange2*:
wlsAbs (*us*,*s*) *A*
$\Longrightarrow$ *freshAbs xs x2* (*A* $[*x2* $\wedge$ *x1*]-*xs*) = *freshAbs xs x1 A*
apply(*erule wlsAbs-cases*) **apply** *simp-all*
unfolding *sw-def* **by** (*auto simp add*: *wls-fresh-swap-exchange2*)

theorem *wls-freshAbs-swapAbs-id*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs xs x1 A* **and** *freshAbs xs x2 A*
shows (*A* $[*x1* $\wedge$ *x2*]-*xs*) = *A*
using *assms* **by** *simp*

lemma *wls-freshAbs-swapAbs-compose-aux*:
⟦*wlsAbs* (*us*,*s*) *A*; *wlsPar P*⟧ $\Longrightarrow$

∀ x y z. {x,y,z} ⊆ *varsOfS P xs* ∧ *freshAbs xs y A* ∧ *freshAbs xs z A* ⟶
((*A* $[*y* ∧ *x*]-*xs*) $[*z* ∧ *y*]-*xs*) = (*A* $[*z* ∧ *x*]-*xs*)
apply(*erule wlsAbs-fresh-cases*)
by *simp-all* (*metis fresh-swap-compose sw-def wls-imp-good*)

theorem *wls-freshAbs-swapAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs xs y A* **and** *freshAbs xs z A*
shows ((*A* $[*y* ∧ *x*]-*xs*) $[*z* ∧ *y*]-*xs*) = (*A* $[*z* ∧ *x*]-*xs*)
proof−
let *?P* =
ParS (λ*xs'*. *if xs'* = *xs then* [*x*,*y*,*z*] *else* []) (λ*s*.[]) (λ-. []) [] ::
('*index*, '*bindex*, '*varSort*, '*var*, '*opSym*, '*sort*) *paramS*
show *?thesis*
using *assms wls-freshAbs-swapAbs-compose-aux*[*of us s A ?P xs*]
unfolding *wlsPar-def* **by** *simp*
qed

theorem *wls-skelAbs-swapAbs*:
wlsAbs (*us*,*s*) *A*
⟹ *skelAbs* (*A* $[*x1* ∧ *x2*]-*xs*) = *skelAbs A*
by (*erule wlsAbs-cases*) (*auto simp*: *wls-skel-swap*)

lemmas *wls-swapAll-freshAll-otherSimps* =
wls-swap-ident wls-swap-involutive wls-swap-inj wls-swap-involutive2 wls-swap-preserves-fresh
wls-fresh-swap-id

wls-swapAbs-ident wls-swapAbs-involutive wls-swapAbs-inj wls-swapAbs-involutive2
wls-swapAbs-preserves-freshAbs
wls-freshAbs-swapAbs-id

wls-swapEnv-ident wls-swapEnv-involutive wls-swapEnv-inj wls-swapEnv-involutive2
wls-swapEnv-preserves-freshEnv
wls-freshEnv-swapEnv-id

7.9 Compositionality properties for the other operators

7.9.1 Environment identity, update and "get" versus other operators

theorem *wls-psubst-idEnv*[*simp*]:
wls s X ⟹ (*X* #[*idEnv*]) = *X*
by *simp*

theorem *wls-psubstEnv-idEnv-id*[*simp*]:
wlsEnv rho ⟹ (*rho* &[*idEnv*]) = *rho*
by *simp*

theorem *wls-swapEnv-updEnv-fresh*:
assumes *zs* ≠ *ys* ∨ *y* ∉ {*z1*,*z2*} **and** *wls* (*asSort ys*) *Y*
and *fresh zs z1 Y* **and** *fresh zs z2 Y*
shows ((*rho* [*y* ← *Y*]-*ys*) &[*z1* ∧ *z2*]-*zs*) = ((*rho* &[*z1* ∧ *z2*]-*zs*) [*y* ← *Y*]-*ys*)
using *assms* **by** (*simp add*: *swapEnv-updEnv-fresh*)

7.9.2 Substitution versus other operators

theorem *wls-fresh-psubst*:
assumes *wls s X* **and** *wlsEnv rho*
shows
fresh zs z (*X* #[*rho*]) =
 (∀ *ys y*. *fresh ys y X* ∨ *freshImEnvAt zs z rho ys y*)
using *assms* **by**(*simp add*: *fresh-psubst*)

theorem *wls-fresh-psubst-E1*:
assumes *wls s X* **and** *wlsEnv rho*
and *rho ys y* = *None* **and** *fresh zs z* (*X* #[*rho*])
shows *fresh ys y X* ∨ (*ys* ≠ *zs* ∨ *y* ≠ *z*)
using *assms fresh-psubst-E1*[*of X rho ys y zs z*] **by** *simp*

theorem *wls-fresh-psubst-E2*:
assumes *wls s X* **and** *wlsEnv rho*
and *rho ys y* = *Some Y* **and** *fresh zs z* (*X* #[*rho*])
shows *fresh ys y X* ∨ *fresh zs z Y*
using *assms fresh-psubst-E2*[*of X rho ys y Y zs z*] **by** *simp*

theorem *wls-fresh-psubst-I1*:
assumes *wls s X* **and** *wlsEnv rho*
and *fresh zs z X* **and** *freshEnv zs z rho*
shows *fresh zs z* (*X* #[*rho*])
using *assms* **by**(*simp add*: *fresh-psubst-I1*)

theorem *wls-psubstEnv-preserves-freshEnv*:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
and *fresh*: *freshEnv zs z rho* *freshEnv zs z rho'*
shows *freshEnv zs z* (*rho* &[*rho'*])
using *assms* **by**(*simp add*: *psubstEnv-preserves-freshEnv*)

theorem *wls-fresh-psubst-I*:
assumes *wls s X* **and** *wlsEnv rho*
and *rho zs z* = *None* ⟹ *fresh zs z X* **and**
  ⋀ *ys y Y*. *rho ys y* = *Some Y* ⟹ *fresh ys y X* ∨ *fresh zs z Y*
shows *fresh zs z* (*X* #[*rho*])
using *assms* **by**(*simp add*: *fresh-psubst-I*)

theorem *wls-fresh-subst*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
shows *fresh zs z* (*X* #[*Y* / *y*]-*ys*) =

(((*zs* = *ys* ∧ *z* = *y*) ∨ *fresh zs z X*) ∧ (*fresh ys y X* ∨ *fresh zs z Y*))
using *assms* **by**(*simp add*: *fresh-subst*)

theorem *wls-fresh-vsubst*:
assumes *wls s X*
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*) =
(((*zs* = *ys* ∧ *z* = *y*) ∨ *fresh zs z X*) ∧ (*fresh ys y X* ∨ (*zs* ≠ *ys* ∨ *z* ≠ *y1*)))
using *assms* **by**(*simp add*: *fresh-vsubst*)

theorem *wls-subst-preserves-fresh*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and *fresh zs z X* **and** *fresh zs z Y*
shows *fresh zs z* (*X* #[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *subst-preserves-fresh*)

theorem *wls-substEnv-preserves-freshEnv*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
and *freshEnv zs z rho* **and** *fresh zs z Y* **and** *zs* ≠ *ys* ∨ *z* ≠ *y*
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-preserves-freshEnv*)

theorem *wls-vsubst-preserves-fresh*:
assumes *wls s X*
and *fresh zs z X* **and** *zs* ≠ *ys* ∨ *z* ≠ *y1*
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-preserves-fresh*)

theorem *wls-vsubstEnv-preserves-freshEnv*:
assumes *wlsEnv rho*
and *freshEnv zs z rho* **and** *zs* ≠ *ys* ∨ *z* ∉ {*y*,*y1*}
shows *freshEnv zs z* (*rho* &[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstEnv-preserves-freshEnv*)

theorem *wls-fresh-fresh-subst*[*simp*]:
assumes *wls* (*asSort ys*) *Y* **and** *wls s X*
and *fresh ys y Y*
shows *fresh ys y* (*X* #[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *fresh-fresh-subst*)

theorem *wls-diff-fresh-vsubst*[*simp*]:
assumes *wls s X*
and *y* ≠ *y1*
shows *fresh ys y* (*X* #[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *diff-fresh-vsubst*)

theorem *wls-fresh-subst-E1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and *fresh zs z* (*X* #[*Y* / *y*]-*ys*) **and** *zs* ≠ *ys* ∨ *z* ≠ *y*
shows *fresh zs z X*

using *assms fresh-subst-E1*[*of X Y zs z ys y*] **by** *simp*

theorem *wls-fresh-vsubst-E1*:
assumes *wls s X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*) **and** *zs* ≠ *ys* ∨ *z* ≠ *y*
shows *fresh zs z X*
using *assms fresh-vsubst-E1*[*of X zs z ys y1 y*] **by** *simp*

theorem *wls-fresh-subst-E2*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and *fresh zs z* (*X* #[*Y* / *y*]-*ys*)
shows *fresh ys y X* ∨ *fresh zs z Y*
using *assms fresh-subst-E2*[*of X Y zs z ys y*] **by** *simp*

theorem *wls-fresh-vsubst-E2*:
assumes *wls s X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
shows *fresh ys y X* ∨ *zs* ≠ *ys* ∨ *z* ≠ *y1*
using *assms fresh-vsubst-E2*[*of X zs z ys y1 y*] **by** *simp*

theorem *wls-psubst-cong*[*fundef-cong*]:
assumes *wls s X* **and** *wlsEnv rho* **and** *wlsEnv rho′*
and ⋀ *ys y*. *fresh ys y X* ∨ *rho ys y* = *rho′ ys y*
shows (*X* #[*rho*]) = (*X* #[*rho′*])
using *assms* **by** (*simp add*: *psubst-cong*)

theorem *wls-fresh-psubst-updEnv*:
assumes *wls* (*asSort ys*) *Y* **and** *wls s X* **and** *wlsEnv rho*
and *fresh ys y X*
shows (*X* #[*rho* [*y* ← *Y*]-*ys*]) = (*X* #[*rho*])
using *assms* **by**(*simp add*: *fresh-psubst-updEnv*)

theorem *wls-freshEnv-psubst-ident*[*simp*]:
assumes *wls s X* **and** *wlsEnv rho*
and ⋀ *zs z*. *freshEnv zs z rho* ∨ *fresh zs z X*
shows (*X* #[*rho*]) = *X*
using *assms* **by** *simp*

theorem *wls-fresh-subst-ident*[*simp*]:
assumes *wls* (*asSort ys*) *Y* **and** *wls s X* **and** *fresh ys y X*
shows (*X* #[*Y* / *y*]-*ys*) = *X*
using *assms* **by**(*simp add*: *fresh-subst-ident*)

theorem *wls-substEnv-updEnv-fresh*:
assumes *wls* (*asSort xs*) *X* **and** *wls* (*asSort ys*) *Y* **and** *fresh ys y X*
shows ((*rho* [*x* ← *X*]-*xs*) &[*Y* / *y*]-*ys*) = ((*rho* &[*Y* / *y*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **by**(*simp add*: *substEnv-updEnv-fresh*)

theorem *wls-fresh-substEnv-updEnv*[*simp*]:

assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
and *freshEnv ys y rho*
shows (*rho* &[*Y* / *y*]-*ys*) = (*rho* [*y* ← *Y*]-*ys*)
using *assms* **by** *simp*

theorem *wls-fresh-vsubst-ident*[*simp*]:
assumes *wls s X* **and** *fresh ys y X*
shows (*X* #[*y1* // *y*]-*ys*) = *X*
using *assms* **by**(*simp add*: *fresh-vsubst-ident*)

theorem *wls-vsubstEnv-updEnv-fresh*:
assumes *wls s X* **and** *fresh ys y X*
shows ((*rho* [*x* ← *X*]-*xs*) &[*y1* // *y*]-*ys*) = ((*rho* &[*y1* // *y*]-*ys*) [*x* ← *X*]-*xs*)
using *assms* **by**(*simp add*: *vsubstEnv-updEnv-fresh*)

theorem *wls-fresh-vsubstEnv-updEnv*[*simp*]:
assumes *wlsEnv rho*
and *freshEnv ys y rho*
shows (*rho* &[*y1* // *y*]-*ys*) = (*rho* [*y* ← *Var ys y1*]-*ys*)
using *assms* **by** *simp*

theorem *wls-swap-psubst*:
assumes *wls s X* **and** *wlsEnv rho*
shows ((*X* #[*rho*]) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[*rho* &[*z1* ∧ *z2*]-*zs*])
using *assms* **by**(*simp add*: *swap-psubst*)

theorem *wls-swap-subst*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
shows ((*X* #[*Y* / *y*]-*ys*) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swap-subst*)

theorem *wls-swap-vsubst*:
assumes *wls s X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swap-vsubst*)

theorem *wls-swapEnv-psubstEnv*:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
shows ((*rho* &[*rho'*]) &[*z1* ∧ *z2*]-*zs*) = ((*rho* &[*z1* ∧ *z2*]-*zs*) &[*rho'* &[*z1* ∧ *z2*]-*zs*])
using *assms* **by**(*simp add*: *swapEnv-psubstEnv*)

theorem *wls-swapEnv-substEnv*:
assumes *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*z1* ∧ *z2*]-*zs*) =
    ((*rho* &[*z1* ∧ *z2*]-*zs*) &[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swapEnv-substEnv*)

theorem *wls-swapEnv-vsubstEnv*:
assumes *wlsEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*z1* ∧ *z2*]-*zs*) =
((*rho* &[*z1* ∧ *z2*]-*zs*) &[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swapEnv-vsubstEnv*)

theorem *wls-psubst-compose*:
assumes *wls s X* **and** *wlsEnv rho* **and** *wlsEnv rho′*
shows ((*X* #[*rho*]) #[*rho′*]) = (*X* #[(*rho* &[*rho′*])])
using *assms* **by**(*simp add*: *psubst-compose*)

theorem *wls-psubstEnv-compose*:
assumes *wlsEnv rho* **and** *wlsEnv rho′* **and** *wlsEnv rho″*
shows ((*rho* &[*rho′*]) &[*rho″*]) = (*rho* &[(*rho′* &[*rho″*])])
using *assms* **by**(*simp add*: *psubstEnv-compose*)

theorem *wls-psubst-subst-compose*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*
shows ((*X* #[*Y* / *y*]-*ys*) #[*rho*]) = (*X* #[(*rho* [*y* ← (*Y* #[*rho*])]-*ys*)])
using *assms* **by**(*simp add*: *psubst-subst-compose*)

theorem *wls-psubst-subst-compose-freshEnv*:
assumes *wlsEnv rho* **and** *wls s X* **and** *wls* (*asSort ys*) *Y*
and *freshEnv ys y rho*
shows ((*X* #[*Y* / *y*]-*ys*) #[*rho*]) = ((*X* #[*rho*]) #[(*Y* #[*rho*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *psubst-subst-compose-freshEnv*)

theorem *wls-psubstEnv-substEnv-compose-freshEnv*:
assumes *wlsEnv rho* **and** *wlsEnv rho′* **and** *wls* (*asSort ys*) *Y*
assumes *freshEnv ys y rho′*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*rho′*]) = ((*rho* &[*rho′*]) &[(*Y* #[*rho′*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *psubstEnv-substEnv-compose-freshEnv*)

theorem *wls-psubstEnv-substEnv-compose*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho′*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*rho′*]) = (*rho* &[(*rho′* [*y* ← (*Y* #[*rho′*])]-*ys*)])
using *assms* **by**(*simp add*: *psubstEnv-substEnv-compose*)

theorem *wls-psubst-vsubst-compose*:
assumes *wls s X* **and** *wlsEnv rho*
shows ((*X* #[*y1* // *y*]-*ys*) #[*rho*]) = (*X* #[(*rho* [*y* ← ((*Var ys y1*) #[*rho*])]-*ys*)])
using *assms* **by**(*simp add*: *psubst-vsubst-compose*)

theorem *wls-psubstEnv-vsubstEnv-compose*:
assumes *wlsEnv rho* **and** *wlsEnv rho′*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*rho′*]) = (*rho* &[(*rho′* [*y* ← ((*Var ys y1*) #[*rho′*])]-*ys*)])
using *assms* **by**(*simp add*: *psubstEnv-vsubstEnv-compose*)

theorem *wls-subst-psubst-compose*:

assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*
shows ((*X* #[*rho*]) #[*Y* / *y*]-*ys*) = (*X* #[(*rho* &[*Y* / *y*]-*ys*)])
using *assms* **by**(*simp add*: *subst-psubst-compose*)

theorem *wls-substEnv-psubstEnv-compose*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho'*
shows ((*rho* &[*rho'*]) &[*Y* / *y*]-*ys*) = (*rho* &[(*rho'* &[*Y* / *y*]-*ys*)])
using *assms* **by**(*simp add*: *substEnv-psubstEnv-compose*)

theorem *wls-vsubst-psubst-compose*:
assumes *wls s X* **and** *wlsEnv rho*
shows ((*X* #[*rho*]) #[*y1* // *y*]-*ys*) = (*X* #[(*rho* &[*y1* // *y*]-*ys*)])
using *assms* **by**(*simp add*: *vsubst-psubst-compose*)

theorem *wls-vsubstEnv-psubstEnv-compose*:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
shows ((*rho* &[*rho'*]) &[*y1* // *y*]-*ys*) = (*rho* &[(*rho'* &[*y1* // *y*]-*ys*)])
using *assms* **by**(*simp add*: *vsubstEnv-psubstEnv-compose*)

theorem *wls-subst-compose1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y1* **and** *wls* (*asSort ys*) *Y2*
shows ((*X* #[*Y1* / *y*]-*ys*) #[*Y2* / *y*]-*ys*) = (*X* #[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *subst-compose1*)

theorem *wls-substEnv-compose1*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y1* **and** *wls* (*asSort ys*) *Y2*
shows ((*rho* &[*Y1* / *y*]-*ys*) &[*Y2* / *y*]-*ys*) = (*rho* &[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-compose1*)

theorem *wls-subst-vsubst-compose1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** $y \neq y1$
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y*]-*ys*) = (*X* #[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *subst-vsubst-compose1*)

theorem *wls-substEnv-vsubstEnv-compose1*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** $y \neq y1$
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Y* / *y*]-*ys*) = (*rho* &[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-vsubstEnv-compose1*)

theorem *wls-vsubst-subst-compose1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
shows ((*X* #[*Y* / *y*]-*ys*) #[*y1* // *y*]-*ys*) = (*X* #[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-subst-compose1*)

theorem *wls-vsubstEnv-substEnv-compose1*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*y1* // *y*]-*ys*) = (*rho* &[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstEnv-substEnv-compose1*)

theorem *wls-vsubst-compose1*:
assumes *wls s X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y*]-*ys*) = (*X* #[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-compose1*)

theorem *wls-vsubstEnv-compose1*:
assumes *wlsEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*y2* // *y*]-*ys*) = (*rho* &[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstEnv-compose1*)

theorem *wls-subst-compose2*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*X* #[*Y* / *y*]-*ys*) #[*Z* / *z*]-*zs*) = ((*X* #[*Z* / *z*]-*zs*) #[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *subst-compose2*)

theorem *wls-substEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*Z* / *z*]-*zs*) = ((*rho* &[*Z* / *z*]-*zs*) &[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-compose2*)

theorem *wls-subst-vsubst-compose2*:
assumes *wls s X* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Z* / *z*]-*zs*) = ((*X* #[*Z* / *z*]-*zs*) #[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *subst-vsubst-compose2*)

theorem *wls-substEnv-vsubstEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Z* / *z*]-*zs*) = ((*rho* &[*Z* / *z*]-*zs*) &[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substEnv-vsubstEnv-compose2*)

theorem *wls-vsubst-subst-compose2*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*X* #[*Y* / *y*]-*ys*) #[*z1* // *z*]-*zs*) = ((*X* #[*z1* // *z*]-*zs*) #[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-subst-compose2*)

theorem *wls-vsubstEnv-substEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*rho* &[*Y* / *y*]-*ys*) &[*z1* // *z*]-*zs*) = ((*rho* &[*z1* // *z*]-*zs*) &[(*Y* #[*z1* //

z]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstEnv-substEnv-compose2*)

theorem *wls-vsubst-compose2*:
assumes *wls s X*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*X* #[*y1* // *y*]-*ys*) #[*z1* // *z*]-*zs*) = ((*X* #[*z1* // *z*]-*zs*) #[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-compose2*)

theorem *wls-vsubstEnv-compose2*:
assumes *wlsEnv rho*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*rho* &[*y1* // *y*]-*ys*) &[*z1* // *z*]-*zs*) =
((*rho* &[*z1* // *z*]-*zs*) &[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstEnv-compose2*)

7.9.3 Properties specific to variable-for-variable substitution

theorem *wls-vsubst-ident*[*simp*]:
assumes *wls s X*
shows (*X* #[*z* // *z*]-*zs*) = *X*
using *assms* **by**(*simp add*: *vsubst-ident*)

theorem *wls-subst-ident*[*simp*]:
assumes *wls s X*
shows (*X* #[(*Var zs z*) / *z*]-*zs*) = *X*
using *assms* **by** *simp*

theorem *wls-vsubst-eq-swap*:
assumes *wls s X* **and** *y1* = *y2* ∨ *fresh ys y1 X*
shows (*X* #[*y1* // *y2*]-*ys*) = (*X* #[*y1* ∧ *y2*]-*ys*)
using *assms* **by**(*simp add*: *vsubst-eq-swap*)

theorem *wls-skel-vsubst*:
assumes *wls s X*
shows *skel* (*X* #[*y1* // *y2*]-*ys*) = *skel X*
using *assms* **by**(*simp add*: *skel-vsubst*)

theorem *wls-subst-vsubst-trans*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** *fresh ys y1 X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y1*]-*ys*) = (*X* #[*Y* / *y*]-*ys*)
using *assms* **by** (*simp add*: *subst-vsubst-trans*)

theorem *wls-vsubst-trans*:
assumes *wls s X* **and** *fresh ys y1 X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y1*]-*ys*) = (*X* #[*y2* // *y*]-*ys*)
using *assms* **by** (*simp add*: *vsubst-trans*)

theorem *wls-vsubst-commute*:
assumes *wls s X*
and $xs \neq xs' \vee \{x,y\} \cap \{x',y'\} = \{\}$ **and** *fresh xs x X* **and** *fresh xs' x' X*
shows $((X \ \#[x \ // \ y]\text{-}xs) \ \#[x' \ // \ y']\text{-}xs') = ((X \ \#[x' \ // \ y']\text{-}xs') \ \#[x \ // \ y]\text{-}xs)$
using *assms* **by**(*simp add*: *vsubst-commute*)

theorem *wls-induct*[*case-names Var Op Abs*]:
assumes
Var: $\bigwedge$ *xs x. phi* (*asSort xs*) (*Var xs x*) **and**
Op:
$\bigwedge$ *delta inp binp.*
⟦*wlsInp delta inp*; *wlsBinp delta binp*;
liftAll2 phi (*arOf delta*) *inp*; *liftAll2 phiAbs* (*barOf delta*) *binp*⟧
$\Longrightarrow$ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *s xs x X.*
⟦*isInBar* (*xs*,*s*); *wls s X*;
$\bigwedge$ *Y*. $(X,Y) \in$ *swapped* $\Longrightarrow$ *phi s Y*;
$\bigwedge$ *ys y1 y2. phi s* (*X* #[*y1* // *y2*]-*ys*);
$\bigwedge$ *Y*. ⟦*wls s Y*; *skel Y* = *skel X*⟧ $\Longrightarrow$ *phi s Y*⟧
$\Longrightarrow$ *phiAbs* (*xs*,*s*) (*Abs xs x X*)
shows
(*wls s X* $\longrightarrow$ *phi s X*) $\wedge$
(*wlsAbs* (*xs*,*s'*) *A* $\longrightarrow$ *phiAbs* (*xs*,*s'*) *A*)
apply(*induction rule*: *wls-templateInduct*
[*of* λ*s. swapped* $\cup$ {(*X*, *X* #[*y1* // *y2*]-*ys*)| *X ys y1 y2*. *True*}
$\cup$ {(*X*,*Y*). *wls s Y* $\wedge$ *skel Y* = *skel X*}])
by (*auto simp add*: *assms swapped-preserves-wls swapped-skel wls-skel-vsubst*
intro!: *Abs*)

theorem *wls-Abs-vsubst-all-aux*:
assumes *wls s X* **and** *wls s X'*
shows
(*Abs xs x X* = *Abs xs x' X'*) =
($\forall$ *y*. (*y* = *x* $\vee$ *fresh xs y X*) $\wedge$ (*y* = *x'* $\vee$ *fresh xs y X'*) $\longrightarrow$
(*X* #[*y* // *x*]-*xs*) = (*X'* #[*y* // *x'*]-*xs*))
using *assms wls-Abs-swap-all* **by** (*simp add*: *wls-vsubst-eq-swap*)

theorem *wls-Abs-vsubst-ex*:
assumes *wls s X* **and** *wls s X'*
shows
(*Abs xs x X* = *Abs xs x' X'*) =
($\exists$ *y*. *y* $\notin$ {*x*,*x'*} $\wedge$ *fresh xs y X* $\wedge$ *fresh xs y X'* $\wedge$
(*X* #[*y* // *x*]-*xs*) = (*X'* #[*y* // *x'*]-*xs*))
proof−
let *?phi* = λ *f y*. *y* $\notin$ {*x*,*x'*} $\wedge$ *fresh xs y X* $\wedge$ *fresh xs y X'*
$\wedge$ (*f xs y x X*) = (*f xs y x' X'*)

{**assume** *Abs xs x X* = *Abs xs x′ X′*
then obtain *y* **where** *?phi swap y* **using** *assms wls-Abs-swap-ex* **by** *auto*
hence *?phi* (λ *xs y x X*. (*X* #[*y* // *x*]-*xs*)) *y*
using *assms* **by**(*simp add*: *wls-vsubst-eq-swap*)
hence ∃ *y*. *?phi* (λ *xs y x X*. (*X* #[*y* // *x*]-*xs*)) *y* **by** *auto*
}
moreover
{**fix** *y* **assume** *?phi* (λ *xs y x X*. (*X* #[*y* // *x*]-*xs*)) *y*
hence *?phi swap y* **using** *assms* **by**(*auto simp add*: *wls-vsubst-eq-swap*)
hence *Abs xs x X* = *Abs xs x′ X′* **using** *assms wls-Abs-swap-ex* **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

theorem *wls-Abs-vsubst-all*:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs x′ X′*) =
(∀ *y*. (*X* #[*y* // *x*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*))
proof(*rule iffI*, *clarify*)
assume ∀ *y*. (*X* #[*y* // *x*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*)
thus *Abs xs x X* = *Abs xs x′ X′*
using *assms* **by**(*auto simp add*: *wls-Abs-vsubst-all-aux*)
next
fix *y*
assume *Abs xs x X* = *Abs xs x′ X′*
then obtain *z* **where** *z-fresh*: *fresh xs z X* ∧ *fresh xs z X′*
and (*X* #[*z* // *x*]-*xs*) = (*X′* #[*z* // *x′*]-*xs*)
using *assms* **by**(*auto simp add*: *wls-Abs-vsubst-ex*)
hence ((*X* #[*z* // *x*]-*xs*) #[*y* // *z*]-*xs*) = ((*X′* #[*z* // *x′*]-*xs*) #[*y* // *z*]-*xs*) **by** *simp*
thus (*X* #[*y* // *x*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*)
using *assms z-fresh wls-vsubst-trans* **by** *auto*
qed

theorem *wls-Abs-subst-all*:
assumes *wls s X* **and** *wls s X′*
shows
(*Abs xs x X* = *Abs xs x′ X′*) =
(∀ *Y*. *wls* (*asSort xs*) *Y* ⟶ (*X* #[*Y* / *x*]-*xs*) = (*X′* #[*Y* / *x′*]-*xs*))
proof(*rule iffI*, *clarify*)
assume ∀ *Y*. *wls* (*asSort xs*) *Y* ⟶ (*X* #[*Y* / *x*]-*xs*) = (*X′* #[*Y* / *x′*]-*xs*)
hence ∀ *y*. (*X* #[*y* // *x*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*)
unfolding *vsubst-def* **by** *simp*
thus *Abs xs x X* = *Abs xs x′ X′*
using *assms wls-Abs-vsubst-all* **by** *auto*
next
fix *Y* **assume** *Y*: *wls* (*asSort xs*) *Y*
assume *Abs xs x X* = *Abs xs x′ X′*

then obtain *z* **where** *z-fresh*: *fresh xs z X* ∧ *fresh xs z X'*
and (*X* #[*z* // *x*]-*xs*) = (*X'* #[*z* // *x'*]-*xs*)
using *assms* **by**(*auto simp add*: *wls-Abs-vsubst-ex*)
hence ((*X* #[*z* // *x*]-*xs*) #[*Y* / *z*]-*xs*) = ((*X'* #[*z* // *x'*]-*xs*) #[*Y* / *z*]-*xs*) **by** *simp*
thus (*X* #[*Y* / *x*]-*xs*) = (*X'* #[*Y* / *x'*]-*xs*)
using *assms z-fresh Y wls-subst-vsubst-trans* **by** *auto*
qed

lemma *Abs-inj-fresh*[*simp*]:
assumes *X*: *wls s X* **and** *X'*: *wls s X'*
and *fresh-X*: *fresh ys x X* **and** *fresh-X'*: *fresh ys x' X'*
and *eq*: *Abs ys x X* = *Abs ys x' X'*
shows *X* = *X'*
proof−
obtain *z* **where** (*X* #[*z* // *x*]-*ys*) = (*X'* #[*z* // *x'*]-*ys*)
using *X X' eq* **by**(*auto simp add*: *wls-Abs-vsubst-ex*)
thus *?thesis* **using** *X X' fresh-X fresh-X'* **by** *simp*
qed

theorem *wls-Abs-vsubst-cong*:
assumes *wls s X* **and** *wls s X'*
and *fresh xs y X* **and** *fresh xs y X'* **and** (*X* #[*y* // *x*]-*xs*) = (*X'* #[*y* // *x'*]-*xs*)
shows *Abs xs x X* = *Abs xs x' X'*
using *assms* **by** (*intro wls-Abs-swap-cong*) (*auto simp*: *wls-vsubst-eq-swap*)

theorem *wls-Abs-vsubst-fresh*[*simp*]:
assumes *wls s X* **and** *fresh xs x' X*
shows *Abs xs x'* (*X* #[*x'* // *x*]-*xs*) = *Abs xs x X*
using *assms* **by**(*simp add*: *wls-vsubst-eq-swap*)

theorem *wls-Abs-subst-Var-fresh*[*simp*]:
assumes *wls s X* **and** *fresh xs x' X*
shows *Abs xs x'* (*subst xs* (*Var xs x'*) *x X*) = *Abs xs x X*
using *assms wls-Abs-vsubst-fresh* **unfolding** *vsubst-def* **by** *simp*

theorem *wls-Abs-vsubst-congSTR*:
assumes *wls s X* **and** *wls s X'*
and *y* = *x* ∨ *fresh xs y X y* = *x'* ∨ *fresh xs y X'*
and (*X* #[*y* // *x*]-*xs*) = (*X'* #[*y* // *x'*]-*xs*)
shows *Abs xs x X* = *Abs xs x' X'*
by (*metis assms wls-Abs-vsubst-fresh wls-vsubst-ident*)

7.9.4 Abstraction versions of the properties

theorem *wls-psubstAbs-idEnv*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ (*A* $[*idEnv*]) = *A*
by *simp*

theorem *wls-freshAbs-psubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
shows
freshAbs zs z (*A* $[*rho*]) =
(∀ *ys y*. *freshAbs ys y A* ∨ *freshImEnvAt zs z rho ys y*)
using *assms* **by**(*simp add*: *freshAbs-psubstAbs*)

theorem *wls-freshAbs-psubstAbs-E1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and *rho ys y* = *None* **and** *freshAbs zs z* (*A* $[*rho*])
shows *freshAbs ys y A* ∨ (*ys* ≠ *zs* ∨ *y* ≠ *z*)
using *assms freshAbs-psubstAbs-E1*[*of A rho ys y zs z*] **by** *simp*

theorem *wls-freshAbs-psubstAbs-E2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and *rho ys y* = *Some Y* **and** *freshAbs zs z* (*A* $[*rho*])
shows *freshAbs ys y A* ∨ *fresh zs z Y*
using *assms freshAbs-psubstAbs-E2*[*of A rho ys y Y zs z*] **by** *simp*

theorem *wls-freshAbs-psubstAbs-I1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and *freshAbs zs z A* **and** *freshEnv zs z rho*
shows *freshAbs zs z* (*A* $[*rho*])
using *assms* **by**(*simp add*: *freshAbs-psubstAbs-I1*)

theorem *wls-freshAbs-psubstAbs-I*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and *rho zs z* = *None* ⟹ *freshAbs zs z A* **and**
⋀ *ys y Y*. *rho ys y* = *Some Y* ⟹ *freshAbs ys y A* ∨ *fresh zs z Y*
shows *freshAbs zs z* (*A* $[*rho*])
using *assms* **by**(*simp add*: *freshAbs-psubstAbs-I*)

theorem *wls-freshAbs-substAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
shows *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*) =
(((*zs* = *ys* ∧ *z* = *y*) ∨ *freshAbs zs z A*) ∧ (*freshAbs ys y A* ∨ *fresh zs z Y*))
using *assms* **by**(*simp add*: *freshAbs-substAbs*)

theorem *wls-freshAbs-vsubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*) =
(((*zs* = *ys* ∧ *z* = *y*) ∨ *freshAbs zs z A*) ∧
(*freshAbs ys y A* ∨ (*zs* ≠ *ys* ∨ *z* ≠ *y1*)))
using *assms* **by**(*simp add*: *freshAbs-vsubstAbs*)

theorem *wls-substAbs-preserves-freshAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*

and *freshAbs zs z A* **and** *fresh zs z Y*
shows *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-preserves-freshAbs*)

theorem *wls-vsubstAbs-preserves-freshAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs zs z A* **and** $zs \neq ys \vee z \neq y1$
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-preserves-freshAbs*)

theorem *wls-fresh-freshAbs-substAbs*[*simp*]:
assumes *wls* (*asSort ys*) *Y* **and** *wlsAbs* (*us*,*s*) *A*
and *fresh ys y Y*
shows *freshAbs ys y* (*A* $[*Y* / *y*]-*ys*)
using *assms* **by** *simp*

theorem *wls-diff-freshAbs-vsubstAbs*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
and $y \neq y1$
shows *freshAbs ys y* (*A* $[*y1* // *y*]-*ys*)
using *assms* **by** *simp*

theorem *wls-freshAbs-substAbs-E1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
and *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*) **and** $z \neq y \vee zs \neq ys$
shows *freshAbs zs z A*
using *assms freshAbs-substAbs-E1*[*of A Y zs z ys y*] **by** *auto*

theorem *wls-freshAbs-vsubstAbs-E1*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*) **and** $z \neq y \vee zs \neq ys$
shows *freshAbs zs z A*
using *assms freshAbs-vsubstAbs-E1*[*of A zs z ys y1 y*] **by** *auto*

theorem *wls-freshAbs-substAbs-E2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
and *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*)
shows *freshAbs ys y A* ∨ *fresh zs z Y*
using *assms freshAbs-substAbs-E2*[*of A Y zs z ys*] **by** *simp*

theorem *wls-freshAbs-vsubstAbs-E2*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
shows *freshAbs ys y A* ∨ $zs \neq ys \vee z \neq y1$
using *assms freshAbs-vsubstAbs-E2*[*of A zs z ys y1 y*] **by** *simp*

theorem *wls-psubstAbs-cong*[*fundef-cong*]:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho* **and** *wlsEnv rho'*
and ⋀ *ys y*. *freshAbs ys y A* ∨ *rho ys y* = *rho' ys y*

shows (*A* $[*rho*]) = (*A* $[*rho'*])
using *assms* **by**(*simp add*: *psubstAbs-cong*)

theorem *wls-freshAbs-psubstAbs-updEnv*:
assumes *wls* (*asSort xs*) *X* **and** *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and *freshAbs xs x A*
shows (*A* $[*rho* [*x* ← *X*]-*xs*]) = (*A* $[*rho*])
using *assms* **by**(*simp add*: *freshAbs-psubstAbs-updEnv*)

lemma *wls-freshEnv-psubstAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
and ⋀ *zs z*. *freshEnv zs z rho* ∨ *freshAbs zs z A*
shows (*A* $[*rho*]) = *A*
using *assms* **by** *simp*

theorem *wls-freshAbs-substAbs-ident*[*simp*]:
assumes *wls* (*asSort xs*) *X* **and** *wlsAbs* (*us*,*s*) *A* **and** *freshAbs xs x A*
shows (*A* $[*X* / *x*]-*xs*) = *A*
using *assms* **by** *simp*

theorem *wls-substAbs-Abs*[*simp*]:
assumes *wls s X* **and** *wls* (*asSort xs*) *Y*
shows ((*Abs xs x X*) $[*Y* / *x*]-*xs*) = *Abs xs x X*
using *assms* **by** *simp*

theorem *wls-freshAbs-vsubstAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A* **and** *freshAbs xs x A*
shows (*A* $[*x1* // *x*]-*xs*) = *A*
using *assms* **by**(*simp add*: *freshAbs-vsubstAbs-ident*)

theorem *wls-swapAbs-psubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
shows ((*A* $[*rho*]) $[*z1* ∧ *z2*]-*zs*) = ((*A* $[*z1* ∧ *z2*]-*zs*) $[*rho* &[*z1* ∧ *z2*]-*zs*])
using *assms* **by**(*simp add*: *swapAbs-psubstAbs*)

theorem *wls-swapAbs-substAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
shows ((*A* $[*Y* / *y*]-*ys*) $[*z1* ∧ *z2*]-*zs*) =
    ((*A* $[*z1* ∧ *z2*]-*zs*) $[(*Y* #[*z1* ∧ *z2*]-*zs*) / (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swapAbs-substAbs*)

theorem *wls-swapAbs-vsubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*z1* ∧ *z2*]-*zs*) =
    ((*A* $[*z1* ∧ *z2*]-*zs*) $[(*y1* @*ys*[*z1* ∧ *z2*]-*zs*) // (*y* @*ys*[*z1* ∧ *z2*]-*zs*)]-*ys*)
using *assms* **by**(*simp add*: *swapAbs-vsubstAbs*)

theorem *wls-psubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho* **and** *wlsEnv rho'*

shows ((*A* $[*rho*]) $[*rho'*]) = (*A* $[(*rho* &[*rho'*])])
using *assms* **by**(*simp add*: *psubstAbs-compose*)

theorem *wls-psubstAbs-substAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*
shows ((*A* $[*Y* / *y*]-*ys*) $[*rho*]) = (*A* $[(*rho* [*y* ← (*Y* #[*rho*])]-*ys*)])
using *assms* **by**(*simp add*: *psubstAbs-substAbs-compose*)

theorem *wls-psubstAbs-substAbs-compose-freshEnv*:
assumes *wlsEnv rho* **and** *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
assumes *freshEnv ys y rho*
shows ((*A* $[*Y* / *y*]-*ys*) $[*rho*]) = ((*A* $[*rho*]) $[(*Y* #[*rho*]) / *y*]-*ys*)
using *assms* **by** (*simp add*: *psubstAbs-substAbs-compose-freshEnv*)

theorem *wls-psubstAbs-vsubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
shows ((*A* $[*y1* // *y*]-*ys*) $[*rho*]) = (*A* $[(*rho* [*y* ← ((*Var ys y1*) #[*rho*])]-*ys*)])
using *assms* **by**(*simp add*: *psubstAbs-vsubstAbs-compose*)

theorem *wls-substAbs-psubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*
shows ((*A* $[*rho*]) $[*Y* / *y*]-*ys*) = (*A* $[(*rho* &[*Y* / *y*]-*ys*)])
using *assms* **by**(*simp add*: *substAbs-psubstAbs-compose*)

theorem *wls-vsubstAbs-psubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
shows ((*A* $[*rho*]) $[*y1* // *y*]-*ys*) = (*A* $[(*rho* &[*y1* // *y*]-*ys*)])
using *assms* **by**(*simp add*: *vsubstAbs-psubstAbs-compose*)

theorem *wls-substAbs-compose1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y1* **and** *wls* (*asSort ys*) *Y2*
shows ((*A* $[*Y1* / *y*]-*ys*) $[*Y2* / *y*]-*ys*) = (*A* $[(*Y1* #[*Y2* / *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-compose1*)

theorem *wls-substAbs-vsubstAbs-compose1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** $y \neq y1$
shows ((*A* $[*y1* // *y*]-*ys*) $[*Y* / *y*]-*ys*) = (*A* $[*y1* // *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-vsubstAbs-compose1*)

theorem *wls-vsubstAbs-substAbs-compose1*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
shows ((*A* $[*Y* / *y*]-*ys*) $[*y1* // *y*]-*ys*) = (*A* $[(*Y* #[*y1* // *y*]-*ys*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-substAbs-compose1*)

theorem *wls-vsubstAbs-compose1*:
assumes *wlsAbs* (*us*,*s*) *A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*y2* // *y*]-*ys*) = (*A* $[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-compose1*)

theorem *wls-substAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*A* $[*Y* / *y*]-*ys*) $[*Z* / *z*]-*zs*) = ((*A* $[*Z* / *z*]-*zs*) $[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-compose2*)

theorem *wls-substAbs-vsubstAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort zs*) *Z*
and *ys* ≠ *zs* ∨ *y* ≠ *z* **and** *fresh*: *fresh ys y Z*
shows ((*A* $[*y1* // *y*]-*ys*) $[*Z* / *z*]-*zs*) = ((*A* $[*Z* / *z*]-*zs*) $[((*Var ys y1*) #[*Z* / *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-vsubstAbs-compose2*)

theorem *wls-vsubstAbs-substAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*A* $[*Y* / *y*]-*ys*) $[*z1* // *z*]-*zs*) = ((*A* $[*z1* // *z*]-*zs*) $[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-substAbs-compose2*)

theorem *wls-vsubstAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}
shows ((*A* $[*y1* // *y*]-*ys*) $[*z1* // *z*]-*zs*) = ((*A* $[*z1* // *z*]-*zs*) $[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-compose2*)

theorem *wls-vsubstAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows (*A* $[*z* // *z*]-*zs*) = *A*
using *assms* **by**(*simp add*: *vsubstAbs-ident*)

theorem *wls-substAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows (*A* $[(*Var zs z*) / *z*]-*zs*) = *A*
using *assms* **by** *simp*

theorem *wls-vsubstAbs-eq-swapAbs*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *y1* = *y2* ∨ *freshAbs ys y1 A*
shows (*A* $[*y1* // *y2*]-*ys*) = (*A* $[*y1* ∧ *y2*]-*ys*)
using *assms vsubstAll-swapAll*[*of Par* [*y1*, *y2*] [] [] [] - - *A*]
unfolding *goodPar-def* **by** *auto*

theorem *wls-skelAbs-vsubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
shows *skelAbs* (*A* $[*y1* // *y2*]-*ys*) = *skelAbs A*
using *assms* **by**(*simp add*: *skelAbs-vsubstAbs*)

theorem *wls-substAbs-vsubstAbs-trans*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *freshAbs ys y1 A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*Y* / *y1*]-*ys*) = (*A* $[*Y* / *y*]-*ys*)
using *assms* **by**(*simp add*: *substAbs-vsubstAbs-trans*)

theorem *wls-vsubstAbs-trans*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *freshAbs ys y1 A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*y2* // *y1*]-*ys*) = (*A* $[*y2* // *y*]-*ys*)
using *assms* **by**(*simp add*: *vsubstAbs-trans*)

theorem *wls-vsubstAbs-commute*:
assumes *wlsAbs* (*us*,*s*) *A*
and $xs \neq xs' \vee \{x,y\} \cap \{x',y'\} = \{\}$ **and** *freshAbs xs x A* **and** *freshAbs xs' x' A*
shows ((*A* $[*x* // *y*]-*xs*) $[*x'* // *y'*]-*xs'*) = ((*A* $[*x'* // *y'*]-*xs'*) $[*x* // *y*]-*xs*)
proof−
  have *freshAbs xs' x'* (*A* $[*x* // *y*]-*xs*)
  using *assms* **by**(*auto simp*: *vsubstAbs-preserves-freshAbs*)
  moreover have *freshAbs xs x* (*A* $[*x'* // *y'*]-*xs'*)
  using *assms* **by**(*auto simp*: *vsubstAbs-preserves-freshAbs*)
  ultimately show *?thesis* **using** *assms*
  by (*auto simp*: *vsubstAbs-eq-swapAbs intro*!: *wls-swapAbs-commute*)
qed

lemmas *wls-psubstAll-freshAll-otherSimps* =
wls-psubst-idEnv wls-psubstEnv-idEnv-id wls-psubstAbs-idEnv
wls-freshEnv-psubst-ident wls-freshEnv-psubstAbs-ident

lemmas *wls-substAll-freshAll-otherSimps* =
wls-fresh-fresh-subst wls-fresh-subst-ident wls-fresh-substEnv-updEnv wls-subst-ident
wls-fresh-freshAbs-substAbs wls-freshAbs-substAbs-ident wls-substAbs-ident
wls-Abs-subst-Var-fresh

lemmas *wls-vsubstAll-freshAll-otherSimps* =
wls-diff-fresh-vsubst wls-fresh-vsubst-ident wls-fresh-vsubstEnv-updEnv wls-vsubst-ident
wls-diff-freshAbs-vsubstAbs wls-freshAbs-vsubstAbs-ident wls-vsubstAbs-ident
wls-Abs-vsubst-fresh

lemmas *wls-allOpers-otherSimps* =
wls-swapAll-freshAll-otherSimps
wls-psubstAll-freshAll-otherSimps
wls-substAll-freshAll-otherSimps
wls-vsubstAll-freshAll-otherSimps

7.10 Operators for down-casting and case-analyzing well-sorted items

The features developed here may occasionally turn out more convenient than obtaining the desired effect by hand, via the corresponding nchotomies.

E.g., when we want to perform the case-analysis uniformly, as part of a function definition, the operators defined in the subsection save some tedious definitions and proofs pertaining to Hilbert choice.

7.10.1 For terms

definition *isVar* **where**
isVar s (*X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term*) ==
∃ *xs x*. *s* = *asSort xs* ∧ *X* = *Var xs x*

definition *castVar* **where**
castVar s (*X* :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term*) ==
SOME xs-x. *s* = *asSort* (*fst xs-x*) ∧ *X* = *Var* (*fst xs-x*) (*snd xs-x*)

definition *isOp* **where**
isOp s X ≡
∃ *delta inp binp*.
wlsInp delta inp ∧ *wlsBinp delta binp* ∧ *s* = *stOf delta* ∧ *X* = *Op delta inp binp*

definition *castOp* **where**
castOp s X ≡
SOME delta-inp-binp.
wlsInp (*fst3 delta-inp-binp*) (*snd3 delta-inp-binp*) ∧
wlsBinp (*fst3 delta-inp-binp*) (*trd3 delta-inp-binp*) ∧
s = *stOf* (*fst3 delta-inp-binp*) ∧
X = *Op* (*fst3 delta-inp-binp*) (*snd3 delta-inp-binp*) (*trd3 delta-inp-binp*)

definition *sortTermCase* **where**
sortTermCase fVar fOp s X ≡
if isVar s X then fVar (*fst* (*castVar s X*)) (*snd* (*castVar s X*))
else if isOp s X then fOp (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*)) (*trd3* (*castOp s X*))
else undefined

lemma *isVar-asSort-Var*[*simp*]:
isVar (*asSort xs*) (*Var xs x*)
unfolding *isVar-def* **by** *auto*

lemma *not-isVar-Op*[*simp*]:
¬ *isVar s* (*Op delta inp binp*)
unfolding *isVar-def* **by** *auto*

lemma *isVar-imp-wls*:
isVar s X ⟹ *wls s X*
unfolding *isVar-def* **by** *auto*

lemmas *isVar-simps* =

isVar-asSort-Var not-isVar-Op

lemma *castVar-asSort-Var*[*simp*]:
castVar (*asSort xs*) (*Var xs x*) = (*xs*,*x*)
unfolding *castVar-def* **by** (*rule some-equality*) *auto*

lemma *isVar-castVar*:
assumes *isVar s X*
shows *asSort* (*fst* (*castVar s X*)) = *s* ∧
    Var (*fst* (*castVar s X*)) (*snd* (*castVar s X*)) = *X*
using *assms isVar-def* **by** *auto*

lemma *asSort-castVar*[*simp*]:
isVar s X ⟹ *asSort* (*fst* (*castVar s X*)) = *s*
using *isVar-castVar* **by** *auto*

lemma *Var-castVar*[*simp*]:
isVar s X ⟹ *Var* (*fst* (*castVar s X*)) (*snd* (*castVar s X*)) = *X*
using *isVar-castVar* **by** *auto*

lemma *castVar-inj*[*simp*]:
assumes ∗: *isVar s X* **and** ∗∗: *isVar s′ X′*
shows (*castVar s X* = *castVar s′ X′*) = (*s* = *s′* ∧ *X* = *X′*)
using *assms Var-castVar asSort-castVar* **by** *fastforce*

lemmas *castVar-simps* =
castVar-asSort-Var
asSort-castVar Var-castVar castVar-inj

lemma *isOp-stOf-Op*[*simp*]:
⟦*wlsInp delta inp*; *wlsBinp delta binp*⟧
⟹ *isOp* (*stOf delta*) (*Op delta inp binp*)
unfolding *isOp-def* **by** *auto*

lemma *not-isOp-Var*[*simp*]:
¬ *isOp s* (*Var xs X*)
unfolding *isOp-def* **by** *auto*

lemma *isOp-imp-wls*:
isOp s X ⟹ *wls s X*
unfolding *isOp-def* **by** *auto*

lemmas *isOp-simps* =
isOp-stOf-Op not-isOp-Var

lemma *castOp-stOf-Op*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*

shows *castOp* (*stOf delta*) (*Op delta inp binp*) = (*delta*,*inp*,*binp*)
using *assms* **unfolding** *castOp-def* **by** (*intro some-equality*) *auto*

lemma *isOp-castOp*:
assumes *isOp s X*
shows *wlsInp* (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*)) ∧
wlsBinp (*fst3* (*castOp s X*)) (*trd3* (*castOp s X*)) ∧
stOf (*fst3* (*castOp s X*)) = *s* ∧
Op (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*)) (*trd3* (*castOp s X*)) = *X*
proof−
let *?phi* = λ *DIB*. *wlsInp* (*fst3 DIB*) (*snd3 DIB*) ∧
wlsBinp (*fst3 DIB*) (*trd3 DIB*) ∧
s = *stOf* (*fst3 DIB*) ∧
X = *Op* (*fst3 DIB*) (*snd3 DIB*) (*trd3 DIB*)
obtain *delta inp binp* **where** *?phi* (*delta*,*inp*,*binp*)
using *assms* **unfolding** *isOp-def* **by** *auto*
hence *?phi* (*castOp s X*) **using** *someI*[*of ?phi*] **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *wlsInp-castOp*[*simp*]:
isOp s X ⟹ *wlsInp* (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*))
using *isOp-castOp* **by** *auto*

lemma *wlsBinp-castOp*[*simp*]:
isOp s X ⟹ *wlsBinp* (*fst3* (*castOp s X*)) (*trd3* (*castOp s X*))
using *isOp-castOp* **by** *auto*

lemma *stOf-castOp*[*simp*]:
isOp s X ⟹ *stOf* (*fst3* (*castOp s X*)) = *s*
using *isOp-castOp* **by** *auto*

lemma *Op-castOp*[*simp*]:
isOp s X ⟹
Op (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*)) (*trd3* (*castOp s X*)) = *X*
using *isOp-castOp* **by** *auto*

lemma *castOp-inj*[*simp*]:
assumes *isOp s X* **and** *isOp s′ X′*
shows (*castOp s X* = *castOp s′ X′*) = (*s* = *s′* ∧ *X* = *X′*)
using *assms Op-castOp stOf-castOp* **by** *fastforce*

lemmas *castOp-simps* =
castOp-stOf-Op wlsInp-castOp wlsBinp-castOp
stOf-castOp Op-castOp castOp-inj

lemma *not-isVar-isOp*:

$\neg$ (*isVar s X* $\wedge$ *isOp s X*)
unfolding *isVar-def isOp-def* **by** *auto*

lemma *isVar-or-isOp*:
wls s X $\Longrightarrow$ *isVar s X* $\vee$ *isOp s X*
by(*erule wls-cases*) *auto*

lemma *sortTermCase-asSort-Var-simp*[*simp*]:
sortTermCase fVar fOp (*asSort xs*) (*Var xs x*) = *fVar xs x*
unfolding *sortTermCase-def* **by** *auto*

lemma *sortTermCase-stOf-Op-simp*[*simp*]:
⟦*wlsInp delta inp*; *wlsBinp delta binp*⟧ $\Longrightarrow$
sortTermCase fVar fOp (*stOf delta*) (*Op delta inp binp*) = *fOp delta inp binp*
unfolding *sortTermCase-def* **by** *auto*

lemma *sortTermCase-cong*[*fundef-cong*]:
assumes $\bigwedge$ *xs x*. *fVar xs x* = *gVar xs x*
and $\bigwedge$ *delta inp binp*. ⟦*wlsInp delta inp*; *wlsInp delta inp*⟧
$\Longrightarrow$ *fOp delta inp binp* = *gOp delta inp binp*
shows *wls s X* $\Longrightarrow$
sortTermCase fVar fOp s X = *sortTermCase gVar gOp s X*
apply(*erule wls-cases*) **using** *assms* **by** *auto*

lemmas *sortTermCase-simps* =
sortTermCase-asSort-Var-simp
sortTermCase-stOf-Op-simp

lemmas *term-cast-simps* =
isOp-simps castOp-simps sortTermCase-simps

7.10.2 For abstractions

Here, the situation will be different than that of terms, since:
- an abstraction can only be built using “Abs", hence we need no “is" operators;
- the constructor “Abs" for abstractions is not injective, so need a more subtle condition on the case-analysis operator.

Yet another difference is that when casting an abstraction “A" such that “wlsAbs (xs,s) A", we need to cast only the value “A", and not the sorting part“xs s", since the latter already contains the desired information. Consequently, below, in the arguments for the case-analysis operator, the sorts “xs s" come before the function “f", and the latter doesnot take sorts into account.

definition *castAbs* **where**

castAbs xs s A ≡ *SOME x-X. wls s* (*snd x-X*) ∧ *A* = *Abs xs* (*fst x-X*) (*snd x-X*)

definition *absCase* **where**
absCase xs s f A ≡ *if wlsAbs* (*xs*,*s*) *A then f* (*fst* (*castAbs xs s A*)) (*snd* (*castAbs xs s A*)) *else undefined*

definition *compatAbsSwap* **where**
compatAbsSwap xs s f ≡
∀ *x X x′ X′*. (∀ *y*. (*y* = *x* ∨ *fresh xs y X*) ∧ (*y* = *x′* ∨ *fresh xs y X′*)
⟶ (*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*))
⟶ *f x X* = *f x′ X′*

definition *compatAbsSubst* **where**
compatAbsSubst xs s f ≡
∀ *x X x′ X′*. (∀ *Y*. *wls* (*asSort xs*) *Y* ⟶ (*X* #[*Y* / *x*]-*xs*) = (*X′* #[*Y* / *x′*]-*xs*))
⟶ *f x X* = *f x′ X′*

definition *compatAbsVsubst* **where**
compatAbsVsubst xs s f ≡
∀ *x X x′ X′*. (∀ *y*. (*X* #[*y* // *x*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*))
⟶ *f x X* = *f x′ X′*

lemma *wlsAbs-castAbs*:
assumes *wlsAbs* (*xs*,*s*) *A*
shows *wls s* (*snd* (*castAbs xs s A*)) ∧
Abs xs (*fst* (*castAbs xs s A*)) (*snd* (*castAbs xs s A*)) = *A*
proof−
let *?phi* = λ *x-X*. *wls s* (*snd x-X*) ∧
A = *Abs xs* (*fst x-X*) (*snd x-X*)
obtain *x X* **where** *?phi* (*x*,*X*) **using** *assms wlsAbs-nchotomy*[*of xs s A*] **by** *auto*
hence *?phi* (*castAbs xs s A*) **unfolding** *castAbs-def* **using** *someI*[*of ?phi*] **by** *auto*
thus *?thesis* **by** *simp*
qed

lemma *wls-castAbs*[*simp*]:
wlsAbs (*xs*,*s*) *A* ⟹ *wls s* (*snd* (*castAbs xs s A*))
using *wlsAbs-castAbs* **by** *auto*

lemma *Abs-castAbs*[*simp*]:
wlsAbs (*xs*,*s*) *A* ⟹ *Abs xs* (*fst* (*castAbs xs s A*)) (*snd* (*castAbs xs s A*)) = *A*
using *wlsAbs-castAbs* **by** *auto*

lemma *castAbs-Abs-swap*:
assumes *isInBar* (*xs*,*s*) **and** *X*: *wls s X*
and *yxX*: *y* = *x* ∨ *fresh xs y X* **and** *yx′X′*: *y* = *x′* ∨ *fresh xs y X′*
and ∗: *castAbs xs s* (*Abs xs x X*) = (*x′*,*X′*)

shows $(X \#[y \wedge x]\text{-}xs) = (X' \#[y \wedge x']\text{-}xs)$
proof−
  have *wlsAbs* (*xs*,*s*) (*Abs xs x X*) **using** *assms* **by** *simp*
  moreover
  have $x' = fst$ (*castAbs xs s* (*Abs xs x X*)) **and**
    $X' = snd$ (*castAbs xs s* (*Abs xs x X*)) **using** ∗ **by** *auto*
  ultimately
  have *wls s* X' **and** *Abs xs x X* = *Abs xs* x' X' **by** *auto*
  thus *?thesis* **using** *yxX* $yx'X'$ *X* **by**(*auto simp add*: *wls-Abs-swap-all*)
qed

lemma *castAbs-Abs-subst*:
assumes *isInBar*: *isInBar* (*xs*,*s*)
and *X*: *wls s X* **and** *Y*: *wls* (*asSort xs*) *Y*
and ∗: *castAbs xs s* (*Abs xs x X*) = (x',X')
shows $(X \#[Y / x]\text{-}xs) = (X' \#[Y / x']\text{-}xs)$
proof−
  have *wlsAbs* (*xs*,*s*) (*Abs xs x X*) **using** *isInBar X* **by** *simp*
  moreover
  have $x' = fst$ (*castAbs xs s* (*Abs xs x X*)) **and**
    $X' = snd$ (*castAbs xs s* (*Abs xs x X*)) **using** ∗ **by** *auto*
  ultimately
  have *wls s* X' **and** *Abs xs x X* = *Abs xs* x' X' **by** *auto*
  thus *?thesis* **using** *Y X* **by**(*auto simp add*: *wls-Abs-subst-all*)
qed

lemma *castAbs-Abs-vsubst*:
assumes *isInBar* (*xs*,*s*) **and** *wls s X*
and *castAbs xs s* (*Abs xs x X*) = (x',X')
shows $(X \#[y // x]\text{-}xs) = (X' \#[y // x']\text{-}xs)$
using *assms* **unfolding** *vsubst-def*
by (*intro castAbs-Abs-subst*) *auto*

lemma *castAbs-inj*[*simp*]:
assumes ∗: *wlsAbs* (*xs*,*s*) *A* **and** ∗∗: *wlsAbs* (*xs*,*s*) A'
shows (*castAbs xs s A* = *castAbs xs s* A') = ($A = A'$)
using *assms Abs-castAbs* **by** *fastforce*

lemmas *castAbs-simps* =
wls-castAbs Abs-castAbs castAbs-inj

lemma *absCase-Abs-swap*[*simp*]:
assumes *isInBar*: *isInBar* (*xs*,*s*) **and** *X*: *wls s X*
and *f-compat*: *compatAbsSwap xs s f*
shows *absCase xs s f* (*Abs xs x X*) = *f x X*
proof−
  obtain x' X' **where** *1*: *castAbs xs s* (*Abs xs x X*) = (x',X')

by (*cases castAbs xs s* (*Abs xs x X*), *auto*)
hence *2*: *absCase xs s f* (*Abs xs x X*) = *f x′ X′*
unfolding *absCase-def* **using** *isInBar X* **by** *auto*
have $\bigwedge$ *y*. (*y* = *x* ∨ *fresh xs y X*) ∧ (*y* = *x′* ∨ *fresh xs y X′*)
⟹ (*X* #[*y* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)
using *isInBar X 1* **by**(*simp add*: *castAbs-Abs-swap*)
hence *f x X* = *f x′ X′* **using** *f-compat*
unfolding *compatAbsSwap-def* **by** *fastforce*
thus *?thesis* **using** *2* **by** *simp*
qed

lemma *absCase-Abs-subst*[*simp*]:
assumes *isInBar*: *isInBar* (*xs*,*s*) **and** *X*: *wls s X*
and *f-compat*: *compatAbsSubst xs s f*
shows *absCase xs s f* (*Abs xs x X*) = *f x X*
proof−
obtain *x′ X′* **where** *1*: *castAbs xs s* (*Abs xs x X*) = (*x′*,*X′*)
by (*cases castAbs xs s* (*Abs xs x X*)) *auto*
hence *2*: *absCase xs s f* (*Abs xs x X*) = *f x′ X′*
unfolding *absCase-def* **using** *isInBar X* **by** *auto*
have $\bigwedge$ *Y*. *wls* (*asSort xs*) *Y* ⟹ (*X* #[*Y* / *x*]-*xs*) = (*X′* #[*Y* / *x′*]-*xs*)
using *isInBar X 1* **by**(*simp add*: *castAbs-Abs-subst*)
hence *f x X* = *f x′ X′* **using** *f-compat* **unfolding** *compatAbsSubst-def* **by** *blast*
thus *?thesis* **using** *2* **by** *simp*
qed

lemma *compatAbsVsubst-imp-compatAbsSubst*[*simp*]:
compatAbsVsubst xs s f ⟹ *compatAbsSubst xs s f*
unfolding *compatAbsSubst-def compatAbsVsubst-def*
vsubst-def **by** *auto*

lemma *absCase-Abs-vsubst*[*simp*]:
assumes *isInBar* (*xs*,*s*) **and** *wls s X*
and *compatAbsVsubst xs s f*
shows *absCase xs s f* (*Abs xs x X*) = *f x X*
using *assms* **by**(*simp add*: *absCase-Abs-subst*)

lemma *absCase-cong*[*fundef-cong*]:
assumes *compatAbsSwap xs s f* ∨ *compatAbsSubst xs s f* ∨ *compatAbsVsubst xs s f*
and *compatAbsSwap xs s f′* ∨ *compatAbsSubst xs s f′* ∨ *compatAbsVsubst xs s f′*
and $\bigwedge$ *x X*. *wls s X* ⟹ *f x X* = *f′ x X*
shows *wlsAbs* (*xs*,*s*) *A* ⟹
absCase xs s f A = *absCase xs s f′ A*
apply(*erule wlsAbs-cases*) **using** *assms* **by** *auto*

lemmas *absCase-simps* = *absCase-Abs-swap absCase-Abs-subst*
compatAbsVsubst-imp-compatAbsSubst absCase-Abs-vsubst

lemmas *abs-cast-simps* = *castAbs-simps absCase-simps*

lemmas *cast-simps* = *term-cast-simps abs-cast-simps*

lemmas *wls-item-simps* =
wlsAll-imp-goodAll paramS-simps Cons-wls-simps all-preserve-wls
wls-freeCons wls-allOpers-simps wls-allOpers-otherSimps Abs-inj-fresh cast-simps

lemmas *wls-copy-of-good-item-simps* = *good-freeCons good-allOpers-simps good-allOpers-otherSimps*
param-simps all-preserve-good

declare *wls-copy-of-good-item-simps* [*simp del*]
declare *qItem-simps* [*simp del*] **declare** *qItem-versus-item-simps* [*simp del*]

end

end

8 Iteration

theory *Iteration* **imports** *Well-Sorted-Terms*
begin

In this section, we introduce first-order models (models, for short). These are structures having operators that match those for terms (including variable-injection, binding operations, freshness, swapping and substitution) and satisfy some clauses, and show that terms form initial models. This gives iteration principles.

As a matter of notation: the prefix "g" will stand for "generalized" – elements of models are referred to as "generalized terms". The actual full prefix will be "ig" (where "i" stands for "iteration"), symbolizing the fact that the models from this section support iteration, and not general recursion. The latter is dealt with by the models introduced in the next section, for which we use the simple prefix "g".

8.1 Models

We have two basic kinds of models:
- fresh-swap (FSw) models, featuring operations corresponding to the concrete syntactic constructs ("Var", "Op", "Abs"), henceforth referred to simply as *the constructs*, and to fresh and swap;
- fresh-swap-subst (FSb) models, featuring substitution instead of swapping.

We also consider two combinations of the above, FSwSb-models and FSbSw-models.

To keep things structurally simple, we use one single Isabelle for all the 4 kinds models, allowing the most generous signature. Since terms are the main actors of our theory, models being considered only for the sake of recursive definitions, we call the items inhabiting these models "generalized" terms, abstractions and inputs, and correspondingly the operations; hence the prefix "g" from the names of the type parameters and operators. (However, we refer to the generalized items using the same notations as for "concrete items": X, A, etc.) Indeed, a model can be regarded as implementing a generalization/axiomatization of the term structure, where now the objects are not terms, but do have term-like properties.

8.1.1 Raw models

record ('*index*,'*bindex*,'*varSort*,'*sort*,'*opSym*,'*var*,'*gTerm*,'*gAbs*)*model* =
  igWls :: '*sort* ⇒ '*gTerm* ⇒ *bool*
  igWlsAbs :: '*varSort* × '*sort* ⇒ '*gAbs* ⇒ *bool*

  igVar :: '*varSort* ⇒ '*var* ⇒ '*gTerm*
  igAbs :: '*varSort* ⇒ '*var* ⇒ '*gTerm* ⇒ '*gAbs*
  igOp :: '*opSym* ⇒ ('*index*,'*gTerm*)*input* ⇒ ('*bindex*,'*gAbs*)*input* ⇒ '*gTerm*

  igFresh :: '*varSort* ⇒ '*var* ⇒ '*gTerm* ⇒ *bool*
  igFreshAbs :: '*varSort* ⇒ '*var* ⇒ '*gAbs* ⇒ *bool*

  igSwap :: '*varSort* ⇒ '*var* ⇒ '*var* ⇒ '*gTerm* ⇒ '*gTerm*
  igSwapAbs :: '*varSort* ⇒ '*var* ⇒ '*var* ⇒ '*gAbs* ⇒ '*gAbs*

  igSubst :: '*varSort* ⇒ '*gTerm* ⇒ '*var* ⇒ '*gTerm* ⇒ '*gTerm*
  igSubstAbs :: '*varSort* ⇒ '*gTerm* ⇒ '*var* ⇒ '*gAbs* ⇒ '*gAbs*

- "igSwap MOD zs z1 z2 X" swaps in X z1 and z2 (assumed of sorts zs).
- "igSubst MOD ys Y x X" substitutes, in X, Y with y (assumed of sort ys).

definition *igFreshInp* **where**
igFreshInp MOD ys y inp == *liftAll* (*igFresh MOD ys y*) *inp*

definition *igFreshBinp* **where**
igFreshBinp MOD ys y binp == *liftAll* (*igFreshAbs MOD ys y*) *binp*

definition *igSwapInp* **where**
igSwapInp MOD zs z1 z2 inp == *lift* (*igSwap MOD zs z1 z2*) *inp*

definition *igSwapBinp* **where**
igSwapBinp MOD zs z1 z2 binp == *lift* (*igSwapAbs MOD zs z1 z2*) *binp*

definition *igSubstInp* **where**
igSubstInp MOD ys Y y inp == lift (igSubst MOD ys Y y) inp

definition *igSubstBinp* **where**
igSubstBinp MOD ys Y y binp == lift (igSubstAbs MOD ys Y y) binp

context *FixSyn*
begin

8.1.2 Well-sorted models of various kinds

We define the following kinds of well-sorted models
- fresh-swap models (predicate "iwlsFSw");
- fresh-subst models ("iwlsFSb");
- fresh-swap-subst models ("iwlsFSwSb");
- fresh-subst-swap models ("iwlsFSbSw").

All of these models are defined as raw models subject to various Horn conditions:
- For "iwlsFSw":
— definition-like clauses for "fresh" and "swap" in terms of the construct operators;
— congruence for abstraction based on fresh and swap (mirroring the abstraction case in the definition of alpha-equivalence for quasi-terms). [2]
- For "iwlsFSb": the same as for "iwlsFSw", except that:
— "swap" is replaced by "subst"; [3]
— The [fresh and swap]-based congrunce clause is replaced by an "abstraction-renaming" clause, which is stronger than the corresponding [fresh and subst]-based congruence clause. [4]
- For "iwlsFSwSb": the clauses for "iwlsFSw", plus some of the definition-like clauses for "subst". [5]
- For "iwlsFSbSw": the clauses for "iwlsFSb", plus definition-like clauses for "swap".

Thus, a fresh-swap-subst model is also a fresh-swap model, and a fresh-subst-swap model is also a fresh-subst model.

For convenience, all these 4 kinds of models are defined on one single type,

[2] Here, by "congruence for abstraction" we do not mean the standard notion of congrunece (satisfied by any operator once or ever), but a *stronger* notion: in order for two abstractions to be equal, it is not required that their ariguments be equal, but that they be in a "permutative" relationship based either on swapping or on substitution.

[3] Note that traditionally alpha-equivalence is defined using "subst", not "swap".

[4] We also define the [fresh and subst]-based congruence clause, although we do not employ it directly in the definition of any kind of model.

[5] Not all the "subst" definition-like clauses from "iwlsFSb" are required for "iwlsFSwSb" – namely, the clause that we call "igSubstIGAbsCls2" is not required here.

that of *raw models*, which interpret the most generous signature, comprizing all the operations and relations required by all 4 kinds of models. Note that, although some operations (namely, "subst" or "swap") may not be involved in the clauses for certain kinds of models, the extra structure is harmless to the development of their theory.

Note that for the models operations and relations we do not actually write "fresh", "swap" and "subst", but "igFresh", "igSwap" and "igSubst".

As usual, we shall have not only term versions, but also abstraction versions of the above operations.

definition *igWlsInp* **where**
igWlsInp MOD delta inp ==
wlsOpS delta ∧ *sameDom* (*arOf delta*) *inp* ∧ *liftAll2* (*igWls MOD*) (*arOf delta*) *inp*

lemmas *igWlsInp-defs* = *igWlsInp-def sameDom-def liftAll2-def*

definition *igWlsBinp* **where**
igWlsBinp MOD delta binp ==
wlsOpS delta ∧ *sameDom* (*barOf delta*) *binp* ∧ *liftAll2* (*igWlsAbs MOD*) (*barOf delta*) *binp*

lemmas *igWlsBinp-defs* = *igWlsBinp-def sameDom-def liftAll2-def*

Domain disjointness:

definition *igWlsDisj* **where**
igWlsDisj MOD == ∀ *s s′ X*. *igWls MOD s X* ∧ *igWls MOD s′ X* ⟶ *s* = *s′*

definition *igWlsAbsDisj* **where**
igWlsAbsDisj MOD ==
∀ *xs s xs′ s′ A*.
  isInBar (*xs*,*s*) ∧ *isInBar* (*xs′*,*s′*) ∧
  igWlsAbs MOD (*xs*,*s*) *A* ∧ *igWlsAbs MOD* (*xs′*,*s′*) *A*
  ⟶ *xs* = *xs′* ∧ *s* = *s′*

definition *igWlsAllDisj* **where**
igWlsAllDisj MOD ==
igWlsDisj MOD ∧ *igWlsAbsDisj MOD*

lemmas *igWlsAllDisj-defs* =
igWlsAllDisj-def
igWlsDisj-def igWlsAbsDisj-def

Abstration domains inhabited only within bound arities:

definition *igWlsAbsIsInBar* **where**
igWlsAbsIsInBar MOD ==
∀ *us s A*. *igWlsAbs MOD* (*us*,*s*) *A* ⟶ *isInBar* (*us*,*s*)

Domain preservation by the operators: weak ("if") versions and strong ("iff") versions (for the latter, we use the suffix "STR"):

The constructs preserve the domains:

definition *igVarIPresIGWls* **where**
igVarIPresIGWls MOD ==
$\forall$ *xs x. igWls MOD (asSort xs) (igVar MOD xs x)*

definition *igAbsIPresIGWls* **where**
igAbsIPresIGWls MOD ==
$\forall$ *xs s x X. isInBar (xs,s)* $\wedge$ *igWls MOD s X* $\longrightarrow$
igWlsAbs MOD (xs,s) (igAbs MOD xs x X)

definition *igAbsIPresIGWlsSTR* **where**
igAbsIPresIGWlsSTR MOD ==
$\forall$ *xs s x X. isInBar (xs,s)* $\longrightarrow$
igWlsAbs MOD (xs,s) (igAbs MOD xs x X) =
igWls MOD s X

lemma *igAbsIPresIGWlsSTR-imp-igAbsIPresIGWls*:
igAbsIPresIGWlsSTR MOD $\Longrightarrow$ *igAbsIPresIGWls MOD*
unfolding *igAbsIPresIGWlsSTR-def igAbsIPresIGWls-def* **by** *simp*

definition *igOpIPresIGWls* **where**
igOpIPresIGWls MOD ==
$\forall$ *delta inp binp.*
igWlsInp MOD delta inp $\wedge$ *igWlsBinp MOD delta binp*
$\longrightarrow$ *igWls MOD (stOf delta) (igOp MOD delta inp binp)*

definition *igOpIPresIGWlsSTR* **where**
igOpIPresIGWlsSTR MOD ==
$\forall$ *delta inp binp.*
igWls MOD (stOf delta) (igOp MOD delta inp binp) =
(igWlsInp MOD delta inp $\wedge$ *igWlsBinp MOD delta binp)*

lemma *igOpIPresIGWlsSTR-imp-igOpIPresIGWls*:
igOpIPresIGWlsSTR MOD $\Longrightarrow$ *igOpIPresIGWls MOD*
unfolding *igOpIPresIGWlsSTR-def igOpIPresIGWls-def* **by** *simp*

definition *igConsIPresIGWls* **where**
igConsIPresIGWls MOD ==
igVarIPresIGWls MOD $\wedge$
igAbsIPresIGWls MOD $\wedge$
igOpIPresIGWls MOD

lemmas *igConsIPresIGWls-defs = igConsIPresIGWls-def*
igVarIPresIGWls-def
igAbsIPresIGWls-def
igOpIPresIGWls-def

definition *igConsIPresIGWlsSTR* **where**
igConsIPresIGWlsSTR MOD ==
igVarIPresIGWls MOD ∧
igAbsIPresIGWlsSTR MOD ∧
igOpIPresIGWlsSTR MOD

lemmas *igConsIPresIGWlsSTR-defs = igConsIPresIGWlsSTR-def*
igVarIPresIGWls-def
igAbsIPresIGWlsSTR-def
igOpIPresIGWlsSTR-def

lemma *igConsIPresIGWlsSTR-imp-igConsIPresIGWls*:
igConsIPresIGWlsSTR MOD ⟹ *igConsIPresIGWls MOD*
unfolding *igConsIPresIGWlsSTR-def igConsIPresIGWls-def*
using
igAbsIPresIGWlsSTR-imp-igAbsIPresIGWls
igOpIPresIGWlsSTR-imp-igOpIPresIGWls
by *auto*

"swap" preserves the domains:

definition *igSwapIPresIGWls* **where**
igSwapIPresIGWls MOD ==
∀ *zs z1 z2 s X. igWls MOD s X* ⟶
igWls MOD s (*igSwap MOD zs z1 z2 X*)

definition *igSwapIPresIGWlsSTR* **where**
igSwapIPresIGWlsSTR MOD ==
∀ *zs z1 z2 s X. igWls MOD s* (*igSwap MOD zs z1 z2 X*) =
igWls MOD s X

lemma *igSwapIPresIGWlsSTR-imp-igSwapIPresIGWls*:
igSwapIPresIGWlsSTR MOD ⟹ *igSwapIPresIGWls MOD*
unfolding *igSwapIPresIGWlsSTR-def igSwapIPresIGWls-def* **by** *simp*

definition *igSwapAbsIPresIGWlsAbs* **where**
igSwapAbsIPresIGWlsAbs MOD ==
∀ *zs z1 z2 us s A.*
isInBar (*us*,*s*) ∧ *igWlsAbs MOD* (*us*,*s*) *A* ⟶
igWlsAbs MOD (*us*,*s*) (*igSwapAbs MOD zs z1 z2 A*)

definition *igSwapAbsIPresIGWlsAbsSTR* **where**
igSwapAbsIPresIGWlsAbsSTR MOD ==
∀ *zs z1 z2 us s A.*
igWlsAbs MOD (*us*,*s*) (*igSwapAbs MOD zs z1 z2 A*) =
igWlsAbs MOD (*us*,*s*) *A*

lemma *igSwapAbsIPresIGWlsAbsSTR-imp-igSwapAbsIPresIGWlsAbs*:
igSwapAbsIPresIGWlsAbsSTR MOD ⟹ *igSwapAbsIPresIGWlsAbs MOD*

unfolding *igSwapAbsIPresIGWlsAbsSTR-def igSwapAbsIPresIGWlsAbs-def* **by** *simp*

definition *igSwapAllIPresIGWlsAll* **where**
igSwapAllIPresIGWlsAll MOD ==
igSwapIPresIGWls MOD ∧ *igSwapAbsIPresIGWlsAbs MOD*

lemmas *igSwapAllIPresIGWlsAll-defs* = *igSwapAllIPresIGWlsAll-def*
igSwapIPresIGWls-def igSwapAbsIPresIGWlsAbs-def

definition *igSwapAllIPresIGWlsAllSTR* **where**
igSwapAllIPresIGWlsAllSTR MOD ==
igSwapIPresIGWlsSTR MOD ∧ *igSwapAbsIPresIGWlsAbsSTR MOD*

lemmas *igSwapAllIPresIGWlsAllSTR-defs* = *igSwapAllIPresIGWlsAllSTR-def*
igSwapIPresIGWlsSTR-def igSwapAbsIPresIGWlsAbsSTR-def

lemma *igSwapAllIPresIGWlsAllSTR-imp-igSwapAllIPresIGWlsAll*:
igSwapAllIPresIGWlsAllSTR MOD ⟹ *igSwapAllIPresIGWlsAll MOD*
unfolding *igSwapAllIPresIGWlsAllSTR-def igSwapAllIPresIGWlsAll-def*
using
igSwapIPresIGWlsSTR-imp-igSwapIPresIGWls
igSwapAbsIPresIGWlsAbsSTR-imp-igSwapAbsIPresIGWlsAbs
by *auto*

"subst" preserves the domains:

definition *igSubstIPresIGWls* **where**
igSubstIPresIGWls MOD ==
∀ *ys Y y s X*. *igWls MOD* (*asSort ys*) *Y* ∧ *igWls MOD s X* ⟶
igWls MOD s (*igSubst MOD ys Y y X*)

definition *igSubstIPresIGWlsSTR* **where**
igSubstIPresIGWlsSTR MOD ==
∀ *ys Y y s X*.
igWls MOD s (*igSubst MOD ys Y y X*) =
(*igWls MOD* (*asSort ys*) *Y* ∧ *igWls MOD s X*)

lemma *igSubstIPresIGWlsSTR-imp-igSubstIPresIGWls*:
igSubstIPresIGWlsSTR MOD ⟹ *igSubstIPresIGWls MOD*
unfolding *igSubstIPresIGWlsSTR-def igSubstIPresIGWls-def* **by** *simp*

definition *igSubstAbsIPresIGWlsAbs* **where**
igSubstAbsIPresIGWlsAbs MOD ==
∀ *ys Y y us s A*.
isInBar (*us*,*s*) ∧ *igWls MOD* (*asSort ys*) *Y* ∧ *igWlsAbs MOD* (*us*,*s*) *A* ⟶
igWlsAbs MOD (*us*,*s*) (*igSubstAbs MOD ys Y y A*)

definition *igSubstAbsIPresIGWlsAbsSTR* **where**
igSubstAbsIPresIGWlsAbsSTR MOD ==
∀ *ys Y y us s A*.

igWlsAbs MOD (us,s) (igSubstAbs MOD ys Y y A) =
(igWls MOD (asSort ys) Y ∧ igWlsAbs MOD (us,s) A)

lemma *igSubstAbsIPresIGWlsAbsSTR-imp-igSubstAbsIPresIGWlsAbs*:
igSubstAbsIPresIGWlsAbsSTR MOD ⟹ igSubstAbsIPresIGWlsAbs MOD
unfolding *igSubstAbsIPresIGWlsAbsSTR-def igSubstAbsIPresIGWlsAbs-def* **by** *simp*

definition *igSubstAllIPresIGWlsAll* **where**
igSubstAllIPresIGWlsAll MOD ==
igSubstIPresIGWls MOD ∧ igSubstAbsIPresIGWlsAbs MOD

lemmas *igSubstAllIPresIGWlsAll-defs = igSubstAllIPresIGWlsAll-def*
igSubstIPresIGWls-def igSubstAbsIPresIGWlsAbs-def

definition *igSubstAllIPresIGWlsAllSTR* **where**
igSubstAllIPresIGWlsAllSTR MOD ==
igSubstIPresIGWlsSTR MOD ∧ igSubstAbsIPresIGWlsAbsSTR MOD

lemmas *igSubstAllIPresIGWlsAllSTR-defs = igSubstAllIPresIGWlsAllSTR-def*
igSubstIPresIGWlsSTR-def igSubstAbsIPresIGWlsAbsSTR-def

lemma *igSubstAllIPresIGWlsAllSTR-imp-igSubstAllIPresIGWlsAll*:
igSubstAllIPresIGWlsAllSTR MOD ⟹ igSubstAllIPresIGWlsAll MOD
unfolding *igSubstAllIPresIGWlsAllSTR-def igSubstAllIPresIGWlsAll-def*
using
igSubstIPresIGWlsSTR-imp-igSubstIPresIGWls
igSubstAbsIPresIGWlsAbsSTR-imp-igSubstAbsIPresIGWlsAbs
by *auto*

Clauses for fresh: fully conditional versions and less conditional, stronger versions (the latter having suffix "STR").

definition *igFreshIGVar* **where**
igFreshIGVar MOD ==
∀ ys y xs x.
ys ≠ xs ∨ y ≠ x ⟶
igFresh MOD ys y (igVar MOD xs x)

definition *igFreshIGAbs1* **where**
igFreshIGAbs1 MOD ==
∀ ys y s X.
isInBar (ys,s) ∧ igWls MOD s X ⟶
igFreshAbs MOD ys y (igAbs MOD ys y X)

definition *igFreshIGAbs1STR* **where**
igFreshIGAbs1STR MOD ==
∀ ys y X. igFreshAbs MOD ys y (igAbs MOD ys y X)

lemma *igFreshIGAbs1STR-imp-igFreshIGAbs1*:
igFreshIGAbs1STR MOD ⟹ igFreshIGAbs1 MOD

unfolding *igFreshIGAbs1STR-def igFreshIGAbs1-def* **by** *simp*

definition *igFreshIGAbs2* **where**
igFreshIGAbs2 MOD ==
$\forall$ *ys y xs x s X.*
isInBar (xs,s) $\wedge$ *igWls MOD s X* $\longrightarrow$
igFresh MOD ys y X $\longrightarrow$ *igFreshAbs MOD ys y (igAbs MOD xs x X)*

definition *igFreshIGAbs2STR* **where**
igFreshIGAbs2STR MOD ==
$\forall$ *ys y xs x X.*
igFresh MOD ys y X $\longrightarrow$ *igFreshAbs MOD ys y (igAbs MOD xs x X)*

lemma *igFreshIGAbs2STR-imp-igFreshIGAbs2*:
igFreshIGAbs2STR MOD $\Longrightarrow$ *igFreshIGAbs2 MOD*
unfolding *igFreshIGAbs2STR-def igFreshIGAbs2-def* **by** *simp*

definition *igFreshIGOp* **where**
igFreshIGOp MOD ==
$\forall$ *ys y delta inp binp.*
igWlsInp MOD delta inp $\wedge$ *igWlsBinp MOD delta binp* $\longrightarrow$
(igFreshInp MOD ys y inp $\wedge$ *igFreshBinp MOD ys y binp)* $\longrightarrow$
igFresh MOD ys y (igOp MOD delta inp binp)

definition *igFreshIGOpSTR* **where**
igFreshIGOpSTR MOD ==
$\forall$ *ys y delta inp binp.*
igFreshInp MOD ys y inp $\wedge$ *igFreshBinp MOD ys y binp* $\longrightarrow$
igFresh MOD ys y (igOp MOD delta inp binp)

lemma *igFreshIGOpSTR-imp-igFreshIGOp*:
igFreshIGOpSTR MOD $\Longrightarrow$ *igFreshIGOp MOD*
unfolding *igFreshIGOpSTR-def igFreshIGOp-def* **by** *simp*

definition *igFreshCls* **where**
igFreshCls MOD ==
igFreshIGVar MOD $\wedge$
igFreshIGAbs1 MOD $\wedge$ *igFreshIGAbs2 MOD* $\wedge$
igFreshIGOp MOD

lemmas *igFreshCls-defs = igFreshCls-def*
igFreshIGVar-def
igFreshIGAbs1-def igFreshIGAbs2-def
igFreshIGOp-def

definition *igFreshClsSTR* **where**
igFreshClsSTR MOD ==
igFreshIGVar MOD $\wedge$
igFreshIGAbs1STR MOD $\wedge$ *igFreshIGAbs2STR MOD* $\wedge$

igFreshIGOpSTR MOD

lemmas *igFreshClsSTR-defs = igFreshClsSTR-def*
igFreshIGVar-def
igFreshIGAbs1STR-def igFreshIGAbs2STR-def
igFreshIGOpSTR-def

lemma *igFreshClsSTR-imp-igFreshCls*:
igFreshClsSTR MOD ⟹ igFreshCls MOD
unfolding *igFreshClsSTR-def igFreshCls-def*
using
igFreshIGAbs1STR-imp-igFreshIGAbs1 igFreshIGAbs2STR-imp-igFreshIGAbs2
igFreshIGOpSTR-imp-igFreshIGOp
by *auto*

definition *igSwapIGVar* **where**
igSwapIGVar MOD ==
∀ *zs z1 z2 xs x.*
igSwap MOD zs z1 z2 (*igVar MOD xs x*) = *igVar MOD xs* (*x @xs*[*z1* ∧ *z2*]-*zs*)

definition *igSwapIGAbs* **where**
igSwapIGAbs MOD ==
∀ *zs z1 z2 xs x s X*.
isInBar (*xs*,*s*) ∧ *igWls MOD s X* ⟶
igSwapAbs MOD zs z1 z2 (*igAbs MOD xs x X*) =
igAbs MOD xs (*x @xs*[*z1* ∧ *z2*]-*zs*) (*igSwap MOD zs z1 z2 X*)

definition *igSwapIGAbsSTR* **where**
igSwapIGAbsSTR MOD ==
∀ *zs z1 z2 xs x X*.
igSwapAbs MOD zs z1 z2 (*igAbs MOD xs x X*) =
igAbs MOD xs (*x @xs*[*z1* ∧ *z2*]-*zs*) (*igSwap MOD zs z1 z2 X*)

lemma *igSwapIGAbsSTR-imp-igSwapIGAbs*:
igSwapIGAbsSTR MOD ⟹ igSwapIGAbs MOD
unfolding *igSwapIGAbsSTR-def igSwapIGAbs-def* **by** *simp*

definition *igSwapIGOp* **where**
igSwapIGOp MOD ==
∀ *zs z1 z2 delta inp binp.*
igWlsInp MOD delta inp ∧ *igWlsBinp MOD delta binp* ⟶
igSwap MOD zs z1 z2 (*igOp MOD delta inp binp*) =
igOp MOD delta (*igSwapInp MOD zs z1 z2 inp*) (*igSwapBinp MOD zs z1 z2 binp*)

definition *igSwapIGOpSTR* **where**
igSwapIGOpSTR MOD ==

∀ zs z1 z2 delta inp binp.
igSwap MOD zs z1 z2 (igOp MOD delta inp binp) =
igOp MOD delta (igSwapInp MOD zs z1 z2 inp) (igSwapBinp MOD zs z1 z2 binp)

lemma *igSwapIGOpSTR-imp-igSwapIGOp*:
igSwapIGOpSTR MOD ⟹ *igSwapIGOp MOD*
unfolding *igSwapIGOpSTR-def igSwapIGOp-def* **by** *simp*

definition *igSwapCls* **where**
igSwapCls MOD ==
igSwapIGVar MOD ∧
igSwapIGAbs MOD ∧
igSwapIGOp MOD

lemmas *igSwapCls-defs* = *igSwapCls-def*
igSwapIGVar-def
igSwapIGAbs-def
igSwapIGOp-def

definition *igSwapClsSTR* **where**
igSwapClsSTR MOD ==
igSwapIGVar MOD ∧
igSwapIGAbsSTR MOD ∧
igSwapIGOpSTR MOD

lemmas *igSwapClsSTR-defs* = *igSwapClsSTR-def*
igSwapIGVar-def
igSwapIGAbsSTR-def
igSwapIGOpSTR-def

lemma *igSwapClsSTR-imp-igSwapCls*:
igSwapClsSTR MOD ⟹ *igSwapCls MOD*
unfolding *igSwapClsSTR-def igSwapCls-def*
using
igSwapIGAbsSTR-imp-igSwapIGAbs
igSwapIGOpSTR-imp-igSwapIGOp
by *auto*

definition *igSubstIGVar1* **where**
igSubstIGVar1 MOD ==
∀ ys y Y xs x.
igWls MOD (asSort ys) Y ⟶
(ys ≠ xs ∨ y ≠ x) ⟶
igSubst MOD ys Y y (igVar MOD xs x) = igVar MOD xs x

definition *igSubstIGVar1STR* **where**

igSubstIGVar1STR MOD ==
(∀ *ys y y1 xs x*.
  (*ys* ≠ *xs* ∨ *x* ≠ *y*) ⟶
  igSubst MOD ys (*igVar MOD ys y1*) *y* (*igVar MOD xs x*) = *igVar MOD xs x*)
∧
(∀ *ys y Y xs x*.
  igWls MOD (*asSort ys*) *Y* ⟶
  (*ys* ≠ *xs* ∨ *y* ≠ *x*) ⟶
  igSubst MOD ys Y y (*igVar MOD xs x*) = *igVar MOD xs x*)

lemma *igSubstIGVar1STR-imp-igSubstIGVar1*:
igSubstIGVar1STR MOD ⟹ *igSubstIGVar1 MOD*
unfolding *igSubstIGVar1STR-def igSubstIGVar1-def* **by** *simp*

definition *igSubstIGVar2* **where**
igSubstIGVar2 MOD ==
∀ *ys y Y*.
  igWls MOD (*asSort ys*) *Y* ⟶
  igSubst MOD ys Y y (*igVar MOD ys y*) = *Y*

definition *igSubstIGVar2STR* **where**
igSubstIGVar2STR MOD ==
(∀ *ys y y1*.
  igSubst MOD ys (*igVar MOD ys y1*) *y* (*igVar MOD ys y*) = *igVar MOD ys y1*)
∧
(∀ *ys y Y*.
  igWls MOD (*asSort ys*) *Y* ⟶
  igSubst MOD ys Y y (*igVar MOD ys y*) = *Y*)

lemma *igSubstIGVar2STR-imp-igSubstIGVar2*:
igSubstIGVar2STR MOD ⟹ *igSubstIGVar2 MOD*
unfolding *igSubstIGVar2STR-def igSubstIGVar2-def* **by** *simp*

definition *igSubstIGAbs* **where**
igSubstIGAbs MOD ==
∀ *ys y Y xs x s X*.
  isInBar (*xs*,*s*) ∧ *igWls MOD* (*asSort ys*) *Y* ∧ *igWls MOD s X* ⟶
  (*xs* ≠ *ys* ∨ *x* ≠ *y*) ∧ *igFresh MOD xs x Y* ⟶
  igSubstAbs MOD ys Y y (*igAbs MOD xs x X*) =
  igAbs MOD xs x (*igSubst MOD ys Y y X*)

definition *igSubstIGAbsSTR* **where**
igSubstIGAbsSTR MOD ==
∀ *ys y Y xs x X*.
  (*xs* ≠ *ys* ∨ *x* ≠ *y*) ∧ *igFresh MOD xs x Y* ⟶
  igSubstAbs MOD ys Y y (*igAbs MOD xs x X*) =
  igAbs MOD xs x (*igSubst MOD ys Y y X*)

lemma *igSubstIGAbsSTR-imp-igSubstIGAbs*:

igSubstIGAbsSTR MOD ⟹ *igSubstIGAbs MOD*
unfolding *igSubstIGAbsSTR-def igSubstIGAbs-def* **by** *simp*

definition *igSubstIGOp* **where**
igSubstIGOp MOD ==
∀ *ys y Y delta inp binp*.
  igWls MOD (*asSort ys*) *Y* ∧
  igWlsInp MOD delta inp ∧ *igWlsBinp MOD delta binp* ⟶
  igSubst MOD ys Y y (*igOp MOD delta inp binp*) =
  igOp MOD delta (*igSubstInp MOD ys Y y inp*) (*igSubstBinp MOD ys Y y binp*)

definition *igSubstIGOpSTR* **where**
igSubstIGOpSTR MOD ==
(∀ *ys y y1 delta inp binp*.
  igSubst MOD ys (*igVar MOD ys y1*) *y* (*igOp MOD delta inp binp*) =
  igOp MOD delta (*igSubstInp MOD ys* (*igVar MOD ys y1*) *y inp*)
    (*igSubstBinp MOD ys* (*igVar MOD ys y1*) *y binp*))
∧
(∀ *ys y Y delta inp binp*.
  igWls MOD (*asSort ys*) *Y* ⟶
  igSubst MOD ys Y y (*igOp MOD delta inp binp*) =
  igOp MOD delta (*igSubstInp MOD ys Y y inp*) (*igSubstBinp MOD ys Y y binp*))

lemma *igSubstIGOpSTR-imp-igSubstIGOp*:
igSubstIGOpSTR MOD ⟹ *igSubstIGOp MOD*
unfolding *igSubstIGOpSTR-def igSubstIGOp-def* **by** *simp*

definition *igSubstCls* **where**
igSubstCls MOD ==
igSubstIGVar1 MOD ∧ *igSubstIGVar2 MOD* ∧
igSubstIGAbs MOD ∧
igSubstIGOp MOD

lemmas *igSubstCls-defs* = *igSubstCls-def*
igSubstIGVar1-def igSubstIGVar2-def
igSubstIGAbs-def
igSubstIGOp-def

definition *igSubstClsSTR* **where**
igSubstClsSTR MOD ==
igSubstIGVar1STR MOD ∧ *igSubstIGVar2STR MOD* ∧
igSubstIGAbsSTR MOD ∧
igSubstIGOpSTR MOD

lemmas *igSubstClsSTR-defs* = *igSubstClsSTR-def*
igSubstIGVar1STR-def igSubstIGVar2STR-def
igSubstIGAbsSTR-def
igSubstIGOpSTR-def

lemma *igSubstClsSTR-imp-igSubstCls*:
igSubstClsSTR MOD $\Longrightarrow$ *igSubstCls MOD*
unfolding *igSubstClsSTR-def igSubstCls-def*
using
igSubstIGVar1STR-imp-igSubstIGVar1
igSubstIGVar2STR-imp-igSubstIGVar2
igSubstIGAbsSTR-imp-igSubstIGAbs
igSubstIGOpSTR-imp-igSubstIGOp
by *auto*

definition *igAbsCongS* **where**
igAbsCongS MOD ==
$\forall$ *xs x x′ y s X X′*.
  isInBar (*xs*,*s*) $\wedge$ *igWls MOD s X* $\wedge$ *igWls MOD s X′* $\longrightarrow$
  igFresh MOD xs y X $\wedge$ *igFresh MOD xs y X′* $\wedge$ *igSwap MOD xs y x X* = *igSwap MOD xs y x′ X′* $\longrightarrow$
  igAbs MOD xs x X = *igAbs MOD xs x′ X′*

definition *igAbsCongSSTR* **where**
igAbsCongSSTR MOD ==
$\forall$ *xs x x′ y X X′*.
  igFresh MOD xs y X $\wedge$ *igFresh MOD xs y X′* $\wedge$ *igSwap MOD xs y x X* = *igSwap MOD xs y x′ X′* $\longrightarrow$
  igAbs MOD xs x X = *igAbs MOD xs x′ X′*

lemma *igAbsCongSSTR-imp-igAbsCongS*:
igAbsCongSSTR MOD $\Longrightarrow$ *igAbsCongS MOD*
unfolding *igAbsCongSSTR-def igAbsCongS-def* **by** *auto*

definition *igAbsCongU* **where**
igAbsCongU MOD ==
$\forall$ *xs x x′ y s X X′*.
  isInBar (*xs*,*s*) $\wedge$ *igWls MOD s X* $\wedge$ *igWls MOD s X′* $\longrightarrow$
  igFresh MOD xs y X $\wedge$ *igFresh MOD xs y X′* $\wedge$
  igSubst MOD xs (*igVar MOD xs y*) *x X* = *igSubst MOD xs* (*igVar MOD xs y*) *x′ X′* $\longrightarrow$
  igAbs MOD xs x X = *igAbs MOD xs x′ X′*

definition *igAbsCongUSTR* **where**
igAbsCongUSTR MOD ==
$\forall$ *xs x x′ y X X′*.
  igFresh MOD xs y X $\wedge$ *igFresh MOD xs y X′* $\wedge$
  igSubst MOD xs (*igVar MOD xs y*) *x X* = *igSubst MOD xs* (*igVar MOD xs y*)

x′ X′ ⟶
  igAbs MOD xs x X = *igAbs MOD xs x′ X′*

lemma *igAbsCongUSTR-imp-igAbsCongU*:
igAbsCongUSTR MOD ⟹ *igAbsCongU MOD*
unfolding *igAbsCongUSTR-def igAbsCongU-def* **by** *auto*

definition *igAbsRen* **where**
igAbsRen MOD ==
∀ *xs y x s X*.
  isInBar (*xs*,*s*) ∧ *igWls MOD s X* ⟶
  igFresh MOD xs y X ⟶
  igAbs MOD xs y (*igSubst MOD xs* (*igVar MOD xs y*) *x X*) = *igAbs MOD xs x*
X

definition *igAbsRenSTR* **where**
igAbsRenSTR MOD ==
∀ *xs y x X*.
  igFresh MOD xs y X ⟶
  igAbs MOD xs y (*igSubst MOD xs* (*igVar MOD xs y*) *x X*) = *igAbs MOD xs x X*

lemma *igAbsRenSTR-imp-igAbsRen*:
igAbsRenSTR MOD ⟹ *igAbsRen MOD*
unfolding *igAbsRenSTR-def igAbsRen-def* **by** *simp*

lemma *igAbsRenSTR-imp-igAbsCongUSTR*:
igAbsRenSTR MOD ⟹ *igAbsCongUSTR MOD*
unfolding *igAbsCongUSTR-def igAbsRenSTR-def* **by** *metis*

Well-sorted fresh-swap models:

definition *iwlsFSw* **where**
iwlsFSw MOD ==
igWlsAllDisj MOD ∧ *igWlsAbsIsInBar MOD* ∧
igConsIPresIGWls MOD ∧ *igSwapAllIPresIGWlsAll MOD* ∧
igFreshCls MOD ∧ *igSwapCls MOD* ∧ *igAbsCongS MOD*

lemmas *iwlsFSw-defs1* = *iwlsFSw-def*
igWlsAllDisj-def igWlsAbsIsInBar-def
igConsIPresIGWls-def igSwapAllIPresIGWlsAll-def
igFreshCls-def igSwapCls-def igAbsCongS-def

lemmas *iwlsFSw-defs* = *iwlsFSw-def*
igWlsAllDisj-defs igWlsAbsIsInBar-def
igConsIPresIGWls-defs igSwapAllIPresIGWlsAll-defs
igFreshCls-defs igSwapCls-defs igAbsCongS-def

definition *iwlsFSwSTR* **where**
iwlsFSwSTR MOD ==
igWlsAllDisj MOD ∧ *igWlsAbsIsInBar MOD* ∧
igConsIPresIGWlsSTR MOD ∧ *igSwapAllIPresIGWlsAllSTR MOD* ∧
igFreshClsSTR MOD ∧ *igSwapClsSTR MOD* ∧ *igAbsCongSSTR MOD*

lemmas *iwlsFSwSTR-defs1 = iwlsFSwSTR-def*
igWlsAllDisj-def igWlsAbsIsInBar-def
igConsIPresIGWlsSTR-def igSwapAllIPresIGWlsAllSTR-def
igFreshClsSTR-def igSwapClsSTR-def igAbsCongSSTR-def

lemmas *iwlsFSwSTR-defs = iwlsFSwSTR-def*
igWlsAllDisj-defs igWlsAbsIsInBar-def
igConsIPresIGWlsSTR-defs igSwapAllIPresIGWlsAllSTR-defs
igFreshClsSTR-defs igSwapClsSTR-defs igAbsCongSSTR-def

lemma *iwlsFSwSTR-imp-iwlsFSw*:
iwlsFSwSTR MOD ⟹ *iwlsFSw MOD*
unfolding *iwlsFSwSTR-def iwlsFSw-def*
using
igConsIPresIGWlsSTR-imp-igConsIPresIGWls
igSwapAllIPresIGWlsAllSTR-imp-igSwapAllIPresIGWlsAll
igFreshClsSTR-imp-igFreshCls
igSwapClsSTR-imp-igSwapCls
igAbsCongSSTR-imp-igAbsCongS
by *auto*

Well-sorted fresh-subst models:

definition *iwlsFSb* **where**
iwlsFSb MOD ==
igWlsAllDisj MOD ∧ *igWlsAbsIsInBar MOD* ∧
igConsIPresIGWls MOD ∧ *igSubstAllIPresIGWlsAll MOD* ∧
igFreshCls MOD ∧ *igSubstCls MOD* ∧ *igAbsRen MOD*

lemmas *iwlsFSb-defs1 = iwlsFSb-def*
igWlsAllDisj-def igWlsAbsIsInBar-def
igConsIPresIGWls-def igSubstAllIPresIGWlsAll-def
igFreshCls-def igSubstCls-def igAbsRen-def

lemmas *iwlsFSb-defs = iwlsFSb-def*
igWlsAllDisj-defs igWlsAbsIsInBar-def
igConsIPresIGWls-defs igSubstAllIPresIGWlsAll-defs
igFreshCls-defs igSubstCls-defs igAbsRen-def

definition *iwlsFSbSwTR* **where**
iwlsFSbSwTR MOD ==
igWlsAllDisj MOD ∧ *igWlsAbsIsInBar MOD* ∧
igConsIPresIGWlsSTR MOD ∧ *igSubstAllIPresIGWlsAllSTR MOD* ∧

igFreshClsSTR MOD ∧ *igSubstClsSTR MOD* ∧ *igAbsRenSTR MOD*

lemmas *wlsFSbSwSTR-defs1* = *iwlsFSbSwTR-def*
igWlsAllDisj-def igWlsAbsIsInBar-def
igConsIPresIGWlsSTR-def igSwapAllIPresIGWlsAllSTR-def
igFreshClsSTR-def igSwapClsSTR-def igAbsRenSTR-def

lemmas *iwlsFSbSwTR-defs* = *iwlsFSbSwTR-def*
igWlsAllDisj-defs igWlsAbsIsInBar-def
igConsIPresIGWlsSTR-defs igSwapAllIPresIGWlsAllSTR-defs
igFreshClsSTR-defs igSwapClsSTR-defs igAbsRenSTR-def

lemma *iwlsFSbSwTR-imp-iwlsFSb*:
iwlsFSbSwTR MOD ⟹ *iwlsFSb MOD*
unfolding *iwlsFSbSwTR-def iwlsFSb-def*
using
igConsIPresIGWlsSTR-imp-igConsIPresIGWls
igSubstAllIPresIGWlsAllSTR-imp-igSubstAllIPresIGWlsAll
igFreshClsSTR-imp-igFreshCls
igSubstClsSTR-imp-igSubstCls
igAbsRenSTR-imp-igAbsRen
by *auto*

Well-sorted fresh-swap-subst-models

definition *iwlsFSwSb* **where**
iwlsFSwSb MOD ==
iwlsFSw MOD ∧ *igSubstAllIPresIGWlsAll MOD* ∧ *igSubstCls MOD*

lemmas *iwlsFSwSb-defs1* = *iwlsFSwSb-def*
iwlsFSw-def igSubstAllIPresIGWlsAll-def igSubstCls-def

lemmas *iwlsFSwSb-defs* = *iwlsFSwSb-def*
iwlsFSw-def igSubstAllIPresIGWlsAll-defs igSubstCls-defs

Well-sorted fresh-subst-swap-models

definition *iwlsFSbSw* **where**
iwlsFSbSw MOD ==
iwlsFSb MOD ∧ *igSwapAllIPresIGWlsAll MOD* ∧ *igSwapCls MOD*

lemmas *iwlsFSbSw-defs1* = *iwlsFSbSw-def*
iwlsFSw-def igSwapAllIPresIGWlsAll-def igSwapCls-def

lemmas *iwlsFSbSw-defs* = *iwlsFSbSw-def*
iwlsFSw-def igSwapAllIPresIGWlsAll-defs igSwapCls-defs

Extension of domain preservation (by swap and subst) to inputs:

First for free inputs:

definition *igSwapInpIPresIGWlsInp* **where**

igSwapInpIPresIGWlsInp MOD ==
∀ *zs z1 z2 delta inp.*
igWlsInp MOD delta inp ⟶
igWlsInp MOD delta (*igSwapInp MOD zs z1 z2 inp*)

definition *igSwapInpIPresIGWlsInpSTR* **where**
igSwapInpIPresIGWlsInpSTR MOD ==
∀ *zs z1 z2 delta inp.*
igWlsInp MOD delta (*igSwapInp MOD zs z1 z2 inp*) =
igWlsInp MOD delta inp

definition *igSubstInpIPresIGWlsInp* **where**
igSubstInpIPresIGWlsInp MOD ==
∀ *ys y Y delta inp.*
igWls MOD (*asSort ys*) *Y* ∧ *igWlsInp MOD delta inp* ⟶
igWlsInp MOD delta (*igSubstInp MOD ys Y y inp*)

definition *igSubstInpIPresIGWlsInpSTR* **where**
igSubstInpIPresIGWlsInpSTR MOD ==
∀ *ys y Y delta inp.*
igWls MOD (*asSort ys*) *Y* ⟶
igWlsInp MOD delta (*igSubstInp MOD ys Y y inp*) =
igWlsInp MOD delta inp

lemma *imp-igSwapInpIPresIGWlsInp*:
igSwapIPresIGWls MOD ⟹ *igSwapInpIPresIGWlsInp MOD*
by (*simp add*:
igSwapInpIPresIGWlsInp-def igWlsInp-def liftAll2-def
igSwapIPresIGWls-def igSwapAbsIPresIGWlsAbs-def igSwapInp-def lift-def
sameDom-def split: *option.splits*)

lemma *imp-igSwapInpIPresIGWlsInpSTR*:
igSwapIPresIGWlsSTR MOD ⟹ *igSwapInpIPresIGWlsInpSTR MOD*
by (*simp add*:
igSwapIPresIGWlsSTR-def igWlsInp-def liftAll2-def
igSwapIPresIGWls-def igSwapInpIPresIGWlsInpSTR-def igSwapInp-def lift-def
sameDom-def split: *option.splits*)
(*smt* (*verit*) *option.distinct*(*1*) *option.exhaust*)

lemma *imp-igSubstInpIPresIGWlsInp*:
igSubstIPresIGWls MOD ⟹ *igSubstInpIPresIGWlsInp MOD*
by (*simp add* : *igSubstInp-def*
igSubstIPresIGWls-def igSubstInpIPresIGWlsInp-def igWlsInp-def liftAll2-def
lift-def sameDom-def split: *option.splits*)

lemma *imp-igSubstInpIPresIGWlsInpSTR*:
igSubstIPresIGWlsSTR MOD ⟹ *igSubstInpIPresIGWlsInpSTR MOD*
by(*simp add*:
igSubstInpIPresIGWlsInpSTR-def igSubstIPresIGWlsSTR-def igSubstInp-def

igWlsInp-def liftAll2-def lift-def sameDom-def
split: *option.splits*) (*smt* (*verit*) *option.distinct*(*1*) *option.exhaust*)

Then for bound inputs:

definition *igSwapBinpIPresIGWlsBinp* **where**
igSwapBinpIPresIGWlsBinp MOD ==
∀ *zs z1 z2 delta binp*.
igWlsBinp MOD delta binp ⟶
igWlsBinp MOD delta (*igSwapBinp MOD zs z1 z2 binp*)

definition *igSwapBinpIPresIGWlsBinpSTR* **where**
igSwapBinpIPresIGWlsBinpSTR MOD ==
∀ *zs z1 z2 delta binp*.
igWlsBinp MOD delta (*igSwapBinp MOD zs z1 z2 binp*) =
igWlsBinp MOD delta binp

definition *igSubstBinpIPresIGWlsBinp* **where**
igSubstBinpIPresIGWlsBinp MOD ==
∀ *ys y Y delta binp*.
igWls MOD (*asSort ys*) *Y* ∧ *igWlsBinp MOD delta binp* ⟶
igWlsBinp MOD delta (*igSubstBinp MOD ys Y y binp*)

definition *igSubstBinpIPresIGWlsBinpSTR* **where**
igSubstBinpIPresIGWlsBinpSTR MOD ==
∀ *ys y Y delta binp*.
igWls MOD (*asSort ys*) *Y* ⟶
igWlsBinp MOD delta (*igSubstBinp MOD ys Y y binp*) =
igWlsBinp MOD delta binp

lemma *imp-igSwapBinpIPresIGWlsBinp*:
igSwapAbsIPresIGWlsAbs MOD ⟹ *igSwapBinpIPresIGWlsBinp MOD*
by(*auto simp add*:
igSwapBinpIPresIGWlsBinp-def igSwapAbsIPresIGWlsAbs-def igSwapBinp-def
igWlsBinp-def liftAll2-def lift-def sameDom-def
split: *option.splits*)

lemma *imp-igSwapBinpIPresIGWlsBinpSTR*:
igSwapAbsIPresIGWlsAbsSTR MOD ⟹ *igSwapBinpIPresIGWlsBinpSTR MOD*
by (*simp add*:
igSwapBinpIPresIGWlsBinpSTR-def igSwapAbsIPresIGWlsAbsSTR-def igSwapBinp-def
igWlsBinp-def liftAll2-def lift-def sameDom-def
split: *option.splits*) (*smt* (*verit*) *option.distinct*(*1*) *option.exhaust surj-pair*)

lemma *imp-igSubstBinpIPresIGWlsBinp*:
igSubstAbsIPresIGWlsAbs MOD ⟹ *igSubstBinpIPresIGWlsBinp MOD*
by (*auto simp add*:
igSubstBinpIPresIGWlsBinp-def igSubstAbsIPresIGWlsAbs-def igSubstBinp-def
igWlsBinp-def liftAll2-def lift-def sameDom-def
split: *option.splits*)

lemma *imp-igSubstBinpIPresIGWlsBinpSTR*:
igSubstAbsIPresIGWlsAbsSTR MOD $\Longrightarrow$ *igSubstBinpIPresIGWlsBinpSTR MOD*
by (*simp add*:
igSubstAbsIPresIGWlsAbsSTR-def igSubstBinpIPresIGWlsBinpSTR-def igSubstBinp-def igWlsBinp-def liftAll2-def lift-def sameDom-def
split: *option.splits*) (*smt* (*verit*) *option.distinct*(*1*) *option.exhaust surj-pair*)

8.2 Morphisms of models

The morphisms between models shall be the usual first-order-logic morphisms, i.e„ functions commuting with the operations and preserving the (freshness) relations. Because they involve the same signature, the morphisms for fresh-swap-subst models (called fresh-swap-subst morphisms) will be the same as those for fresh-subst-swap-models.

8.2.1 Preservation of the domains

definition *ipresIGWls* **where**
ipresIGWls h MOD MOD′ ==
$\forall$ *s X*. *igWls MOD s X* $\longrightarrow$ *igWls MOD′ s* (*h X*)

definition *ipresIGWlsAbs* **where**
ipresIGWlsAbs hA MOD MOD′ ==
$\forall$ *us s A*. *igWlsAbs MOD* (*us*,*s*) *A* $\longrightarrow$ *igWlsAbs MOD′* (*us*,*s*) (*hA A*)

definition *ipresIGWlsAll* **where**
ipresIGWlsAll h hA MOD MOD′ ==
ipresIGWls h MOD MOD′ $\wedge$ *ipresIGWlsAbs hA MOD MOD′*

lemmas *ipresIGWlsAll-defs* = *ipresIGWlsAll-def*
ipresIGWls-def ipresIGWlsAbs-def

8.2.2 Preservation of the constructs

definition *ipresIGVar* **where**
ipresIGVar h MOD MOD′ ==
$\forall$ *xs x*. *h* (*igVar MOD xs x*) = *igVar MOD′ xs x*

definition *ipresIGAbs* **where**
ipresIGAbs h hA MOD MOD′ ==
$\forall$ *xs x s X*. *isInBar* (*xs*,*s*) $\wedge$ *igWls MOD s X* $\longrightarrow$
hA (*igAbs MOD xs x X*) = *igAbs MOD′ xs x* (*h X*)

definition *ipresIGOp*
where
ipresIGOp h hA MOD MOD′ ==
$\forall$ *delta inp binp*.
igWlsInp MOD delta inp $\wedge$ *igWlsBinp MOD delta binp* $\longrightarrow$

h (*igOp MOD delta inp binp*) = *igOp MOD′ delta* (*lift h inp*) (*lift hA binp*)

definition *ipresIGCons* **where**
ipresIGCons h hA MOD MOD′ ==
ipresIGVar h MOD MOD′ ∧
ipresIGAbs h hA MOD MOD′ ∧
ipresIGOp h hA MOD MOD′

lemmas *ipresIGCons-defs* = *ipresIGCons-def*
ipresIGVar-def
ipresIGAbs-def
ipresIGOp-def

8.2.3 Preservation of freshness

definition *ipresIGFresh* **where**
ipresIGFresh h MOD MOD′ ==
∀ *ys y s X*.
igWls MOD s X ⟶
igFresh MOD ys y X ⟶ *igFresh MOD′ ys y* (*h X*)

definition *ipresIGFreshAbs* **where**
ipresIGFreshAbs hA MOD MOD′ ==
∀ *ys y us s A*.
igWlsAbs MOD (*us*,*s*) *A* ⟶
igFreshAbs MOD ys y A ⟶ *igFreshAbs MOD′ ys y* (*hA A*)

definition *ipresIGFreshAll* **where**
ipresIGFreshAll h hA MOD MOD′ ==
ipresIGFresh h MOD MOD′ ∧ *ipresIGFreshAbs hA MOD MOD′*

lemmas *ipresIGFreshAll-defs* = *ipresIGFreshAll-def*
ipresIGFresh-def ipresIGFreshAbs-def

8.2.4 Preservation of swapping

definition *ipresIGSwap* **where**
ipresIGSwap h MOD MOD′ ==
∀ *zs z1 z2 s X*.
igWls MOD s X ⟶
h (*igSwap MOD zs z1 z2 X*) = *igSwap MOD′ zs z1 z2* (*h X*)

definition *ipresIGSwapAbs* **where**
ipresIGSwapAbs hA MOD MOD′ ==
∀ *zs z1 z2 us s A*.
igWlsAbs MOD (*us*,*s*) *A* ⟶
hA (*igSwapAbs MOD zs z1 z2 A*) = *igSwapAbs MOD′ zs z1 z2* (*hA A*)

definition *ipresIGSwapAll* **where**
ipresIGSwapAll h hA MOD MOD′ ==

ipresIGSwap h MOD MOD′ ∧ *ipresIGSwapAbs hA MOD MOD′*

lemmas *ipresIGSwapAll-defs* = *ipresIGSwapAll-def*
ipresIGSwap-def ipresIGSwapAbs-def

8.2.5 Preservation of subst

definition *ipresIGSubst* **where**
ipresIGSubst h MOD MOD′ ==
∀ *ys Y y s X*.
  igWls MOD (*asSort ys*) *Y* ∧ *igWls MOD s X* ⟶
  h (*igSubst MOD ys Y y X*) = *igSubst MOD′ ys* (*h Y*) *y* (*h X*)

definition *ipresIGSubstAbs* **where**
ipresIGSubstAbs h hA MOD MOD′ ==
∀ *ys Y y us s A*.
  igWls MOD (*asSort ys*) *Y* ∧ *igWlsAbs MOD* (*us*,*s*) *A* ⟶
  hA (*igSubstAbs MOD ys Y y A*) = *igSubstAbs MOD′ ys* (*h Y*) *y* (*hA A*)

definition *ipresIGSubstAll* **where**
ipresIGSubstAll h hA MOD MOD′ ==
ipresIGSubst h MOD MOD′ ∧
ipresIGSubstAbs h hA MOD MOD′

lemmas *ipresIGSubstAll-defs* = *ipresIGSubstAll-def*
ipresIGSubst-def ipresIGSubstAbs-def

8.2.6 Fresh-swap morphisms

definition *FSwImorph* **where**
FSwImorph h hA MOD MOD′ ==
ipresIGWlsAll h hA MOD MOD′ ∧ *ipresIGCons h hA MOD MOD′* ∧
ipresIGFreshAll h hA MOD MOD′ ∧ *ipresIGSwapAll h hA MOD MOD′*

lemmas *FSwImorph-defs1* = *FSwImorph-def*
ipresIGWlsAll-def ipresIGCons-def
ipresIGFreshAll-def ipresIGSwapAll-def

lemmas *FSwImorph-defs* = *FSwImorph-def*
ipresIGWlsAll-defs ipresIGCons-defs
ipresIGFreshAll-defs ipresIGSwapAll-defs

8.2.7 Fresh-subst morphisms

definition *FSbImorph* **where**
FSbImorph h hA MOD MOD′ ==
ipresIGWlsAll h hA MOD MOD′ ∧ *ipresIGCons h hA MOD MOD′* ∧
ipresIGFreshAll h hA MOD MOD′ ∧ *ipresIGSubstAll h hA MOD MOD′*

lemmas *FSbImorph-defs1* = *FSbImorph-def*

ipresIGWlsAll-def ipresIGCons-def
ipresIGFreshAll-def ipresIGSubstAll-def

lemmas *FSbImorph-defs* = *FSbImorph-def*
ipresIGWlsAll-defs ipresIGCons-defs
ipresIGFreshAll-defs ipresIGSubstAll-defs

8.2.8 Fresh-swap-subst morphisms

definition *FSwSbImorph* **where**
FSwSbImorph h hA MOD MOD′ ==
FSwImorph h hA MOD MOD′ ∧ *ipresIGSubstAll h hA MOD MOD′*

lemmas *FSwSbImorph-defs1* = *FSwSbImorph-def*
FSwImorph-def ipresIGSubstAll-def

lemmas *FSwSbImorph-defs* = *FSwSbImorph-def*
FSwImorph-defs ipresIGSubstAll-defs

8.2.9 Basic facts

FSwSb morphisms are the same as FSbSw morphisms:

lemma *FSwSbImorph-iff*:
FSwSbImorph h hA MOD MOD′ =
(*FSbImorph h hA MOD MOD′* ∧ *ipresIGSwapAll h hA MOD MOD′*)
unfolding *FSwSbImorph-def FSbImorph-def FSwImorph-def* **by** *auto*

Some facts for free inpus:

lemma *igSwapInp-None*[*simp*]:
(*igSwapInp MOD zs z1 z2 inp i* = *None*) = (*inp i* = *None*)
unfolding *igSwapInp-def* **by**(*simp add*: *lift-None*)

lemma *igSubstInp-None*[*simp*]:
(*igSubstInp MOD ys Y y inp i* = *None*) = (*inp i* = *None*)
unfolding *igSubstInp-def* **by**(*simp add*: *lift-None*)

lemma *imp-igWlsInp*:
igWlsInp MOD delta inp ⟹ *ipresIGWls h MOD MOD′*
⟹ *igWlsInp MOD′ delta* (*lift h inp*)
by (*simp add*: *igWlsInp-def ipresIGWls-def liftAll2-def lift-def*
sameDom-def split: *option.splits*)

corollary *FSwImorph-igWlsInp*:
assumes *igWlsInp MOD delta inp* **and** *FSwImorph h hA MOD MOD′*
shows *igWlsInp MOD′ delta* (*lift h inp*)
using *assms* **unfolding** *FSwImorph-def ipresIGWlsAll-def*
using *imp-igWlsInp* **by** *auto*

corollary *FSbImorph-igWlsInp*:

assumes *igWlsInp MOD delta inp* **and** *FSbImorph h hA MOD MOD'*
shows *igWlsInp MOD' delta* (*lift h inp*)
using *assms* **unfolding** *FSbImorph-def ipresIGWlsAll-def*
using *imp-igWlsInp* **by** *auto*

lemma *FSwSbImorph-igWlsInp*:
assumes *igWlsInp MOD delta inp* **and** *FSwSbImorph h hA MOD MOD'*
shows *igWlsInp MOD' delta* (*lift h inp*)
using *assms* **unfolding** *FSwSbImorph-def* **using** *FSwImorph-igWlsInp* **by** *auto*

Similar facts for bound inpus:

lemma *igSwapBinp-None*[*simp*]:
(*igSwapBinp MOD zs z1 z2 binp i* = *None*) = (*binp i* = *None*)
unfolding *igSwapBinp-def* **by**(*simp add*: *lift-None*)

lemma *igSubstBinp-None*[*simp*]:
(*igSubstBinp MOD ys Y y binp i* = *None*) = (*binp i* = *None*)
unfolding *igSubstBinp-def* **by**(*simp add*: *lift-None*)

lemma *imp-igWlsBinp*:
assumes ∗: *igWlsBinp MOD delta binp*
and ∗∗: *ipresIGWlsAbs hA MOD MOD'*
shows *igWlsBinp MOD' delta* (*lift hA binp*)
using *assms* **by** (*simp add*: *igWlsBinp-def ipresIGWlsAbs-def liftAll2-def lift-def sameDom-def split*: *option.splits*)

corollary *FSwImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSwImorph h hA MOD MOD'*
shows *igWlsBinp MOD' delta* (*lift hA binp*)
using *assms* **unfolding** *FSwImorph-def ipresIGWlsAll-def*
using *imp-igWlsBinp* **by** *auto*

corollary *FSbImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSbImorph h hA MOD MOD'*
shows *igWlsBinp MOD' delta* (*lift hA binp*)
using *assms* **unfolding** *FSbImorph-def ipresIGWlsAll-def*
using *imp-igWlsBinp* **by** *auto*

lemma *FSwSbImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSwSbImorph h hA MOD MOD'*
shows *igWlsBinp MOD' delta* (*lift hA binp*)
using *assms* **unfolding** *FSwSbImorph-def* **using** *FSwImorph-igWlsBinp* **by** *auto*

lemmas *input-igSwap-igSubst-None* =
igSwapInp-None igSubstInp-None
igSwapBinp-None igSubstBinp-None

8.2.10 Identity and composition

lemma *id-FSwImorph*: *FSwImorph id id MOD MOD*
unfolding *FSwImorph-defs* **by** *auto*

lemma *id-FSbImorph*: *FSbImorph id id MOD MOD*
unfolding *FSbImorph-defs* **by** *auto*

lemma *id-FSwSbImorph*: *FSwSbImorph id id MOD MOD*
unfolding *FSwSbImorph-def* **apply**(*auto simp add*: *id-FSwImorph*)
unfolding *ipresIGSubstAll-defs* **by** *auto*

lemma *comp-ipresIGWls*:
assumes *ipresIGWls h MOD MOD'* **and** *ipresIGWls h' MOD' MOD''*
shows *ipresIGWls* (*h' o h*) *MOD MOD''*
using *assms* **unfolding** *ipresIGWls-def* **by** *auto*

lemma *comp-ipresIGWlsAbs*:
assumes *ipresIGWlsAbs hA MOD MOD'* **and** *ipresIGWlsAbs hA' MOD' MOD''*
shows *ipresIGWlsAbs* (*hA' o hA*) *MOD MOD''*
using *assms* **unfolding** *ipresIGWlsAbs-def* **by** *auto*

lemma *comp-ipresIGWlsAll*:
assumes *ipresIGWlsAll h hA MOD MOD'* **and** *ipresIGWlsAll h' hA' MOD' MOD''*
shows *ipresIGWlsAll* (*h' o h*) (*hA' o hA*) *MOD MOD''*
using *assms* **unfolding** *ipresIGWlsAll-def*
using *comp-ipresIGWls comp-ipresIGWlsAbs* **by** *auto*

lemma *comp-ipresIGVar*:
assumes *ipresIGVar h MOD MOD'* **and** *ipresIGVar h' MOD' MOD''*
shows *ipresIGVar* (*h' o h*) *MOD MOD''*
using *assms* **unfolding** *ipresIGVar-def* **by** *auto*

lemma *comp-ipresIGAbs*:
assumes *ipresIGWls h MOD MOD'*
and *ipresIGAbs h hA MOD MOD'* **and** *ipresIGAbs h' hA' MOD' MOD''*
shows *ipresIGAbs* (*h' o h*) (*hA' o hA*) *MOD MOD''*
using *assms* **unfolding** *ipresIGWls-def ipresIGAbs-def* **by** *fastforce*

lemma *comp-ipresIGOp*:
assumes *ipres*: *ipresIGWls h MOD MOD'* **and** *ipresAbs*: *ipresIGWlsAbs hA MOD MOD'*
and *h*: *ipresIGOp h hA MOD MOD'* **and** *h'*: *ipresIGOp h' hA' MOD' MOD''*
shows *ipresIGOp* (*h' o h*) (*hA' o hA*) *MOD MOD''*
using *assms* **by** (*auto simp*: *imp-igWlsInp imp-igWlsBinp ipresIGOp-def lift-comp*)

lemma *comp-ipresIGCons*:
assumes *ipresIGWlsAll h hA MOD MOD'*
and *ipresIGCons h hA MOD MOD'* **and** *ipresIGCons h' hA' MOD' MOD''*
shows *ipresIGCons* (*h' o h*) (*hA' o hA*) *MOD MOD''*

using *assms* **unfolding** *ipresIGWlsAll-def ipresIGCons-def*
using *comp-ipresIGVar comp-ipresIGAbs comp-ipresIGOp* **by** *auto*

lemma *comp-ipresIGFresh*:
assumes *ipresIGWls h MOD MOD′*
and *ipresIGFresh h MOD MOD′* **and** *ipresIGFresh h′ MOD′ MOD″*
shows *ipresIGFresh* (*h′ o h*) *MOD MOD″*
using *assms* **unfolding** *ipresIGWls-def ipresIGFresh-def* **by** *fastforce*

lemma *comp-ipresIGFreshAbs*:
assumes *ipresIGWlsAbs hA MOD MOD′*
and *ipresIGFreshAbs hA MOD MOD′* **and** *ipresIGFreshAbs hA′ MOD′ MOD″*
shows *ipresIGFreshAbs* (*hA′ o hA*) *MOD MOD″*
using *assms* **unfolding** *ipresIGWlsAbs-def ipresIGFreshAbs-def* **by** *fastforce*

lemma *comp-ipresIGFreshAll*:
assumes *ipresIGWlsAll h hA MOD MOD′*
and *ipresIGFreshAll h hA MOD MOD′* **and** *ipresIGFreshAll h′ hA′ MOD′ MOD″*
shows *ipresIGFreshAll* (*h′ o h*) (*hA′ o hA*) *MOD MOD″*
using *assms*
unfolding *ipresIGWlsAll-def ipresIGFreshAll-def*
using *comp-ipresIGFresh comp-ipresIGFreshAbs* **by** *auto*

lemma *comp-ipresIGSwap*:
assumes *ipresIGWls h MOD MOD′*
and *ipresIGSwap h MOD MOD′* **and** *ipresIGSwap h′ MOD′ MOD″*
shows *ipresIGSwap* (*h′ o h*) *MOD MOD″*
using *assms* **unfolding** *ipresIGWls-def ipresIGSwap-def* **by** *fastforce*

lemma *comp-ipresIGSwapAbs*:
assumes *ipresIGWlsAbs hA MOD MOD′*
and *ipresIGSwapAbs hA MOD MOD′* **and** *ipresIGSwapAbs hA′ MOD′ MOD″*
shows *ipresIGSwapAbs* (*hA′ o hA*) *MOD MOD″*
using *assms* **unfolding** *ipresIGWlsAbs-def ipresIGSwapAbs-def* **by** *fastforce*

lemma *comp-ipresIGSwapAll*:
assumes *ipresIGWlsAll h hA MOD MOD′*
and *ipresIGSwapAll h hA MOD MOD′* **and** *ipresIGSwapAll h′ hA′ MOD′ MOD″*
shows *ipresIGSwapAll* (*h′ o h*) (*hA′ o hA*) *MOD MOD″*
using *assms*
unfolding *ipresIGWlsAll-def ipresIGSwapAll-def*
using *comp-ipresIGSwap comp-ipresIGSwapAbs* **by** *auto*

lemma *comp-ipresIGSubst*:
assumes *ipresIGWls h MOD MOD′*
and *ipresIGSubst h MOD MOD′* **and** *ipresIGSubst h′ MOD′ MOD″*
shows *ipresIGSubst* (*h′ o h*) *MOD MOD″*
using *assms* **unfolding** *ipresIGWls-def ipresIGSubst-def*
apply *auto* **by** *blast*

lemma *comp-ipresIGSubstAbs*:
assumes ∗: *igWlsAbsIsInBar MOD*
and *h*: *ipresIGWls h MOD MOD′* **and** *hA*: *ipresIGWlsAbs hA MOD MOD′*
and *hhA*: *ipresIGSubstAbs h hA MOD MOD′* **and** *h′hA′*: *ipresIGSubstAbs h′ hA′ MOD′ MOD′′*
shows *ipresIGSubstAbs* (*h′ o h*) (*hA′ o hA*) *MOD MOD′′*
using *assms* **by**(*fastforce simp*: *igWlsAbsIsInBar-def ipresIGSubstAbs-def ipresIGWls-def ipresIGWlsAbs-def*)

lemma *comp-ipresIGSubstAll*:
assumes *igWlsAbsIsInBar MOD*
and *ipresIGWlsAll h hA MOD MOD′*
and *ipresIGSubstAll h hA MOD MOD′* **and** *ipresIGSubstAll h′ hA′ MOD′ MOD′′*
shows *ipresIGSubstAll* (*h′ o h*) (*hA′ o hA*) *MOD MOD′′*
using *assms* **unfolding** *ipresIGWlsAll-def ipresIGSubstAll-def*
using *comp-ipresIGSubst comp-ipresIGSubstAbs* **by** *auto*

lemma *comp-FSwImorph*:
assumes ∗: *FSwImorph h hA MOD MOD′* **and** ∗∗: *FSwImorph h′ hA′ MOD′ MOD′′*
shows *FSwImorph* (*h′ o h*) (*hA′ o hA*) *MOD MOD′′*
using *assms* **unfolding** *FSwImorph-def*
using *comp-ipresIGWlsAll comp-ipresIGCons comp-ipresIGFreshAll comp-ipresIGSwapAll* **by** *auto*

lemma *comp-FSbImorph*:
assumes *igWlsAbsIsInBar MOD*
and *FSbImorph h hA MOD MOD′* **and** *FSbImorph h′ hA′ MOD′ MOD′′*
shows *FSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD MOD′′*
using *assms* **unfolding** *FSbImorph-def*
using *comp-ipresIGWlsAll comp-ipresIGCons comp-ipresIGFreshAll comp-ipresIGSubstAll* **by** *auto*

lemma *comp-FSwSbImorph*:
assumes *igWlsAbsIsInBar MOD*
and *FSwSbImorph h hA MOD MOD′* **and** *FSwSbImorph h′ hA′ MOD′ MOD′′*
shows *FSwSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD MOD′′*
using *assms* **unfolding** *FSwSbImorph-def*
using *comp-FSwImorph FSwImorph-def comp-ipresIGSubstAll FixSyn-axioms* **by** *blast*

8.3 The term model

We show that terms form fresh-swap-subst and fresh-subst-swap models.

8.3.1 Definitions and simplification rules

definition *termMOD* **where**

termMOD ==
(|*igWls* = *wls*, *igWlsAbs* = *wlsAbs*,
igVar = *Var*, *igAbs* = *Abs*, *igOp* = *Op*,
igFresh = *fresh*, *igFreshAbs* = *freshAbs*,
igSwap = *swap*, *igSwapAbs* = *swapAbs*,
igSubst = *subst*, *igSubstAbs* = *substAbs*|)

lemma *igWls-termMOD*[*simp*]: *igWls termMOD* = *wls*
unfolding *termMOD-def* **by** *simp*

lemma *igWlsAbs-termMOD*[*simp*]: *igWlsAbs termMOD* = *wlsAbs*
unfolding *termMOD-def* **by** *simp*

lemma *igWlsInp-termMOD-wlsInp*[*simp*]:
igWlsInp termMOD delta inp = *wlsInp delta inp*
unfolding *igWlsInp-def wlsInp-iff* **by** *simp*

lemma *igWlsBinp-termMOD-wlsBinp*[*simp*]:
igWlsBinp termMOD delta binp = *wlsBinp delta binp*
unfolding *igWlsBinp-def wlsBinp-iff* **by** *simp*

lemmas *igWlsAll-termMOD-simps* =
igWls-termMOD igWlsAbs-termMOD
igWlsInp-termMOD-wlsInp igWlsBinp-termMOD-wlsBinp

lemma *igVar-termMOD*[*simp*]: *igVar termMOD* = *Var*
unfolding *termMOD-def* **by** *simp*

lemma *igAbs-termMOD*[*simp*]: *igAbs termMOD* = *Abs*
unfolding *termMOD-def* **by** *simp*

lemma *igOp-termMOD*[*simp*]: *igOp termMOD* = *Op*
unfolding *termMOD-def* **by** *simp*

lemmas *igCons-termMOD-simps* =
igVar-termMOD igAbs-termMOD igOp-termMOD

lemma *igFresh-termMOD*[*simp*]: *igFresh termMOD* = *fresh*
unfolding *termMOD-def* **by** *simp*

lemma *igFreshAbs-termMOD*[*simp*]: *igFreshAbs termMOD* = *freshAbs*
unfolding *termMOD-def* **by** *simp*

lemma *igFreshInp-termMOD*[*simp*]: *igFreshInp termMOD* = *freshInp*
unfolding *igFreshInp-def*[*abs-def*] *freshInp-def*[*abs-def*] **by** *simp*

lemma *igFreshBinp-termMOD*[*simp*]: *igFreshBinp termMOD* = *freshBinp*
unfolding *igFreshBinp-def*[*abs-def*] *freshBinp-def*[*abs-def*] **by** *simp*

lemmas *igFreshAll-termMOD-simps* =
igFresh-termMOD igFreshAbs-termMOD
igFreshInp-termMOD igFreshBinp-termMOD

lemma *igSwap-termMOD*[*simp*]: *igSwap termMOD* = *swap*
unfolding *termMOD-def* **by** *simp*

lemma *igSwapAbs-termMOD*[*simp*]: *igSwapAbs termMOD* = *swapAbs*
unfolding *termMOD-def* **by** *simp*

lemma *igSwapInp-termMOD*[*simp*]: *igSwapInp termMOD* = *swapInp*
unfolding *igSwapInp-def*[*abs-def*] *swapInp-def*[*abs-def*] **by** *simp*

lemma *igSwapBinp-termMOD*[*simp*]: *igSwapBinp termMOD* = *swapBinp*
unfolding *igSwapBinp-def*[*abs-def*] *swapBinp-def*[*abs-def*] **by** *simp*

lemmas *igSwapAll-termMOD-simps* =
igSwap-termMOD igSwapAbs-termMOD
igSwapInp-termMOD igSwapBinp-termMOD

lemma *igSubst-termMOD*[*simp*]: *igSubst termMOD* = *subst*
unfolding *termMOD-def* **by** *simp*

lemma *igSubstAbs-termMOD*[*simp*]: *igSubstAbs termMOD* = *substAbs*
unfolding *termMOD-def* **by** *simp*

lemma *igSubstInp-termMOD*[*simp*]: *igSubstInp termMOD* = *substInp*
by (*simp add*: *igSubstInp-def*[*abs-def*] *substInp-def*[*abs-def*]
psubstInp-def[*abs-def*] *subst-def*)

lemma *igSubstBinp-termMOD*[*simp*]: *igSubstBinp termMOD* = *substBinp*
by (*simp add*: *igSubstBinp-def*[*abs-def*] *substBinp-def*[*abs-def*]
psubstBinp-def[*abs-def*] *substAbs-def*)

lemmas *igSubstAll-termMOD-simps* =
igSubst-termMOD igSubstAbs-termMOD
igSubstInp-termMOD igSubstBinp-termMOD

lemmas *structure-termMOD-simps* =
igWlsAll-termMOD-simps
igFreshAll-termMOD-simps
igSwapAll-termMOD-simps
igSubstAll-termMOD-simps

8.3.2 Well-sortedness of the term model

Domains are disjoint:

lemma *termMOD-igWlsDisj*: *igWlsDisj termMOD*
unfolding *igWlsDisj-def* **using** *wls-disjoint* **by** *auto*

lemma *termMOD-igWlsAbsDisj*: *igWlsAbsDisj termMOD*
unfolding *igWlsAbsDisj-def* **using** *wlsAbs-disjoint* **by** *auto*

lemma *termMOD-igWlsAllDisj*: *igWlsAllDisj termMOD*
unfolding *igWlsAllDisj-def*
using *termMOD-igWlsDisj termMOD-igWlsAbsDisj* **by** *simp*

Abstraction domains inhabited only within bound arities:

lemma *termMOD-igWlsAbsIsInBar*: *igWlsAbsIsInBar termMOD*
unfolding *igWlsAbsIsInBar-def* **using** *wlsAbs-nchotomy* **by** *simp*

The syntactic constructs preserve the domains:

lemma *termMOD-igVarIPresIGWls*: *igVarIPresIGWls termMOD*
unfolding *igVarIPresIGWls-def* **by** *simp*

lemma *termMOD-igAbsIPresIGWls*: *igAbsIPresIGWls termMOD*
unfolding *igAbsIPresIGWls-def* **by** *simp*

lemma *termMOD-igOpIPresIGWls*: *igOpIPresIGWls termMOD*
unfolding *igOpIPresIGWls-def* **by** *simp*

lemma *termMOD-igConsIPresIGWls*: *igConsIPresIGWls termMOD*
unfolding *igConsIPresIGWls-def*
using *termMOD-igVarIPresIGWls termMOD-igAbsIPresIGWls termMOD-igOpIPresIGWls*
by *auto*

Swap preserves the domains:

lemma *termMOD-igSwapIPresIGWls*: *igSwapIPresIGWls termMOD*
unfolding *igSwapIPresIGWls-def* **by** *simp*

lemma *termMOD-igSwapAbsIPresIGWlsAbs*: *igSwapAbsIPresIGWlsAbs termMOD*
unfolding *igSwapAbsIPresIGWlsAbs-def* **by** *simp*

lemma *termMOD-igSwapAllIPresIGWlsAll*: *igSwapAllIPresIGWlsAll termMOD*
unfolding *igSwapAllIPresIGWlsAll-def*
using *termMOD-igSwapIPresIGWls termMOD-igSwapAbsIPresIGWlsAbs* **by** *auto*

“Subst" preserves the domains:

lemma *termMOD-igSubstIPresIGWls*: *igSubstIPresIGWls termMOD*
unfolding *igSubstIPresIGWls-def* **by** *simp*

lemma *termMOD-igSubstAbsIPresIGWlsAbs*: *igSubstAbsIPresIGWlsAbs termMOD*
unfolding *igSubstAbsIPresIGWlsAbs-def* **by** *simp*

lemma *termMOD-igSubstAllIPresIGWlsAll*: *igSubstAllIPresIGWlsAll termMOD*
unfolding *igSubstAllIPresIGWlsAll-def*
using *termMOD-igSubstIPresIGWls termMOD-igSubstAbsIPresIGWlsAbs* **by** *auto*

The “fresh" clauses hold:

lemma *termMOD-igFreshIGVar*: *igFreshIGVar termMOD*
unfolding *igFreshIGVar-def* **by** *simp*

lemma *termMOD-igFreshIGAbs1*: *igFreshIGAbs1 termMOD*
unfolding *igFreshIGAbs1-def* **by** *auto*

lemma *termMOD-igFreshIGAbs2*: *igFreshIGAbs2 termMOD*
unfolding *igFreshIGAbs2-def* **by** *auto*

lemma *termMOD-igFreshIGOp*: *igFreshIGOp termMOD*
unfolding *igFreshIGOp-def* **by** *simp*

lemma *termMOD-igFreshCls*: *igFreshCls termMOD*
unfolding *igFreshCls-def*
using *termMOD-igFreshIGVar termMOD-igFreshIGAbs1 termMOD-igFreshIGAbs2 termMOD-igFreshIGOp*
by *simp*

The “swap" clauses hold:

lemma *termMOD-igSwapIGVar*: *igSwapIGVar termMOD*
unfolding *igSwapIGVar-def* **by** *simp*

lemma *termMOD-igSwapIGAbs*: *igSwapIGAbs termMOD*
unfolding *igSwapIGAbs-def* **by** *auto*

lemma *termMOD-igSwapIGOp*: *igSwapIGOp termMOD*
unfolding *igSwapIGOp-def* **by** *simp*

lemma *termMOD-igSwapCls*: *igSwapCls termMOD*
unfolding *igSwapCls-def*
using *termMOD-igSwapIGVar termMOD-igSwapIGAbs termMOD-igSwapIGOp* **by** *simp*

The “subst" clauses hold:

lemma *termMOD-igSubstIGVar1*: *igSubstIGVar1 termMOD*
unfolding *igSubstIGVar1-def* **by** *auto*

lemma *termMOD-igSubstIGVar2*: *igSubstIGVar2 termMOD*
unfolding *igSubstIGVar2-def* **by** *auto*

lemma *termMOD-igSubstIGAbs*: *igSubstIGAbs termMOD*
unfolding *igSubstIGAbs-def* **by** *auto*

lemma *termMOD-igSubstIGOp*: *igSubstIGOp termMOD*
unfolding *igSubstIGOp-def* **by** *simp*

lemma *termMOD-igSubstCls*: *igSubstCls termMOD*
unfolding *igSubstCls-def*

using *termMOD-igSubstIGVar1 termMOD-igSubstIGVar2*
termMOD-igSubstIGAbs termMOD-igSubstIGOp **by** *simp*

The swap-congruence clause for abstractions holds:

lemma *termMOD-igAbsCongS*: *igAbsCongS termMOD*
unfolding *igAbsCongS-def* **using** *wls-Abs-swap-cong*
by (*metis igAbs-termMOD igFresh-termMOD igSwap-termMOD igWls-termMOD*)

The subst-renaming clause for abstractions holds:

lemma *termMOD-igAbsRen*: *igAbsRen termMOD*
unfolding *igAbsRen-def* **by** *auto*

lemma *termMOD-iwlsFSw*: *iwlsFSw termMOD*
unfolding *iwlsFSw-def*
using
termMOD-igWlsAllDisj termMOD-igWlsAbsIsInBar
termMOD-igConsIPresIGWls termMOD-igSwapAllIPresIGWlsAll
termMOD-igFreshCls termMOD-igSwapCls termMOD-igAbsCongS
by *auto*

lemma *termMOD-iwlsFSb*: *iwlsFSb termMOD*
unfolding *iwlsFSb-def*
using
termMOD-igWlsAllDisj termMOD-igWlsAbsIsInBar
termMOD-igConsIPresIGWls termMOD-igSubstAllIPresIGWlsAll
termMOD-igFreshCls termMOD-igSubstCls termMOD-igAbsRen
by *auto*

lemma *termMOD-iwlsFSwSb*: *iwlsFSwSb termMOD*
unfolding *iwlsFSwSb-def*
using *termMOD-iwlsFSw termMOD-igSubstAllIPresIGWlsAll termMOD-igSubstCls*
by *simp*

lemma *termMOD-iwlsFSbSw*: *iwlsFSbSw termMOD*
unfolding *iwlsFSbSw-def*
using *termMOD-iwlsFSb termMOD-igSwapAllIPresIGWlsAll termMOD-igSwapCls*
by *simp*

8.3.3 Direct description of morphisms from the term models

definition *ipresWls* **where**
ipresWls h MOD ==
$\forall$ *s X. wls s X* $\longrightarrow$ *igWls MOD s* (*h X*)

lemma *ipresIGWls-termMOD*[*simp*]:
ipresIGWls h termMOD MOD = ipresWls h MOD
unfolding *ipresIGWls-def ipresWls-def* **by** *simp*

definition *ipresWlsAbs* **where**

ipresWlsAbs hA MOD ==
∀ *us s A. wlsAbs* (*us*,*s*) *A* ⟶ *igWlsAbs MOD* (*us*,*s*) (*hA A*)

lemma *ipresIGWlsAbs-termMOD*[*simp*]:
ipresIGWlsAbs hA termMOD MOD = *ipresWlsAbs hA MOD*
unfolding *ipresIGWlsAbs-def ipresWlsAbs-def* **by** *simp*

definition *ipresWlsAll* **where**
ipresWlsAll h hA MOD ==
ipresWls h MOD ∧ *ipresWlsAbs hA MOD*

lemmas *ipresWlsAll-defs* = *ipresWlsAll-def*
ipresWls-def ipresWlsAbs-def

lemma *ipresIGWlsAll-termMOD*[*simp*]:
ipresIGWlsAll h hA termMOD MOD = *ipresWlsAll h hA MOD*
unfolding *ipresIGWlsAll-def ipresWlsAll-def* **by** *simp*

lemmas *ipresIGWlsAll-termMOD-simps* =
ipresIGWls-termMOD ipresIGWlsAbs-termMOD ipresIGWlsAll-termMOD

definition *ipresVar* **where**
ipresVar h MOD ==
∀ *xs x. h* (*Var xs x*) = *igVar MOD xs x*

lemma *ipresIGVar-termMOD*[*simp*]:
ipresIGVar h termMOD MOD = *ipresVar h MOD*
unfolding *ipresIGVar-def ipresVar-def* **by** *simp*

definition *ipresAbs* **where**
ipresAbs h hA MOD ==
∀ *xs x s X. isInBar* (*xs*,*s*) ∧ *wls s X* ⟶ *hA* (*Abs xs x X*) = *igAbs MOD xs x* (*h*
X)

lemma *ipresIGAbs-termMOD*[*simp*]:
ipresIGAbs h hA termMOD MOD = *ipresAbs h hA MOD*
unfolding *ipresIGAbs-def ipresAbs-def* **by** *simp*

definition *ipresOp* **where**
ipresOp h hA MOD ==
∀ *delta inp binp.*
wlsInp delta inp ∧ *wlsBinp delta binp* ⟶
h (*Op delta inp binp*) =
igOp MOD delta (*lift h inp*) (*lift hA binp*)

lemma *ipresIGOp-termMOD*[*simp*]:
ipresIGOp h hA termMOD MOD = *ipresOp h hA MOD*
unfolding *ipresIGOp-def ipresOp-def* **by** *simp*

definition *ipresCons* **where**
ipresCons h hA MOD ==
ipresVar h MOD ∧
ipresAbs h hA MOD ∧
ipresOp h hA MOD

lemmas *ipresCons-defs = ipresCons-def*
ipresVar-def
ipresAbs-def
ipresOp-def

lemma *ipresIGCons-termMOD*[*simp*]:
ipresIGCons h hA termMOD MOD = ipresCons h hA MOD
unfolding *ipresIGCons-def ipresCons-def* **by** *simp*

lemmas *ipresIGCons-termMOD-simps =*
ipresIGVar-termMOD ipresIGAbs-termMOD ipresIGOp-termMOD
ipresIGCons-termMOD

definition *ipresFresh* **where**
ipresFresh h MOD ==
∀ *ys y s X*.
wls s X ⟶
fresh ys y X ⟶ *igFresh MOD ys y* (*h X*)

lemma *ipresIGFresh-termMOD*[*simp*]:
ipresIGFresh h termMOD MOD = ipresFresh h MOD
unfolding *ipresIGFresh-def ipresFresh-def* **by** *simp*

definition *ipresFreshAbs* **where**
ipresFreshAbs hA MOD ==
∀ *ys y us s A*.
wlsAbs (*us*,*s*) *A* ⟶
freshAbs ys y A ⟶ *igFreshAbs MOD ys y* (*hA A*)

lemma *ipresIGFreshAbs-termMOD*[*simp*]:
ipresIGFreshAbs hA termMOD MOD = ipresFreshAbs hA MOD
unfolding *ipresIGFreshAbs-def ipresFreshAbs-def* **by** *simp*

definition *ipresFreshAll* **where**
ipresFreshAll h hA MOD ==
ipresFresh h MOD ∧ *ipresFreshAbs hA MOD*

lemmas *ipresFreshAll-defs = ipresFreshAll-def*
ipresFresh-def ipresFreshAbs-def

lemma *ipresIGFreshAll-termMOD*[*simp*]:
ipresIGFreshAll h hA termMOD MOD = ipresFreshAll h hA MOD
unfolding *ipresIGFreshAll-def ipresFreshAll-def* **by** *simp*

lemmas *ipresIGFreshAll-termMOD-simps* =
ipresIGFresh-termMOD ipresIGFreshAbs-termMOD ipresIGFreshAll-termMOD

definition *ipresSwap* **where**
ipresSwap h MOD ==
∀ *zs z1 z2 s X*.
  wls s X ⟶
  h (*X* #[*z1* ∧ *z2*]-*zs*) = *igSwap MOD zs z1 z2* (*h X*)

lemma *ipresIGSwap-termMOD*[*simp*]:
ipresIGSwap h termMOD MOD = *ipresSwap h MOD*
unfolding *ipresIGSwap-def ipresSwap-def* **by** *simp*

definition *ipresSwapAbs* **where**
ipresSwapAbs hA MOD ==
∀ *zs z1 z2 us s A*.
  wlsAbs (*us*,*s*) *A* ⟶
  hA (*A* $[*z1* ∧ *z2*]-*zs*) = *igSwapAbs MOD zs z1 z2* (*hA A*)

lemma *ipresIGSwapAbs-termMOD*[*simp*]:
ipresIGSwapAbs hA termMOD MOD = *ipresSwapAbs hA MOD*
unfolding *ipresIGSwapAbs-def ipresSwapAbs-def* **by** *simp*

definition *ipresSwapAll* **where**
ipresSwapAll h hA MOD ==
 ipresSwap h MOD ∧ *ipresSwapAbs hA MOD*

lemmas *ipresSwapAll-defs* = *ipresSwapAll-def*
ipresSwap-def ipresSwapAbs-def

lemma *ipresIGSwapAll-termMOD*[*simp*]:
ipresIGSwapAll h hA termMOD MOD = *ipresSwapAll h hA MOD*
unfolding *ipresIGSwapAll-def ipresSwapAll-def* **by** *simp*

lemmas *ipresIGSwapAll-termMOD-simps* =
ipresIGSwap-termMOD ipresIGSwapAbs-termMOD ipresIGSwapAll-termMOD

definition *ipresSubst* **where**
ipresSubst h MOD ==
∀ *ys Y y s X*.
  wls (*asSort ys*) *Y* ∧ *wls s X* ⟶
  h (*subst ys Y y X*) = *igSubst MOD ys* (*h Y*) *y* (*h X*)

lemma *ipresIGSubst-termMOD*[*simp*]:
ipresIGSubst h termMOD MOD = *ipresSubst h MOD*
unfolding *ipresIGSubst-def ipresSubst-def* **by** *simp*

definition *ipresSubstAbs* **where**

ipresSubstAbs h hA MOD ==
∀ *ys Y y us s A*.
wls (*asSort ys*) *Y* ∧ *wlsAbs* (*us*,*s*) *A* ⟶
hA (*A* $[*Y* / *y*]-*ys*) = *igSubstAbs MOD ys* (*h Y*) *y* (*hA A*)

lemma *ipresIGSubstAbs-termMOD*[*simp*]:
ipresIGSubstAbs h hA termMOD MOD = *ipresSubstAbs h hA MOD*
unfolding *ipresIGSubstAbs-def ipresSubstAbs-def* **by** *simp*

definition *ipresSubstAll* **where**
ipresSubstAll h hA MOD ==
ipresSubst h MOD ∧ *ipresSubstAbs h hA MOD*

lemmas *ipresSubstAll-defs* = *ipresSubstAll-def*
ipresSubst-def ipresSubstAbs-def

lemma *ipresIGSubstAll-termMOD*[*simp*]:
ipresIGSubstAll h hA termMOD MOD = *ipresSubstAll h hA MOD*
unfolding *ipresIGSubstAll-def ipresSubstAll-def* **by** *simp*

lemmas *ipresIGSubstAll-termMOD-simps* =
ipresIGSubst-termMOD ipresIGSubstAbs-termMOD ipresIGSubstAll-termMOD

definition *termFSwImorph* **where**
termFSwImorph h hA MOD ==
ipresWlsAll h hA MOD ∧ *ipresCons h hA MOD* ∧
ipresFreshAll h hA MOD ∧ *ipresSwapAll h hA MOD*

lemmas *termFSwImorph-defs1* = *termFSwImorph-def*
ipresWlsAll-def ipresCons-def
ipresFreshAll-def ipresSwapAll-def

lemmas *termFSwImorph-defs* = *termFSwImorph-def*
ipresWlsAll-defs ipresCons-defs
ipresFreshAll-defs ipresSwapAll-defs

lemma *FSwImorph-termMOD*[*simp*]:
FSwImorph h hA termMOD MOD = *termFSwImorph h hA MOD*
unfolding *FSwImorph-def termFSwImorph-def* **by** *simp*

definition *termFSbImorph* **where**
termFSbImorph h hA MOD ==
ipresWlsAll h hA MOD ∧ *ipresCons h hA MOD* ∧
ipresFreshAll h hA MOD ∧ *ipresSubstAll h hA MOD*

lemmas *termFSbImorph-defs1* = *termFSbImorph-def*
ipresWlsAll-def ipresCons-def
ipresFreshAll-def ipresSubstAll-def

lemmas *termFSbImorph-defs* = *termFSbImorph-def*
ipresWlsAll-defs ipresCons-defs
ipresFreshAll-defs ipresSubstAll-defs

lemma *FSbImorph-termMOD*[*simp*]:
FSbImorph h hA termMOD MOD = *termFSbImorph h hA MOD*
unfolding *FSbImorph-def termFSbImorph-def* **by** *simp*

definition *termFSwSbImorph* **where**
termFSwSbImorph h hA MOD ==
termFSwImorph h hA MOD ∧ *ipresSubstAll h hA MOD*

lemmas *termFSwSbImorph-defs1* = *termFSwSbImorph-def*
termFSwImorph-def ipresSubstAll-def

lemmas *termFSwSbImorph-defs* = *termFSwSbImorph-def*
termFSwImorph-defs ipresSubstAll-defs

Term FSwSb morphisms are the same as FSbSw morphisms:

lemma *termFSwSbImorph-iff*:
termFSwSbImorph h hA MOD =
(*termFSbImorph h hA MOD* ∧ *ipresSwapAll h hA MOD*)
unfolding *termFSwSbImorph-def termFSwImorph-def termFSbImorph-def ipres-SubstAll-def*
unfolding *FSwSbImorph-def FSbImorph-def FSwImorph-def* **by** *auto*

lemma *FSwSbImorph-termMOD*[*simp*]:
FSwSbImorph h hA termMOD MOD = *termFSwSbImorph h hA MOD*
unfolding *FSwSbImorph-def termFSwSbImorph-def* **by** *simp*

lemma *ipresWls-wlsInp*:
assumes *wlsInp delta inp* **and** *ipresWls h MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
using *assms imp-igWlsInp*[*of termMOD delta inp h MOD*] **by** *auto*

lemma *termFSwImorph-wlsInp*:
assumes *wlsInp delta inp* **and** *termFSwImorph h hA MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
using *assms FSwImorph-igWlsInp*[*of termMOD delta inp h hA MOD*] **by** *auto*

lemma *termFSwSbImorph-wlsInp*:
assumes *wlsInp delta inp* **and** *termFSwSbImorph h hA MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
using *assms FSwSbImorph-igWlsInp*[*of termMOD delta inp h hA MOD*] **by** *auto*

lemma *ipresWls-wlsBinp*:
assumes *wlsBinp delta binp* **and** *ipresWlsAbs hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
using *assms imp-igWlsBinp*[*of termMOD delta binp hA MOD*] **by** *auto*

lemma *termFSwImorph-wlsBinp*:
assumes *wlsBinp delta binp* **and** *termFSwImorph h hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
using *assms FSwImorph-igWlsBinp*[*of termMOD delta binp h hA MOD*] **by** *auto*

lemma *termFSwSbImorph-wlsBinp*:
assumes *wlsBinp delta binp* **and** *termFSwSbImorph h hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
using *assms FSwSbImorph-igWlsBinp*[*of termMOD delta binp h hA MOD*] **by** *auto*

lemma *id-termFSwImorph*: *termFSwImorph id id termMOD*
using *id-FSwImorph*[*of termMOD*] **by** *simp*

lemma *id-termFSbImorph*: *termFSbImorph id id termMOD*
using *id-FSbImorph*[*of termMOD*] **by** *simp*

lemma *id-termFSwSbImorph*: *termFSwSbImorph id id termMOD*
using *id-FSwSbImorph*[*of termMOD*] **by** *simp*

lemma *comp-termFSwImorph*:
assumes ∗: *termFSwImorph h hA MOD* **and** ∗∗: *FSwImorph h′ hA′ MOD MOD′*
shows *termFSwImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
using *assms comp-FSwImorph*[*of h hA termMOD MOD h′ hA′ MOD′*] **by** *auto*

lemma *comp-termFSbImorph*:
assumes ∗: *termFSbImorph h hA MOD* **and** ∗∗: *FSbImorph h′ hA′ MOD MOD′*
shows *termFSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
using *assms comp-FSbImorph*[*of termMOD h hA MOD h′ hA′ MOD′*]
termMOD-igWlsAbsIsInBar **by** *auto*

lemma *comp-termFSwSbImorph*:
assumes ∗: *termFSwSbImorph h hA MOD* **and** ∗∗: *FSwSbImorph h′ hA′ MOD MOD′*
shows *termFSwSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
using *assms comp-FSwSbImorph*[*of termMOD h hA MOD h′ hA′ MOD′*]
termMOD-igWlsAbsIsInBar **by** *auto*

lemmas *mapFrom-termMOD-simps* =
ipresIGWlsAll-termMOD-simps
ipresIGCons-termMOD-simps
ipresIGFreshAll-termMOD-simps
ipresIGSwapAll-termMOD-simps
ipresIGSubstAll-termMOD-simps
FSwImorph-termMOD FSbImorph-termMOD FSwSbImorph-termMOD

lemmas *termMOD-simps* =
structure-termMOD-simps mapFrom-termMOD-simps

8.3.4 Sufficient criteria for being a morphism to a well-sorted model (of various kinds)

In a nutshell: in these cases, we only need to check preservation of the syntactic constructs, "ipresCons".

lemma *ipresCons-imp-ipresWlsAll*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igConsIPresIGWls MOD*
shows *ipresWlsAll h hA MOD*
proof−
  {**fix** *s X us s′ A*
   have (*wls s X* ⟶ *igWls MOD s* (*h X*)) ∧
        (*wlsAbs* (*us*,*s′*) *A* ⟶ *igWlsAbs MOD* (*us*,*s′*) (*hA A*))
   proof(*induction rule*: *wls-rawInduct*)
    case (*Var xs x*)
    then show *?case*
     by (*metis assms igConsIPresIGWls-def igVarIPresIGWls-def ipresCons-def ipresVar-def*)
   next
    case (*Op delta inp binp*)
    have *igWlsInp MOD delta* (*lift h inp*) ∧ *igWlsBinp MOD delta* (*lift hA binp*)
    using *Op* **unfolding** *igWlsInp-def igWlsBinp-def wlsInp-iff wlsBinp-iff*
    by *simp* (*simp add*: *liftAll2-def lift-def split*: *option.splits*)
    hence *igWls MOD* (*stOf delta*) (*igOp MOD delta* (*lift h inp*) (*lift hA binp*))
    using ∗∗ **unfolding** *igConsIPresIGWls-def igOpIPresIGWls-def* **by** *simp*
    thus *?case* **using** *Op* ∗ **unfolding** *ipresCons-def ipresOp-def* **by** *simp*
   next
    case (*Abs s xs x X*)
    then show *?case*
     by (*metis assms igAbsIPresIGWls-def igConsIPresIGWls-def ipresAbs-def ipresCons-def*)
   qed
  }
  thus *?thesis* **unfolding** *ipresWlsAll-defs* **by** *simp*
qed

lemma *ipresCons-imp-ipresFreshAll*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igFreshCls MOD*
and *igConsIPresIGWls MOD*
shows *ipresFreshAll h hA MOD*
proof−
  have ∗∗∗: *ipresWlsAll h hA MOD*
  using *assms ipresCons-imp-ipresWlsAll* **by** *auto*
  hence ∗∗∗∗:
  ⋀ *delta inp. wlsInp delta inp* ⟹ *igWlsInp MOD delta* (*lift h inp*)
  ⋀ *delta binp. wlsBinp delta binp* ⟹ *igWlsBinp MOD delta* (*lift hA binp*)
  unfolding *ipresWlsAll-def* **using** *ipresWls-wlsInp ipresWls-wlsBinp* **by** *auto*

  {**fix** *s X us s′ A ys y*
   have (*wls s X* ⟶ *fresh ys y X* ⟶ *igFresh MOD ys y* (*h X*)) ∧

(*wlsAbs* (*us*,*s*′) *A* ⟶ *freshAbs ys y A* ⟶ *igFreshAbs MOD ys y* (*hA A*))
proof(*induction rule*: *wls-rawInduct*)
case (*Var xs x*)
then show *?case*
by (*metis* ∗ ∗∗ *fresh-Var-simp igFreshCls-def igFreshIGVar-def ipresCons-def ipresVar-def*)
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
assume *y-fresh*: *fresh ys y* (*Op delta inp binp*)
{**fix** *i X* **assume** *inp*: *inp i* = *Some X*
then obtain *s* **where** *arOf delta i* = *Some s*
using *Op* **unfolding** *wlsInp-iff sameDom-def* **by** *fastforce*
hence *igFresh MOD ys y* (*h X*)
using *Op.IH y-fresh inp* **unfolding** *freshInp-def liftAll-def liftAll2-def*
by (*metis freshInp-def liftAll-def wls-fresh-Op-simp*)
}
moreover
{**fix** *i A* **assume** *binp*: *binp i* = *Some A*
then obtain *us-s* **where** *barOf delta i* = *Some us-s*
using *Op* **unfolding** *wlsBinp-iff sameDom-def* **by** *force*
hence *igFreshAbs MOD ys y* (*hA A*)
using *Op.IH y-fresh binp* **unfolding** *freshBinp-def liftAll-def liftAll2-def*
by *simp* (*metis* (*no-types*, *opaque-lifting*) *freshBinp-def liftAll-def old.prod.exhaust*)
}
ultimately have *igFreshInp MOD ys y* (*lift h inp*) ∧ *igFreshBinp MOD ys y* (*lift hA binp*)
unfolding *igFreshInp-def igFreshBinp-def liftAll-lift-comp* **unfolding** *liftAll-def* **by** *auto*
moreover have *igWlsInp MOD delta* (*lift h inp*) ∧ *igWlsBinp MOD delta* (*lift hA binp*)
using *Op* ∗∗∗∗ **by** *simp*
ultimately have *igFresh MOD ys y* (*igOp MOD delta* (*lift h inp*) (*lift hA binp*))
using ∗∗ **unfolding** *igFreshCls-def igFreshIGOp-def* **by** *simp*
thus *igFresh MOD ys y* (*h* (*Op delta inp binp*))
using *Op* ∗ **unfolding** *ipresCons-def ipresOp-def* **by** *simp*
qed
next
case (*Abs s xs x X*)
hence *hX-wls*: *igWls MOD s* (*h X*)
using ∗∗∗ **unfolding** *ipresWlsAll-def ipresWls-def* **by** *simp*
thus *?case*
using *Abs assms* **by** (*cases ys* = *xs* ∧ *y* = *x*)
(*simp-all add*: *igFreshCls-def igFreshIGAbs1-def igFreshIGAbs2-def ipresAbs-def ipresCons-def*)
qed
}
thus *?thesis* **unfolding** *ipresFreshAll-defs* **by** *auto*

qed

lemma *ipresCons-imp-ipresSwapAll*:
assumes $*$: *ipresCons h hA MOD* **and** $**$: *igSwapCls MOD*
and *igConsIPresIGWls MOD*
shows *ipresSwapAll h hA MOD*
proof−
  have $***$: *ipresWlsAll h hA MOD*
  using *assms ipresCons-imp-ipresWlsAll* **by** *auto*
  hence $****$:
  $\bigwedge$ *delta inp. wlsInp delta inp* $\Longrightarrow$ *igWlsInp MOD delta* (*lift h inp*)
  $\bigwedge$ *delta binp. wlsBinp delta binp* $\Longrightarrow$ *igWlsBinp MOD delta* (*lift hA binp*)
  unfolding *ipresWlsAll-def* **using** *ipresWls-wlsInp ipresWls-wlsBinp* **by** *auto*

  {**fix** *s X us s' A zs z1 z2*
   have (*wls s X* $\longrightarrow$ *h* (*swap zs z1 z2 X*) = *igSwap MOD zs z1 z2* (*h X*)) $\wedge$
        (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *hA* (*swapAbs zs z1 z2 A*) = *igSwapAbs MOD zs z1 z2*
(*hA A*))
   proof(*induction rule*: *wls-rawInduct*)
     case (*Var xs x*)
     then show *?case*
       by (*metis* $*$ $**$ *igSwapCls-def igSwapIGVar-def ipresCons-def ipresVar-def*
swap-Var-simp)
   next
     case (*Op delta inp binp*)
     let *?inpsw* = *swapInp zs z1 z2 inp*   **let** *?binpsw* = *swapBinp zs z1 z2 binp*
     let *?Left* = *h* (*Op delta ?inpsw ?binpsw*)
     let *?Right* = *igSwap MOD zs z1 z2* (*h* (*Op delta inp binp*))
     have *wlsLiftInp*:
     igWlsInp MOD delta (*lift h inp*) $\wedge$ *igWlsBinp MOD delta* (*lift hA binp*)
     using *Op* $****$ **by** *simp*
     have *wlsInp delta ?inpsw* $\wedge$ *wlsBinp delta ?binpsw*
     using *Op* **by** *simp*
     hence *?Left* = *igOp MOD delta* (*lift h ?inpsw*) (*lift hA ?binpsw*)
     using $*$ **unfolding** *ipresCons-def ipresOp-def* **by** *simp*
     moreover
     have *lift h ?inpsw* = *igSwapInp MOD zs z1 z2* (*lift h inp*) $\wedge$
        lift hA ?binpsw = *igSwapBinp MOD zs z1 z2* (*lift hA binp*)
     using *Op* $*$ *not-None-eq*
     by (*simp add*: *igSwapCls-def igSwapIGOp-def wlsInp-iff wlsBinp-iff*
     swapInp-def swapBinp-def igSwapInp-def igSwapBinp-def
     lift-comp fun-eq-iff liftAll2-def lift-def sameDom-def split: *option.splits*)
     (*metis not-None-eq old.prod.exhaust*)
     moreover
     have *igOp MOD delta* (*igSwapInp MOD zs z1 z2* (*lift h inp*))
                 (*igSwapBinp MOD zs z1 z2* (*lift hA binp*)) =
        igSwap MOD zs z1 z2 (*igOp MOD delta* (*lift h inp*) (*lift hA binp*))
     using *wlsLiftInp* $**$ **unfolding** *igSwapCls-def igSwapIGOp-def* **by** *simp*
     moreover

have *igSwap MOD zs z1 z2* (*igOp MOD delta* (*lift h inp*) (*lift hA binp*)) = *?Right*
using *Op* ∗ **unfolding** *ipresCons-def ipresOp-def* **by** *simp*
ultimately have *?Left* = *?Right* **by** *simp*
then show *?case* **by** (*simp add*: *Op*)
next
case (*Abs s xs x X*)
let *?Xsw* = *swap zs z1 z2 X* **let** *?xsw* = *x* @*xs*[*z1* ∧ *z2*]-*zs*
have *hX*: *igWls MOD s* (*h X*) **using** *Abs.IH* ∗∗∗ **unfolding** *ipresWlsAll-def ipresWls-def* **by** *simp*
let *?Left* = *hA* (*Abs xs ?xsw ?Xsw*)
let *?Right* = *igSwapAbs MOD zs z1 z2* (*hA* (*Abs xs x X*))
have *wls s* (*swap zs z1 z2 X*) **using** *Abs* **by** *simp*
hence *?Left* = *igAbs MOD xs ?xsw* (*h ?Xsw*)
using *Abs* ∗ **unfolding** *ipresCons-def ipresAbs-def* **by** *blast*
also note *Abs*(*3*)
also have *igAbs MOD xs ?xsw* (*igSwap MOD zs z1 z2* (*h X*)) =
igSwapAbs MOD zs z1 z2 (*igAbs MOD xs x* (*h X*))
using *Abs hX* ∗∗ **by** (*auto simp*: *igSwapCls-def igSwapIGAbs-def*)
also have . . . = *?Right* **using** *Abs* ∗ **by** (*auto simp*: *ipresCons-def ipresAbs-def*)
finally have *?Left* = *?Right* **.**
then show *?case* **using** *Abs*(*2*) **by** *auto*
qed
}
thus *?thesis* **unfolding** *ipresSwapAll-defs* **by** *auto*
qed

lemma *ipresCons-imp-ipresSubstAll-aux*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
assumes *P*: *wlsPar P*
shows
(*wls s X* ⟶
(∀ *ys y Y*. *y* ∈ *varsOfS P ys* ∧ *Y* ∈ *termsOfS P* (*asSort ys*) ⟶
h (*X* #[*Y* / *y*]-*ys*) = *igSubst MOD ys* (*h Y*) *y* (*h X*)))
∧
(*wlsAbs* (*us*,*s*′) *A* ⟶
(∀ *ys y Y*. *y* ∈ *varsOfS P ys* ∧ *Y* ∈ *termsOfS P* (*asSort ys*) ⟶
hA (*A* $[*Y* / *y*]-*ys*) = *igSubstAbs MOD ys* (*h Y*) *y* (*hA A*)))
proof−
have ∗∗∗: *ipresWlsAll h hA MOD*
using *assms ipresCons-imp-ipresWlsAll* **by** *auto*
hence ∗∗∗∗:
⋀ *delta inp*. *wlsInp delta inp* ⟹ *igWlsInp MOD delta* (*lift h inp*)
⋀ *delta binp*. *wlsBinp delta binp* ⟹ *igWlsBinp MOD delta* (*lift hA binp*)
unfolding *ipresWlsAll-def* **using** *ipresWls-wlsInp ipresWls-wlsBinp* **by** *auto*
have ∗∗∗∗∗: *ipresFreshAll h hA MOD*
using *assms ipresCons-imp-ipresFreshAll* **by** *auto*

show *?thesis*
proof(*induction rule*: *wls-induct-fresh*[*of P*])
case *Par*
then show *?case* **using** *P* **by** *auto*
next
case (*Var xs x*)
then show *?case* **using** *assms*
by (*simp add*: *ipresWlsAll-def ipresWls-def igSubstCls-def igSubstIGVar2-def ipresCons-def ipresVar-def*)
(*metis* ∗∗∗ *FixSyn.ipresWlsAll-defs*(*1*) *FixSyn.ipresWlsAll-defs*(*2*) *FixSyn-axioms*

igSubstIGVar1-def wlsPar-def wls-subst-Var-simp1 wls-subst-Var-simp2)
next
case (*Op delta inp binp*)
show *?case* **proof** *safe*
fix *ys y Y*
assume *yP*: *y* ∈ *varsOfS P ys* **and** *YP*: *Y* ∈ *termsOfS P* (*asSort ys*)
hence *Y*: *wls* (*asSort ys*) *Y* **using** *P* **by** *auto*
hence *hY*: *igWls MOD* (*asSort ys*) (*h Y*)
using ∗∗∗ **unfolding** *ipresWlsAll-def ipresWls-def* **by** *simp*
have *sinp*: *wlsInp delta* (*substInp ys Y y inp*) ∧
wlsBinp delta (*substBinp ys Y y binp*) **using** *Y Op* **by** *simp*
have *liftInp*: *igWlsInp MOD delta* (*lift h inp*) ∧
igWlsBinp MOD delta (*lift hA binp*)
using *Op* ∗∗∗∗ **by** *simp*
let *?Left* = *h* ((*Op delta inp binp*) #[*Y* / *y*]-*ys*)
let *?Right* = *igSubst MOD ys* (*h Y*) *y* (*h* (*Op delta inp binp*))
have *?Left* = *igOp MOD delta* (*lift h* (*substInp ys Y y inp*))
(*lift hA* (*substBinp ys Y y binp*))
using *sinp* ∗ **unfolding** *ipresCons-def ipresOp-def*
by (*simp add*: *Op.IH*(*1*) *Op.IH*(*2*) *Y*)
moreover
have *lift h* (*substInp ys Y y inp*) = *igSubstInp MOD ys* (*h Y*) *y* (*lift h inp*) ∧
lift hA (*substBinp ys Y y binp*) = *igSubstBinp MOD ys* (*h Y*) *y* (*lift hA binp*)
using *Op YP yP* **by** (*simp add*: *substInp-def2 igSubstInp-def substBinp-def2 igSubstBinp-def lift-comp*
lift-def liftAll2-def fun-eq-iff wlsInp-iff wlsBinp-iff sameDom-def split: *option.splits*)
(*metis* (*no-types*, *opaque-lifting*) *not-Some-eq option.distinct*(*1*) *sinp wlsBinp.simps*)
moreover
have *igOp MOD delta* (*igSubstInp MOD ys* (*h Y*) *y* (*lift h inp*))
(*igSubstBinp MOD ys* (*h Y*) *y* (*lift hA binp*)) =
igSubst MOD ys (*h Y*) *y* (*igOp MOD delta* (*lift h inp*) (*lift hA binp*))
using *hY liftInp* ∗∗ **unfolding** *igSubstCls-def igSubstIGOp-def* **by** *simp*
moreover have . . . = *?Right* **using** *Op* ∗ **unfolding** *ipresCons-def ipresOp-def* **by** *simp*
ultimately show *?Left* = *?Right* **by** *simp*

qed
next
case (*Abs s xs x X*)
show *?case* **proof** *safe*
fix *ys y Y*
assume *yP*: $y \in$ *varsOfS P ys* **and** *YP*: $Y \in$ *termsOfS P* (*asSort ys*)
hence *x-diff*: $ys \neq xs \vee y \neq x$
and *Y*: *wls* (*asSort ys*) *Y* **and** *x-fresh*: *fresh xs x Y* **using** *P Abs* **by** *auto*
hence *hY*: *igWls MOD* (*asSort ys*) (*h Y*)
using $***$ **unfolding** *ipresWlsAll-def ipresWls-def* **by** *simp*
have *hX*: *igWls MOD s* (*h X*)
using *Abs* $***$ **unfolding** *ipresWlsAll-def ipresWls-def* **by** *simp*
let *?Xsb* = *subst ys Y y X*
have *Xsb*: *wls s ?Xsb* **using** *Y Abs* **by** *simp*
have *x-igFresh*: *igFresh MOD xs x* (*h Y*)
using *Y x-fresh* $*****$ **unfolding** *ipresFreshAll-def ipresFresh-def* **by** *simp*
let *?Left* = *hA* (*Abs xs x X* $[*Y* / *y*]-*ys*)
let *?Right* = *igSubstAbs MOD ys* (*h Y*) *y* (*hA* (*Abs xs x X*))
have *?Left* = *hA* (*Abs xs x ?Xsb*) **using** *Y Abs x-diff x-fresh* **by** *auto*
also have ... = *igAbs MOD xs x* (*h ?Xsb*)
using *Abs Xsb* $*$ **unfolding** *ipresCons-def ipresAbs-def* **by** *fastforce*
also have ... = *igAbs MOD xs x* (*igSubst MOD ys* (*h Y*) *y* (*h X*))
using *yP YP Abs.IH* **by** *simp*
also have ... = *igSubstAbs MOD ys* (*h Y*) *y* (*igAbs MOD xs x* (*h X*))
using *Abs hY hX x-diff x-igFresh* $**$
by (*auto simp*: *igSubstCls-def igSubstIGAbs-def*)
also have ... = *?Right* **using** *Abs* $*$ **by** (*auto simp*: *ipresCons-def ipresAbs-def*)

finally show *?Left* = *?Right* **.**
qed
qed
qed

lemma *ipresCons-imp-ipresSubst*:
assumes $*$: *ipresCons h hA MOD* **and** $**$: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubst h MOD*
unfolding *ipresSubst-def* **apply** *clarify*
subgoal for *ys Y y s X*
using *assms ipresCons-imp-ipresSubstAll-aux*
[*of h hA MOD*
ParS (λ*zs. if zs = ys then* [*y*] *else* [])
($\lambda s'$. *if* s' = *asSort ys then* [*Y*] *else* [])
(λ-. [])
[]]
unfolding *wlsPar-def* **by** *auto* **.**

lemma *ipresCons-imp-ipresSubstAbs*:
assumes $*$: *ipresCons h hA MOD* **and** $**$: *igSubstCls MOD*

and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubstAbs h hA MOD*
unfolding *ipresSubstAbs-def* **apply** *clarify*
subgoal for *ys Y y us s A*
using *assms ipresCons-imp-ipresSubstAll-aux*
[*of h hA MOD*
ParS (λ*zs. if zs = ys then* [*y*] *else* [])
(λ*s′. if s′ = asSort ys then* [*Y*] *else* [])
(λ-. [])
[]]
unfolding *wlsPar-def* **by** *auto* **.**

lemma *ipresCons-imp-ipresSubstAll*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubstAll h hA MOD*
unfolding *ipresSubstAll-def* **using** *assms*
ipresCons-imp-ipresSubst ipresCons-imp-ipresSubstAbs **by** *auto*

lemma *iwlsFSw-termFSwImorph-iff*:
iwlsFSw MOD $\Longrightarrow$ *termFSwImorph h hA MOD = ipresCons h hA MOD*
unfolding *iwlsFSw-def termFSwImorph-def*
using *ipresCons-imp-ipresWlsAll*
ipresCons-imp-ipresFreshAll ipresCons-imp-ipresSwapAll **by** *auto*

corollary *iwlsFSwSTR-termFSwImorph-iff*:
iwlsFSwSTR MOD $\Longrightarrow$ *termFSwImorph h hA MOD = ipresCons h hA MOD*
using *iwlsFSwSTR-imp-iwlsFSw iwlsFSw-termFSwImorph-iff* **by** *fastforce*

lemma *iwlsFSb-termFSbImorph-iff*:
iwlsFSb MOD $\Longrightarrow$ *termFSbImorph h hA MOD = ipresCons h hA MOD*
unfolding *iwlsFSb-def termFSbImorph-def*
using *ipresCons-imp-ipresWlsAll*
ipresCons-imp-ipresFreshAll ipresCons-imp-ipresSubstAll
unfolding *igSubstCls-def* **by** *fastforce+*

corollary *iwlsFSbSwTR-termFSbImorph-iff*:
iwlsFSbSwTR MOD $\Longrightarrow$ *termFSbImorph h hA MOD = ipresCons h hA MOD*
using *iwlsFSbSwTR-imp-iwlsFSb iwlsFSb-termFSbImorph-iff* **by** *fastforce*

lemma *iwlsFSwSb-termFSwSbImorph-iff*:
iwlsFSwSb MOD $\Longrightarrow$ *termFSwSbImorph h hA MOD = ipresCons h hA MOD*
unfolding *termFSwSbImorph-def iwlsFSwSb-def*
apply(*simp add*: *iwlsFSw-termFSwImorph-iff*)
unfolding *iwlsFSw-def* **using** *ipresCons-imp-ipresSubstAll* **by** *auto*

lemma *iwlsFSbSw-termFSwSbImorph-iff*:
iwlsFSbSw MOD $\Longrightarrow$ *termFSwSbImorph h hA MOD = ipresCons h hA MOD*
unfolding *termFSwSbImorph-iff iwlsFSbSw-def*

apply(*simp add*: *iwlsFSb-termFSbImorph-iff*)
unfolding *iwlsFSb-def* **using** *ipresCons-imp-ipresSwapAll* **by** *auto*

end

8.4 The "error" model of associated to a model

The error model will have the operators act like the original ones on well-formed terms, except that will return "ERR" (error) or "True" (in the case of fresh) whenever one of the inputs (variables, terms or abstractions) is "ERR" or is not well-formed.

The error model is more convenient than the original one, since one can define more easily a map from the model of terms to the former. This map shall be defined by the universal property of quotients, via a map from quasi-terms whose kernel includes the alpha-equivalence relation. The latter property (of including the alpha-equivalence would not be achievable with the original model as tariget, since alpha is defined unsortedly and the model clauses hold sortedly.

We shall only need error models associated to fresh-swap and to fresh-subst models.

8.4.1 Preliminaries

datatype *'a withERR* = *ERR* | *OK 'a*

context *FixSyn*
begin

definition *OKI* **where**
OKI inp = *lift OK inp*

definition *check* **where**
check eX == *THE X*. *eX* = *OK X*

definition *checkI* **where**
checkI einp == *lift check einp*

lemma *check-ex-unique*:
$eX \neq ERR \Longrightarrow (EX!\ X.\ eX = OK\ X)$
by(*cases eX*, *auto*)

lemma *check-OK*[*simp*]:
check (*OK X*) = *X*
unfolding *check-def* **using** *check-ex-unique theI'* **by** *auto*

lemma *OK-check*[*simp*]:

$eX \neq ERR \Longrightarrow OK\ (check\ eX) = eX$
unfolding *check-def* **using** *check-ex-unique theI'* **by** *auto*

lemma *checkI-OKI*[*simp*]:
$checkI\ (OKI\ inp) = inp$
unfolding *OKI-def checkI-def lift-def* **apply**(*rule ext*)
by(*case-tac inp i, auto*)

lemma *OKI-checkI*[*simp*]:
assumes $liftAll\ (\lambda\ X.\ X \neq ERR)\ einp$
shows $OKI\ (checkI\ einp) = einp$
unfolding *OKI-def checkI-def lift-def* **apply**(*rule ext*)
using *assms* **unfolding** *liftAll-def* **by** (*case-tac einp i, auto*)

lemma *OKI-inj*[*simp*]:
fixes *inp inp'* :: (*'index,'gTerm*)*input*
shows $(OKI\ inp = OKI\ inp') = (inp = inp')$
apply(*auto*) **unfolding** *OKI-def*
using *lift-preserves-inj*[*of OK*]
unfolding *inj-on-def* **by** *auto*

lemmas *OK-OKI-simps* =
check-OK OK-check checkI-OKI OKI-checkI OKI-inj

8.4.2 Definitions and notations

definition *errMOD* ::
(*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model* ⇒
(*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm withERR,'gAbs withERR*)*model*
where
errMOD MOD ==
(|*igWls* = λ *s eX. case eX of ERR* ⇒ *False* | *OK X* ⇒ *igWls MOD s X*,
igWlsAbs = λ (*us,s*) *eA. case eA of ERR* ⇒ *False* | *OK A* ⇒ *igWlsAbs MOD* (*us,s*) *A*,

igVar = λ *xs x. OK* (*igVar MOD xs x*),
igAbs = λ*xs x eX.*
if ($eX \neq ERR \wedge (\exists\ s.$ *isInBar* (*xs,s*) $\wedge$ *igWls MOD s* (*check eX*)))
then OK (*igAbs MOD xs x* (*check eX*))
else ERR,
igOp = λ*delta einp ebinp.*
if liftAll (λ *X.* $X \neq ERR$) *einp* $\wedge$ *liftAll* (λ *A.* $A \neq ERR$) *ebinp*
$\wedge$ *igWlsInp MOD delta* (*checkI einp*) $\wedge$ *igWlsBinp MOD delta* (*checkI ebinp*)
then OK (*igOp MOD delta* (*checkI einp*) (*checkI ebinp*))
else ERR,
igFresh = λ*ys y eX.*
if $eX \neq ERR \wedge (\exists\ s.$ *igWls MOD s* (*check eX*))
then igFresh MOD ys y (*check eX*)

else True,
igFreshAbs = *λys y eA*.
if eA ≠ *ERR* ∧ (∃ *us s*. *igWlsAbs MOD* (*us*,*s*) (*check eA*))
then igFreshAbs MOD ys y (*check eA*)
else True,
igSwap = *λzs z1 z2 eX*.
if eX ≠ *ERR* ∧ (∃ *s*. *igWls MOD s* (*check eX*))
then OK (*igSwap MOD zs z1 z2* (*check eX*))
else ERR,
igSwapAbs = *λzs z1 z2 eA*.
if eA ≠ *ERR* ∧ (∃ *us s*. *igWlsAbs MOD* (*us*,*s*) (*check eA*))
then OK (*igSwapAbs MOD zs z1 z2* (*check eA*))
else ERR,
igSubst = *λys eY y eX*.
if eY ≠ *ERR* ∧ *igWls MOD* (*asSort ys*) (*check eY*)
∧ *eX* ≠ *ERR* ∧ (∃ *s*. *igWls MOD s* (*check eX*))
then OK (*igSubst MOD ys* (*check eY*) *y* (*check eX*))
else ERR,
igSubstAbs = *λys eY y eA*.
if eY ≠ *ERR* ∧ *igWls MOD* (*asSort ys*) (*check eY*)
∧ *eA* ≠ *ERR* ∧ (∃ *us s*. *igWlsAbs MOD* (*us*,*s*) (*check eA*))
then OK (*igSubstAbs MOD ys* (*check eY*) *y* (*check eA*))
else ERR
⦈

abbreviation *eWls* **where** *eWls MOD* == *igWls* (*errMOD MOD*)
abbreviation *eWlsAbs* **where** *eWlsAbs MOD* == *igWlsAbs* (*errMOD MOD*)
abbreviation *eWlsInp* **where** *eWlsInp MOD* == *igWlsInp* (*errMOD MOD*)
abbreviation *eWlsBinp* **where** *eWlsBinp MOD* == *igWlsBinp* (*errMOD MOD*)
abbreviation *eVar* **where** *eVar MOD* == *igVar* (*errMOD MOD*)
abbreviation *eAbs* **where** *eAbs MOD* == *igAbs* (*errMOD MOD*)
abbreviation *eOp* **where** *eOp MOD* == *igOp* (*errMOD MOD*)
abbreviation *eFresh* **where** *eFresh MOD* == *igFresh* (*errMOD MOD*)
abbreviation *eFreshAbs* **where** *eFreshAbs MOD* == *igFreshAbs* (*errMOD MOD*)
abbreviation *eFreshInp* **where** *eFreshInp MOD* == *igFreshInp* (*errMOD MOD*)
abbreviation *eFreshBinp* **where** *eFreshBinp MOD* == *igFreshBinp* (*errMOD MOD*)
abbreviation *eSwap* **where** *eSwap MOD* == *igSwap* (*errMOD MOD*)
abbreviation *eSwapAbs* **where** *eSwapAbs MOD* == *igSwapAbs* (*errMOD MOD*)
abbreviation *eSwapInp* **where** *eSwapInp MOD* == *igSwapInp* (*errMOD MOD*)
abbreviation *eSwapBinp* **where** *eSwapBinp MOD* == *igSwapBinp* (*errMOD MOD*)
abbreviation *eSubst* **where** *eSubst MOD* == *igSubst* (*errMOD MOD*)
abbreviation *eSubstAbs* **where** *eSubstAbs MOD* == *igSubstAbs* (*errMOD MOD*)
abbreviation *eSubstInp* **where** *eSubstInp MOD* == *igSubstInp* (*errMOD MOD*)
abbreviation *eSubstBinp* **where** *eSubstBinp MOD* == *igSubstBinp* (*errMOD MOD*)

8.4.3 Simplification rules

lemma *eWls-simp1* [*simp*]:
eWls MOD s (*OK X*) = *igWls MOD s X*
unfolding *errMOD-def* **by** *simp*

lemma *eWls-simp2* [*simp*]:
eWls MOD s ERR = *False*
unfolding *errMOD-def* **by** *simp*

lemma *eWlsAbs-simp1* [*simp*]:
eWlsAbs MOD (*us*,*s*) (*OK A*) = *igWlsAbs MOD* (*us*,*s*) *A*
unfolding *errMOD-def* **by** *simp*

lemma *eWlsAbs-simp2* [*simp*]:
eWlsAbs MOD (*us*,*s*) *ERR* = *False*
unfolding *errMOD-def* **by** *simp*

lemma *eWlsInp-simp1* [*simp*]:
eWlsInp MOD delta (*OKI inp*) = *igWlsInp MOD delta inp*
by (*fastforce simp*: *OKI-def sameDom-def liftAll2-def lift-def igWlsInp-def*
split: *option.splits*)

lemma *eWlsInp-simp2* [*simp*]:
¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) *einp* ⟹ ¬ *eWlsInp MOD delta einp*
by (*force simp*: *sameDom-def liftAll-def liftAll2-def lift-def igWlsInp-def*)

corollary *eWlsInp-simp3* [*simp*]:
¬ *eWlsInp MOD delta* (λ*i*. *Some ERR*)
by (*auto simp*: *liftAll-def*)

lemma *eWlsBinp-simp1* [*simp*]:
eWlsBinp MOD delta (*OKI binp*) = *igWlsBinp MOD delta binp*
by (*fastforce simp*: *OKI-def sameDom-def liftAll2-def lift-def igWlsBinp-def*
split: *option.splits*)

lemma *eWlsBinp-simp2* [*simp*]:
¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) *ebinp* ⟹ ¬ *eWlsBinp MOD delta ebinp*
by (*force simp*: *sameDom-def liftAll-def liftAll2-def lift-def igWlsBinp-def*)

corollary *eWlsBinp-simp3* [*simp*]:
¬ *eWlsBinp MOD delta* (λ*i*. *Some ERR*)
by (*auto simp*: *liftAll-def*)

lemmas *eWlsAll-simps* =
eWls-simp1 eWls-simp2
eWlsAbs-simp1 eWlsAbs-simp2
eWlsInp-simp1 eWlsInp-simp2 eWlsInp-simp3
eWlsBinp-simp1 eWlsBinp-simp2 eWlsBinp-simp3

lemma *eVar-simp*[*simp*]:
eVar MOD xs x = *OK* (*igVar MOD xs x*)
unfolding *errMOD-def* **by** *simp*

lemma *eAbs-simp1*[*simp*]:
⟦*isInBar* (*xs*,*s*); *igWls MOD s X*⟧ ⟹ *eAbs MOD xs x* (*OK X*) = *OK* (*igAbs MOD xs x X*)
unfolding *errMOD-def* **by** *auto*

lemma *eAbs-simp2*[*simp*]:
∀ *s*. ¬ (*isInBar* (*xs*,*s*) ∧ *igWls MOD s X*) ⟹ *eAbs MOD xs x* (*OK X*) = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eAbs-simp3*[*simp*]:
eAbs MOD xs x ERR = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eOp-simp1*[*simp*]:
assumes *igWlsInp MOD delta inp* **and** *igWlsBinp MOD delta binp*
shows *eOp MOD delta* (*OKI inp*) (*OKI binp*) = *OK* (*igOp MOD delta inp binp*)
unfolding *errMOD-def* **apply** *simp*
unfolding *liftAll-def OKI-def lift-def*
using *assms* **by** (*auto split*: *option.splits*)

lemma *eOp-simp2*[*simp*]:
assumes ¬ *igWlsInp MOD delta inp*
shows *eOp MOD delta* (*OKI inp*) *ebinp* = *ERR*
using *assms* **unfolding** *errMOD-def* **by** *auto*

lemma *eOp-simp3*[*simp*]:
assumes ¬ *igWlsBinp MOD delta binp*
shows *eOp MOD delta einp* (*OKI binp*) = *ERR*
using *assms* **unfolding** *errMOD-def* **by** *auto*

lemma *eOp-simp4*[*simp*]:
assumes ¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) *einp*
shows *eOp MOD delta einp ebinp* = *ERR*
using *assms* **unfolding** *errMOD-def* **by** *auto*

corollary *eOp-simp5*[*simp*]:
eOp MOD delta (λ*i*. *Some ERR*) *ebinp* = *ERR*
by (*auto simp*: *liftAll-def*)

lemma *eOp-simp6*[*simp*]:
assumes ¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) *ebinp*
shows *eOp MOD delta einp ebinp* = *ERR*
using *assms* **unfolding** *errMOD-def* **by** *auto*

corollary *eOp-simp7*[*simp*]:

eOp MOD delta einp (*λi. Some ERR*) = *ERR*
by (*auto simp*: *liftAll-def*)

lemmas *eCons-simps* =
eVar-simp
eAbs-simp1 eAbs-simp2 eAbs-simp3
eOp-simp1 eOp-simp2 eOp-simp3 eOp-simp4 eOp-simp5 eOp-simp6 eOp-simp7

lemma *eFresh-simp1*[*simp*]:
igWls MOD s X ⟹ *eFresh MOD ys y* (*OK X*) = *igFresh MOD ys y X*
unfolding *errMOD-def* **by** *auto*

lemma *eFresh-simp2*[*simp*]:
∀ *s*. ¬ *igWls MOD s X* ⟹ *eFresh MOD ys y* (*OK X*)
unfolding *errMOD-def* **by** *auto*

lemma *eFresh-simp3*[*simp*]:
eFresh MOD ys y ERR
unfolding *errMOD-def* **by** *auto*

lemma *eFreshAbs-simp1*[*simp*]:
igWlsAbs MOD (*us,s*) *A* ⟹ *eFreshAbs MOD ys y* (*OK A*) = *igFreshAbs MOD ys y A*
unfolding *errMOD-def* **by** *auto*

lemma *eFreshAbs-simp2*[*simp*]:
∀ *us s*. ¬ *igWlsAbs MOD* (*us,s*) *A* ⟹ *eFreshAbs MOD ys y* (*OK A*)
unfolding *errMOD-def* **by** *auto*

lemma *eFreshAbs-simp3*[*simp*]:
eFreshAbs MOD ys y ERR
unfolding *errMOD-def* **by** *auto*

lemma *eFreshInp-simp*[*simp*]:
igWlsInp MOD delta inp
⟹ *eFreshInp MOD ys y* (*OKI inp*) = *igFreshInp MOD ys y inp*
by (*force simp*: *igFreshInp-def OKI-def liftAll-lift-comp igWlsInp-defs intro*!: *liftAll-cong*)

lemma *eFreshBinp-simp*[*simp*]:
igWlsBinp MOD delta binp
⟹ *eFreshBinp MOD ys y* (*OKI binp*) = *igFreshBinp MOD ys y binp*
by (*force simp*: *igFreshBinp-def OKI-def liftAll-lift-comp igWlsBinp-defs intro*!: *liftAll-cong*)

lemmas *eFreshAll-simps* =
eFresh-simp1 eFresh-simp2 eFresh-simp3
eFreshAbs-simp1 eFreshAbs-simp2 eFreshAbs-simp3
eFreshInp-simp

eFreshBinp-simp

lemma *eSwap-simp1* [*simp*]:
igWls MOD s X
$\Longrightarrow$ *eSwap MOD zs z1 z2* (*OK X*) = *OK* (*igSwap MOD zs z1 z2 X*)
unfolding *errMOD-def* **by** *auto*

lemma *eSwap-simp2* [*simp*]:
$\forall$ *s*. $\neg$ *igWls MOD s X* $\Longrightarrow$ *eSwap MOD zs z1 z2* (*OK X*) = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSwap-simp3* [*simp*]:
eSwap MOD zs z1 z2 ERR = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSwapAbs-simp1* [*simp*]:
igWlsAbs MOD (*us*,*s*) *A*
$\Longrightarrow$ *eSwapAbs MOD zs z1 z2* (*OK A*) = *OK* (*igSwapAbs MOD zs z1 z2 A*)
unfolding *errMOD-def* **by** *auto*

lemma *eSwapAbs-simp2* [*simp*]:
$\forall$ *us s*. $\neg$ *igWlsAbs MOD* (*us*,*s*) *A* $\Longrightarrow$ *eSwapAbs MOD zs z1 z2* (*OK A*) = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSwapAbs-simp3* [*simp*]:
eSwapAbs MOD zs z1 z2 ERR = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSwapInp-simp1* [*simp*]:
igWlsInp MOD delta inp
$\Longrightarrow$ *eSwapInp MOD zs z1 z2* (*OKI inp*) = *OKI* (*igSwapInp MOD zs z1 z2 inp*)
by (*force simp*: *igSwapInp-def OKI-def lift-comp igWlsInp-defs intro*!: *lift-cong*)

lemma *eSwapInp-simp2* [*simp*]:
assumes $\neg$ *liftAll* (λ *eX*. *eX* $\neq$ *ERR*) *einp*
shows $\neg$ *liftAll* (λ *eX*. *eX* $\neq$ *ERR*) (*eSwapInp MOD zs z1 z2 einp*)
using *assms* **unfolding** *liftAll-def igSwapInp-def lift-def* **by** (*auto split*: *option.splits*)

lemma *eSwapBinp-simp1* [*simp*]:
igWlsBinp MOD delta binp
$\Longrightarrow$ *eSwapBinp MOD zs z1 z2* (*OKI binp*) = *OKI* (*igSwapBinp MOD zs z1 z2 binp*)
by (*force simp*: *igSwapBinp-def OKI-def lift-comp igWlsBinp-defs intro*!: *lift-cong*)

lemma *eSwapBinp-simp2* [*simp*]:
assumes $\neg$ *liftAll* (λ *eA*. *eA* $\neq$ *ERR*) *ebinp*
shows $\neg$ *liftAll* (λ *eA*. *eA* $\neq$ *ERR*) (*eSwapBinp MOD zs z1 z2 ebinp*)
using *assms* **unfolding** *liftAll-def igSwapBinp-def lift-def* **by** (*auto split*: *option.splits*)

lemmas *eSwapAll-simps* =
eSwap-simp1 eSwap-simp2 eSwap-simp3
eSwapAbs-simp1 eSwapAbs-simp2 eSwapAbs-simp3
eSwapInp-simp1 eSwapInp-simp2
eSwapBinp-simp1 eSwapBinp-simp2

lemma *eSubst-simp1*[*simp*]:
⟦*igWls MOD* (*asSort ys*) *Y*; *igWls MOD s X*⟧
⟹ *eSubst MOD ys* (*OK Y*) *y* (*OK X*) = *OK* (*igSubst MOD ys Y y X*)
unfolding *errMOD-def* **by** *auto*

lemma *eSubst-simp2*[*simp*]:
¬ *igWls MOD* (*asSort ys*) *Y* ⟹ *eSubst MOD ys* (*OK Y*) *y eX* = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubst-simp3*[*simp*]:
∀ *s*. ¬ *igWls MOD s X* ⟹ *eSubst MOD ys eY y* (*OK X*) = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubst-simp4*[*simp*]:
eSubst MOD ys eY y ERR = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubst-simp5*[*simp*]:
eSubst MOD ys ERR y eX = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubstAbs-simp1*[*simp*]:
⟦*igWls MOD* (*asSort ys*) *Y*; *igWlsAbs MOD* (*us*,*s*) *A*⟧
⟹ *eSubstAbs MOD ys* (*OK Y*) *y* (*OK A*) = *OK* (*igSubstAbs MOD ys Y y A*)
unfolding *errMOD-def* **by** *auto*

lemma *eSubstAbs-simp2*[*simp*]:
¬ *igWls MOD* (*asSort ys*) *Y* ⟹ *eSubstAbs MOD ys* (*OK Y*) *y eA* = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubstAbs-simp3*[*simp*]:
∀ *us s*. ¬ *igWlsAbs MOD* (*us*,*s*) *A* ⟹ *eSubstAbs MOD ys eY y* (*OK A*) = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubstAbs-simp4*[*simp*]:
eSubstAbs MOD ys eY y ERR = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubstAbs-simp5*[*simp*]:
eSubstAbs MOD ys ERR y eA = *ERR*
unfolding *errMOD-def* **by** *auto*

lemma *eSubstInp-simp1*[*simp*]:

⟦*igWls MOD* (*asSort ys*) *Y*; *igWlsInp MOD delta inp*⟧
$\Longrightarrow$ *eSubstInp MOD ys* (*OK Y*) *y* (*OKI inp*) = *OKI* (*igSubstInp MOD ys Y y inp*)
by (*force simp*: *igSubstInp-def OKI-def lift-comp igWlsInp-defs intro*!: *lift-cong*)

lemma *eSubstInp-simp2*[*simp*]:
assumes ¬ *liftAll* (λ*eX*. *eX* ≠ *ERR*) *einp*
shows ¬ *liftAll* (λ*eX*. *eX* ≠ *ERR*) (*eSubstInp MOD ys eY y einp*)
using *assms* **unfolding** *lift-def igSubstInp-def liftAll-def* **by** (*auto split*: *option.splits*)

lemma *eSubstInp-simp3*[*simp*]:
assumes ∗: ¬ *igWls MOD* (*asSort ys*) *Y* **and** ∗∗: ¬ *einp* = (λ *i*. *None*)
shows ¬ *liftAll* (λ*eX*. *eX* ≠ *ERR*) (*eSubstInp MOD ys* (*OK Y*) *y einp*)
using *assms* **by** (*auto simp*: *igSubstInp-def liftAll-lift-comp lift-def liftAll-def split*: *option.splits*)

lemma *eSubstInp-simp4*[*simp*]:
assumes ¬ *einp* = (λ *i*. *None*)
shows ¬ *liftAll* (λ*eX*. *eX* ≠ *ERR*) (*eSubstInp MOD ys ERR y einp*)
using *assms* **by** (*auto simp*: *igSubstInp-def liftAll-lift-comp lift-def liftAll-def split*: *option.splits*)

lemma *eSubstBinp-simp1*[*simp*]:
⟦*igWls MOD* (*asSort ys*) *Y*; *igWlsBinp MOD delta binp*⟧
$\Longrightarrow$ *eSubstBinp MOD ys* (*OK Y*) *y* (*OKI binp*) = *OKI* (*igSubstBinp MOD ys Y y binp*)
by (*force simp*: *igSubstBinp-def OKI-def lift-comp igWlsBinp-defs intro*!: *lift-cong*)

lemma *eSubstBinp-simp2*[*simp*]:
assumes ¬ *liftAll* (λ*eA*. *eA* ≠ *ERR*) *ebinp*
shows ¬ *liftAll* (λ*eA*. *eA* ≠ *ERR*) (*eSubstBinp MOD ys eY y ebinp*)
using *assms* **by** (*auto simp*: *igSubstBinp-def liftAll-lift-comp lift-def liftAll-def split*: *option.splits*)

lemma *eSubstBinp-simp3*[*simp*]:
assumes ∗: ¬ *igWls MOD* (*asSort ys*) *Y* **and** ∗∗: ¬ *ebinp* = (λ *i*. *None*)
shows ¬ *liftAll* (λ*eA*. *eA* ≠ *ERR*) (*eSubstBinp MOD ys* (*OK Y*) *y ebinp*)
using *assms* **by** (*auto simp*: *igSubstBinp-def liftAll-lift-comp lift-def liftAll-def split*: *option.splits*)

lemma *eSubstBinp-simp4*[*simp*]:
assumes ¬ *ebinp* = (λ *i*. *None*)
shows ¬ *liftAll* (λ*eA*. *eA* ≠ *ERR*) (*eSubstBinp MOD ys ERR y ebinp*)
using *assms* **by** (*auto simp*: *igSubstBinp-def liftAll-lift-comp lift-def liftAll-def split*: *option.splits*)

lemmas *eSubstAll-simps* =
eSubst-simp1 eSubst-simp2 eSubst-simp3 eSubst-simp4 eSubst-simp5
eSubstAbs-simp1 eSubstAbs-simp2 eSubstAbs-simp3 eSubstAbs-simp4 eSubstAbs-simp5

eSubstInp-simp1 eSubstInp-simp2 eSubstInp-simp3 eSubstInp-simp4
eSubstBinp-simp1 eSubstBinp-simp2 eSubstBinp-simp3 eSubstBinp-simp4

lemmas *error-model-simps* =
OK-OKI-simps
eWlsAll-simps
eCons-simps
eFreshAll-simps
eSwapAll-simps
eSubstAll-simps

8.4.4 Nchotomies

lemma *eWls-nchotomy*:
(∃ *X*. *eX* = *OK X* ∧ *igWls MOD s X*) ∨ ¬ *eWls MOD s eX*
unfolding *errMOD-def* **by**(*cases eX*) *auto*

lemma *eWlsAbs-nchotomy*:
(∃ *A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*) ∨ ¬ *eWlsAbs MOD* (*us*,*s*) *eA*
unfolding *errMOD-def* **by**(*cases eA*) *auto*

lemma *eAbs-nchotomy*:
((∃ *s X*. *eX* = *OK X* ∧ *isInBar* (*xs*,*s*) ∧ *igWls MOD s X*)) ∨ (*eAbs MOD xs x eX* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eOp-nchotomy*:
(∃ *inp binp*. *einp* = *OKI inp* ∧ *igWlsInp MOD delta inp* ∧
ebinp = *OKI binp* ∧ *igWlsBinp MOD delta binp*)
∨
(*eOp MOD delta einp ebinp* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OKI-checkI* **by** *force*

lemma *eFresh-nchotomy*:
(∃ *s X*. *eX* = *OK X* ∧ *igWls MOD s X*) ∨ *eFresh MOD ys y eX*
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eFreshAbs-nchotomy*:
(∃ *us s A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*)
∨ *eFreshAbs MOD ys y eA*
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eSwap-nchotomy*:
(∃ *s X*. *eX* = *OK X* ∧ *igWls MOD s X*) ∨
(*eSwap MOD zs z1 z2 eX* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eSwapAbs-nchotomy*:
(∃ *us s A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*) ∨

(*eSwapAbs MOD zs z1 z2 eA* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eSubst-nchotomy*:
(∃ *Y*. *eY* = *OK Y* ∧
igWls MOD (*asSort ys*) *Y*) ∧ (∃ *s X*. *eX* = *OK X* ∧ *igWls MOD s X*)
∨
(*eSubst MOD ys eY y eX* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

lemma *eSubstAbs-nchotomy*:
(∃ *Y*. *eY* = *OK Y* ∧ *igWls MOD* (*asSort ys*) *Y*) ∧
(∃ *us s A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*)
∨
(*eSubstAbs MOD ys eY y eA* = *ERR*)
unfolding *errMOD-def* **apply** *simp* **using** *OK-check* **by** *fastforce*

8.4.5 Inversion rules

lemma *eWls-invert*:
assumes *eWls MOD s eX*
shows ∃ *X*. *eX* = *OK X* ∧ *igWls MOD s X*
using *assms eWls-nchotomy* **by** *blast*

lemma *eWlsAbs-invert*:
assumes *eWlsAbs MOD* (*us*,*s*) *eA*
shows ∃ *A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*
using *assms eWlsAbs-nchotomy* **by** *blast*

lemma *eWlsInp-invert*:
assumes *eWlsInp MOD delta einp*
shows ∃ *inp*. *igWlsInp MOD delta inp* ∧ *einp* = *OKI inp*
proof
  let *?inp* = *checkI einp*
  have *wlsOpS delta* **using** *assms* **unfolding** *igWlsInp-def* **by** *simp*
  moreover have *sameDom* (*arOf delta*) *?inp*
  using *assms* **unfolding** *igWlsInp-def checkI-def* **by** *simp*
  moreover have *liftAll2* (*igWls MOD*) (*arOf delta*) *?inp*
  using *assms eWls-invert*
  by (*fastforce simp*: *igWlsInp-def checkI-def liftAll2-def lift-def sameDom-def*
  split: *option.splits*)
  ultimately have *igWlsInp MOD delta ?inp* **unfolding** *igWlsInp-def* **by** *simp*
  moreover
  {**have** *liftAll* ($\lambda eX.\ eX \neq ERR$) *einp*
   using *assms* **using** *eWlsInp-simp2* **by** *blast*
   hence *einp* = *OKI ?inp* **by** *simp*
  }
  ultimately show *igWlsInp MOD delta ?inp* ∧ *einp* = *OKI ?inp* **by** *simp*
qed

lemma *eWlsBinp-invert*:
assumes *eWlsBinp MOD delta ebinp*
shows ∃ *binp. igWlsBinp MOD delta binp* ∧ *ebinp* = *OKI binp*
proof
  let *?binp* = *checkI ebinp*
  have *wlsOpS delta* **using** *assms* **unfolding** *igWlsBinp-def* **by** *simp*
  moreover have *sameDom* (*barOf delta*) *?binp*
  using *assms* **unfolding** *igWlsBinp-def checkI-def* **by** *simp*
  moreover have *liftAll2* (*igWlsAbs MOD*) (*barOf delta*) *?binp*
  using *assms eWlsAbs-invert*
  by (*fastforce simp*: *igWlsBinp-def checkI-def liftAll2-def lift-def sameDom-def split*: *option.splits*)
  ultimately have *igWlsBinp MOD delta ?binp* **unfolding** *igWlsBinp-def* **by** *simp*
  moreover
  {have *liftAll* (λ*eA. eA* ≠ *ERR*) *ebinp*
   using *assms* **using** *eWlsBinp-simp2* **by** *blast*
   hence *ebinp* = *OKI ?binp* **by** *simp*
  }
  ultimately show *igWlsBinp MOD delta ?binp* ∧ *ebinp* = *OKI ?binp* **by** *simp*
qed

lemma *eAbs-invert*:
assumes *eAbs MOD xs x eX* = *OK A*
shows ∃ *s X*. *eX* = *OK X* ∧ *isInBar* (*xs*,*s*) ∧ *A* = *igAbs MOD xs x X* ∧ *igWls MOD s X*
proof−
  have *1*: *eAbs MOD xs x eX* ≠ *ERR* **using** *assms* **by** *auto*
  then obtain *s X* **where** ∗: *eX* = *OK X*
  and ∗∗: *isInBar* (*xs*,*s*) **and** ∗∗∗: *igWls MOD s X*
  using *eAbs-nchotomy*[*of eX*] **by** *fastforce*
  hence *eAbs MOD xs x eX* = *OK* (*igAbs MOD xs x X*) **by** *simp*
  thus *?thesis* **using** *assms* ∗ ∗∗ ∗∗∗ **by** *auto*
qed

lemma *eOp-invert*:
assumes *eOp MOD delta einp ebinp* = *OK X*
shows
∃ *inp binp. einp* = *OKI inp* ∧ *ebinp* = *OKI binp* ∧
            X = *igOp MOD delta inp binp* ∧
            igWlsInp MOD delta inp ∧ *igWlsBinp MOD delta binp*
proof−
  have *eOp MOD delta einp ebinp* ≠ *ERR* **using** *assms* **by** *auto*
  then obtain *inp binp* **where** ∗: *einp* = *OKI inp*   *ebinp* = *OKI binp*
  igWlsInp MOD delta inp   *igWlsBinp MOD delta binp*
  using *eOp-nchotomy* **by** *blast*
  hence *eOp MOD delta einp ebinp* = *OK* (*igOp MOD delta inp binp*) **by** *simp*
  thus *?thesis* **using** *assms* ∗ **by** *auto*
qed

lemma *eFresh-invert*:
assumes ¬ *eFresh MOD ys y eX*
shows ∃ *s X*. *eX* = *OK X* ∧ ¬ *igFresh MOD ys y X* ∧ *igWls MOD s X*
proof−
  obtain *s X* **where** ∗: *eX* = *OK X* **and** ∗∗: *igWls MOD s X*
  using *assms eFresh-nchotomy*[*of eX*] **by** *fastforce*
  hence *eFresh MOD ys y eX* = *igFresh MOD ys y X* **by** *simp*
  thus *?thesis* **using** *assms* ∗ ∗∗ **by** *auto*
qed

lemma *eFreshAbs-invert*:
assumes ¬ *eFreshAbs MOD ys y eA*
shows ∃ *us s A*. *eA* = *OK A* ∧ ¬ *igFreshAbs MOD ys y A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*
proof−
  obtain *us s A* **where** ∗: *eA* = *OK A* **and** ∗∗: *igWlsAbs MOD* (*us*,*s*) *A*
  using *assms eFreshAbs-nchotomy*[*of eA*] **by** *fastforce*
  hence *eFreshAbs MOD ys y eA* = *igFreshAbs MOD ys y A* **by** *simp*
  thus *?thesis* **using** *assms* ∗ ∗∗ **by** *auto*
qed

lemma *eSwap-invert*:
assumes *eSwap MOD zs z1 z2 eX* = *OK Y*
shows ∃ *s X*. *eX* = *OK X* ∧ *Y* = *igSwap MOD zs z1 z2 X* ∧ *igWls MOD s X*
proof−
  have *1*: *eSwap MOD zs z1 z2 eX* ≠ *ERR* **using** *assms* **by** *auto*
  then obtain *s X* **where** ∗: *eX* = *OK X* **and** ∗∗: *igWls MOD s X*
  using *eSwap-nchotomy*[*of eX*] **by** *fastforce*
  hence *eSwap MOD zs z1 z2 eX* = *OK* (*igSwap MOD zs z1 z2 X*) **by** *simp*
  thus *?thesis* **using** *assms* ∗ ∗∗ **by** *auto*
qed

lemma *eSwapAbs-invert*:
assumes *eSwapAbs MOD zs z1 z2 eA* = *OK B*
shows ∃ *us s A*. *eA* = *OK A* ∧ *B* = *igSwapAbs MOD zs z1 z2 A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*
proof−
  have *1*: *eSwapAbs MOD zs z1 z2 eA* ≠ *ERR* **using** *assms* **by** *auto*
  then obtain *us s A* **where** ∗: *eA* = *OK A* **and** ∗∗: *igWlsAbs MOD* (*us*,*s*) *A*
  using *eSwapAbs-nchotomy*[*of eA*] **by** *fastforce*
  hence *eSwapAbs MOD zs z1 z2 eA* = *OK* (*igSwapAbs MOD zs z1 z2 A*) **by** *simp*
  thus *?thesis* **using** *assms* ∗ ∗∗ **by** *auto*
qed

lemma *eSubst-invert*:
assumes *eSubst MOD ys eY y eX* = *OK Z*
shows
∃ *s X Y*. *eY* = *OK Y* ∧ *eX* = *OK X* ∧ *igWls MOD s X* ∧ *igWls MOD* (*asSort*

ys) *Y* ∧
Z = *igSubst MOD ys Y y X*
proof−
have *1*: *eSubst MOD ys eY y eX* ≠ *ERR* **using** *assms* **by** *auto*
then obtain *s X Y* **where** ∗: *eX* = *OK X* *eY* = *OK Y*
igWls MOD s X *igWls MOD* (*asSort ys*) *Y*
using *eSubst-nchotomy*[*of eY - - eX*] **by** *fastforce*
hence *eSubst MOD ys eY y eX* = *OK* (*igSubst MOD ys Y y X*) **by** *simp*
thus *?thesis* **using** *assms* ∗ **by** *auto*
qed

lemma *eSubstAbs-invert*:
assumes *eSubstAbs MOD ys eY y eA* = *OK Z*
shows
∃ *us s A Y*. *eY* = *OK Y* ∧ *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A* ∧ *igWls MOD* (*asSort ys*) *Y* ∧
Z = *igSubstAbs MOD ys Y y A*
proof−
have *1*: *eSubstAbs MOD ys eY y eA* ≠ *ERR* **using** *assms* **by** *auto*
then obtain *us s A Y* **where** ∗: *eA* = *OK A* *eY* = *OK Y*
igWlsAbs MOD (*us*,*s*) *A* *igWls MOD* (*asSort ys*) *Y*
using *eSubstAbs-nchotomy*[*of eY - - eA*] **by** *fastforce*
hence *eSubstAbs MOD ys eY y eA* = *OK* (*igSubstAbs MOD ys Y y A*) **by** *simp*
thus *?thesis* **using** *assms* ∗ **by** *auto*
qed

8.4.6 The error model is strongly well-sorted as a fresh-swap-subst and as a fresh-subst-swap model

That is, provided the original model is a well-sorted fresh-swap model.

The domains are disjoint:

lemma *errMOD-igWlsDisj*:
assumes *igWlsDisj MOD*
shows *igWlsDisj* (*errMOD MOD*)
using *assms* **unfolding** *errMOD-def igWlsDisj-def*
apply *clarify* **subgoal for** - - *X* **by**(*cases X*) *auto* **.**

lemma *errMOD-igWlsAbsDisj*:
assumes *igWlsAbsDisj MOD*
shows *igWlsAbsDisj* (*errMOD MOD*)
using *assms* **unfolding** *errMOD-def igWlsAbsDisj-def*
apply *clarify* **subgoal for** - - - - *A* **by**(*cases A*) *fastforce+* **.**

lemma *errMOD-igWlsAllDisj*:
assumes *igWlsAllDisj MOD*
shows *igWlsAllDisj* (*errMOD MOD*)
using *assms* **unfolding** *igWlsAllDisj-def*
using *errMOD-igWlsDisj errMOD-igWlsAbsDisj* **by** *auto*

Only "bound arity" abstraction domains are inhabited:

lemma *errMOD-igWlsAbsIsInBar*:
assumes *igWlsAbsIsInBar MOD*
shows *igWlsAbsIsInBar* (*errMOD MOD*)
using *assms eWlsAbs-invert* **unfolding** *igWlsAbsIsInBar-def* **by** *blast*

The operators preserve the domains strongly:

lemma *errMOD-igVarIPresIGWlsSTR*:
assumes *igVarIPresIGWls MOD*
shows *igVarIPresIGWls* (*errMOD MOD*)
using *assms* **unfolding** *errMOD-def igVarIPresIGWls-def* **by** *simp*

lemma *errMOD-igAbsIPresIGWlsSTR*:
assumes ∗: *igAbsIPresIGWls MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igAbsIPresIGWlsSTR* (*errMOD MOD*)
using *assms* **by** (*fastforce simp*: *errMOD-def igAbsIPresIGWls-def igAbsIPresIG-WlsSTR-def*
igWlsAbsIsInBar-def igWlsAbsDisj-def split: *withERR.splits*)

lemma *errMOD-igOpIPresIGWlsSTR*:
fixes *MOD* :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model*
assumes *igOpIPresIGWls MOD*
shows *igOpIPresIGWlsSTR* (*errMOD MOD*)
by (*simp add*: *igOpIPresIGWlsSTR-def igOpIPresIGWls-def*)
(*smt* (*verit*) *assms eOp-nchotomy eOp-simp1 eWlsBinp-invert*
eWlsBinp-simp1 eWlsInp-invert eWlsInp-simp1 eWls-simp1 eWls-simp2 igOpIPresIG-Wls-def)

lemma *errMOD-igConsIPresIGWlsSTR*:
assumes *igConsIPresIGWls MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igConsIPresIGWlsSTR* (*errMOD MOD*)
using *assms* **unfolding** *igConsIPresIGWls-def igConsIPresIGWlsSTR-def igWl-sAllDisj-def*
using
errMOD-igVarIPresIGWlsSTR[*of MOD*]
errMOD-igAbsIPresIGWlsSTR[*of MOD*]
errMOD-igOpIPresIGWlsSTR[*of MOD*]
by *auto*

lemma *errMOD-igSwapIPresIGWlsSTR*:
assumes *igSwapIPresIGWls MOD* **and** *igWlsDisj MOD*
shows *igSwapIPresIGWlsSTR* (*errMOD MOD*)
using ‹*igSwapIPresIGWls MOD*›
using *assms* **by** (*fastforce simp*: *errMOD-def igSwapIPresIGWls-def igSwapIPresIG-WlsSTR-def*
igWlsDisj-def split: *withERR.splits*)

lemma *errMOD-igSwapAbsIPresIGWlsAbsSTR*:
assumes ∗: *igSwapAbsIPresIGWlsAbs MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igSwapAbsIPresIGWlsAbsSTR* (*errMOD MOD*)
using *assms* **by** (*simp add*: *errMOD-def igSwapAbsIPresIGWlsAbs-def igSwapAbsIPresIGWlsAbsSTR-def*
igWlsAbsIsInBar-def igWlsAbsDisj-def split: *withERR.splits*) *blast*

lemma *errMOD-igSwapAllIPresIGWlsAllSTR*:
assumes *igSwapAllIPresIGWlsAll MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igSwapAllIPresIGWlsAllSTR* (*errMOD MOD*)
using *assms*
unfolding *igSwapAllIPresIGWlsAll-def igSwapAllIPresIGWlsAllSTR-def igWlsAllDisj-def*
using *errMOD-igSwapIPresIGWlsSTR*[*of MOD*] *errMOD-igSwapIPresIGWlsSTR*[*of MOD*]
errMOD-igSwapAbsIPresIGWlsAbsSTR[*of MOD*]
by *auto*

lemma *errMOD-igSubstIPresIGWlsSTR*:
assumes *igSubstIPresIGWls MOD* **and** *igWlsDisj MOD*
shows *igSubstIPresIGWlsSTR* (*errMOD MOD*)
using ‹*igSubstIPresIGWls MOD*›
using *assms* **by** (*fastforce simp*: *errMOD-def igSubstIPresIGWls-def igSubstIPresIGWlsSTR-def*
igWlsDisj-def split: *withERR.splits*)

lemma *errMOD-igSubstAbsIPresIGWlsAbsSTR*:
assumes ∗: *igSubstAbsIPresIGWlsAbs MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igSubstAbsIPresIGWlsAbsSTR* (*errMOD MOD*)
using *assms* **by** (*simp add*: *errMOD-def igSubstAbsIPresIGWlsAbs-def igSubstAbsIPresIGWlsAbsSTR-def*
igWlsAbsIsInBar-def igWlsAbsDisj-def split: *withERR.splits*) *blast*

lemma *errMOD-igSubstAllIPresIGWlsAllSTR*:
assumes *igSubstAllIPresIGWlsAll MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igSubstAllIPresIGWlsAllSTR* (*errMOD MOD*)
using *assms*
unfolding *igSubstAllIPresIGWlsAll-def igSubstAllIPresIGWlsAllSTR-def igWlsAllDisj-def*
using *errMOD-igSubstIPresIGWlsSTR*[*of MOD*] *errMOD-igSubstIPresIGWlsSTR*[*of MOD*]
errMOD-igSubstAbsIPresIGWlsAbsSTR[*of MOD*]
by *auto*

The strong “fresh" clauses are satisfied:

lemma *errMOD-igFreshIGVarSTR*:
assumes *igVarIPresIGWls MOD* **and** *igFreshIGVar MOD*
shows *igFreshIGVar* (*errMOD MOD*)
using *assms eFresh-simp1*
by(*fastforce simp*: *igVarIPresIGWls-def igFreshIGVar-def*)

lemma *errMOD-igFreshIGAbs1STR*:
assumes ∗: *igAbsIPresIGWls MOD* **and** ∗∗: *igFreshIGAbs1 MOD*
shows *igFreshIGAbs1STR* (*errMOD MOD*)
unfolding *igFreshIGAbs1STR-def* **proof**(*clarify*)
  fix *ys y eX*
  show *eFreshAbs MOD ys y* (*eAbs MOD ys y eX*)
  proof(*cases eX* ≠ *ERR*)
    define *X* **where** *X* ≡ *check eX*
    case *True*
    hence *eX*: *eX* = *OK X* **unfolding** *X-def* **using** *OK-check* **by** *auto*
    show *?thesis* **using** *assms eFreshAbs-simp1* **unfolding** *eX*
    by (*cases* ∃ *s*. *isInBar* (*ys*,*s*) ∧ *igWls MOD s X*)
    (*fastforce simp*: *igAbsIPresIGWls-def igFreshIGAbs1-def*)+
  qed *auto*
qed

lemma *errMOD-igFreshIGAbs2STR*:
assumes *igAbsIPresIGWls MOD* **and** *igFreshIGAbs2 MOD*
shows *igFreshIGAbs2STR* (*errMOD MOD*)
unfolding *igFreshIGAbs2STR-def* **proof**(*clarify*)
  fix *ys y xs x eX*
  assume ∗: *eFresh MOD ys y eX*
  define *X* **where** *X* ≡ *check eX*
  show *eFreshAbs MOD ys y* (*eAbs MOD xs x eX*)
  proof(*cases eX* ≠ *ERR*)
    case *True*
    hence *eX*: *eX* = *OK X* **unfolding** *X-def* **using** *OK-check* **by** *auto*
    show *?thesis* **unfolding** *eX*
    using *assms* ∗ *eFreshAbs-invert eX*
    by (*cases* ∃ *s*. *isInBar* (*xs*,*s*) ∧ *igWls MOD s X*)
    (*fastforce simp*: *igAbsIPresIGWls-def igFreshIGAbs2-def*)+
  qed *auto*
qed

lemma *errMOD-igFreshIGOpSTR*:
fixes *MOD* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*)*model*
assumes *igOpIPresIGWls MOD* **and** *igFreshIGOp MOD*
shows *igFreshIGOpSTR* (*errMOD MOD*)
unfolding *igFreshIGOpSTR-def* **apply** *clarify*
subgoal for *ys y delta einp ebinp*
apply(*cases liftAll* (λ*eX*. *eX* ≠ *ERR*) *einp* ∧

liftAll ($\lambda eA.\ eA \neq ERR$) *ebinp*)
using *assms* **by** (*simp-all add*: *igOpIPresIGWls-def igFreshIGOp-def*)
(*metis eFreshBinp-simp eFreshInp-simp eFresh-invert eOp-invert*)+ **.**

lemma *errMOD-igFreshClsSTR*:
assumes *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *igFreshClsSTR* (*errMOD MOD*)
using *assms* **unfolding** *igConsIPresIGWls-def igFreshCls-def igFreshClsSTR-def*
using
errMOD-igFreshIGVarSTR
errMOD-igFreshIGAbs1STR errMOD-igFreshIGAbs2STR
errMOD-igFreshIGOpSTR
by *auto*

The strong "swap" clauses are satisfied:

lemma *errMOD-igSwapIGVarSTR*:
fixes *MOD* :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model*
assumes *igVarIPresIGWls MOD* **and** *igSwapIGVar MOD*
shows *igSwapIGVar* (*errMOD MOD*)
using *assms* **by** (*simp add*: *igVarIPresIGWls-def igSwapIGVar-def*) (*metis eSwap-simp1*)

lemma *errMOD-igSwapIGAbsSTR*:
assumes $*$: *igAbsIPresIGWls MOD* **and** $**$: *igWlsDisj MOD*
and $***$: *igSwapIPresIGWls MOD* **and** $****$: *igSwapIGAbs MOD*
shows *igSwapIGAbsSTR* (*errMOD MOD*)
unfolding *igSwapIGAbsSTR-def* **apply**(*clarify*)
subgoal for *zs z1 z2 xs x eX*
apply (*cases eX*)
 subgoal by *auto*
 subgoal for *X*
 apply(*cases* $\exists$ *s. isInBar* (*xs,s*) $\wedge$ *igWls MOD s X*)
  subgoal using *assms*
  using *assms OK-check*
  by (*simp-all add*: *igAbsIPresIGWls-def igSwapIPresIGWls-def igSwapIGAbs-def igWlsDisj-def*)
    (*smt* (*verit*) *eAbs-simp1 eSwapAbs-simp1 eSwap-simp1 withERR.inject*)
  subgoal using *assms*
  by(*simp-all add*: *igAbsIPresIGWls-def igSwapIPresIGWls-def igSwapIGAbs-def igWlsDisj-def*)
   (*metis check-OK eAbs-nchotomy eSwap-invert*) **. . .**

lemma *errMOD-igSwapIGOpSTR*:
assumes *igWlsAbsIsInBar MOD* **and** *igOpIPresIGWls MOD*
and *igSwapIPresIGWls MOD* **and** *igSwapAbsIPresIGWlsAbs MOD*
and *igWlsDisj MOD* **and** *igWlsAbsDisj MOD*
and *igSwapIGOp MOD*
shows *igSwapIGOpSTR* (*errMOD MOD*)
unfolding *igSwapIGOpSTR-def* **proof**(*clarify*)
  have *0*: *igSwapInpIPresIGWlsInp MOD* $\wedge$ *igSwapBinpIPresIGWlsBinp MOD*

using ‹*igSwapIPresIGWls MOD*› ‹*igSwapAbsIPresIGWlsAbs MOD*›
imp-igSwapInpIPresIGWlsInp imp-igSwapBinpIPresIGWlsBinp **by** *auto*
have *igSwapIPresIGWlsSTR* (*errMOD MOD*) ∧
igSwapAbsIPresIGWlsAbsSTR (*errMOD MOD*)
using *assms errMOD-igSwapIPresIGWlsSTR*
errMOD-igSwapAbsIPresIGWlsAbsSTR **by** *auto*
hence *1*: *igSwapInpIPresIGWlsInpSTR* (*errMOD MOD*) ∧
igSwapBinpIPresIGWlsBinpSTR (*errMOD MOD*)
using *imp-igSwapInpIPresIGWlsInpSTR*
imp-igSwapBinpIPresIGWlsBinpSTR **by** *auto*
fix *zs*::*'varSort* **and** *z1 z2* ::*'var* **and** *delta einp ebinp*
let *?Left* = *eSwap MOD zs z1 z2* (*eOp MOD delta einp ebinp*)
let *?einpsw* = *eSwapInp MOD zs z1 z2 einp*
let *?ebinpsw* = *eSwapBinp MOD zs z1 z2 ebinp*
let *?Right* = *eOp MOD delta ?einpsw ?ebinpsw*
show *?Left* = *?Right*
proof(*cases liftAll* (λ*eX*. *eX* ≠ *ERR*) *einp* ∧
liftAll (λ*eA*. *eA* ≠ *ERR*) *ebinp*)
case *True* **note** *t* = *True*
moreover obtain *inp* **and** *binp* **where**
inp = *checkI einp* **and** *binp* = *checkI ebinp* **by** *blast*
ultimately have *einp*: *einp* = *OKI inp* *ebinp* = *OKI binp* **by** *auto*
show *?thesis*
proof(*cases igWlsInp MOD delta inp* ∧ *igWlsBinp MOD delta binp*)
case *False*
hence *?Left* = *ERR* **unfolding** *einp* **by** *auto*
have ¬ (*eWlsInp MOD delta einp* ∧ *eWlsBinp MOD delta ebinp*)
unfolding *einp* **using** *False* **by** *auto*
hence *2*: ¬ (*eWlsInp MOD delta ?einpsw* ∧ *eWlsBinp MOD delta ?ebinpsw*)
using *1* **unfolding** *igSwapInpIPresIGWlsInpSTR-def*
igSwapBinpIPresIGWlsBinpSTR-def **by** *auto*
{**fix** *X* **assume** *?Right* = *OK X*
then obtain *inpsw* **and** *binpsw*
where *?einpsw* = *OKI inpsw* **and** *?ebinpsw* = *OKI binpsw*
and *igWlsInp MOD delta inpsw* **and** *igWlsBinp MOD delta binpsw*
and *X* = *igOp MOD delta inpsw binpsw*
using *eOp-invert*[*of MOD delta ?einpsw ?ebinpsw X*] **by** *auto*
hence *False* **using** *2* **by** *auto*
}
with ‹*?Left* = *ERR*› **show** *?thesis* **by** (*cases ?Right*) *auto*
next
case *True*
moreover have *igWls MOD* (*stOf delta*) (*igOp MOD delta inp binp*)
using *True* ‹*igOpIPresIGWls MOD*› **unfolding** *igOpIPresIGWls-def* **by** *simp*
moreover have *igWlsInp MOD delta* (*igSwapInp MOD zs z1 z2 inp*) ∧
igWlsBinp MOD delta (*igSwapBinp MOD zs z1 z2 binp*)
using *0* **unfolding** *igSwapInpIPresIGWlsInp-def igSwapBinpIPresIGWls-Binp-def*
using *True* **by** *simp*

ultimately show *?thesis* **using** ‹*igSwapIGOp MOD*› **unfolding** *einp igSwapIGOp-def*
by *auto*
qed
qed *auto*
qed

lemma *errMOD-igSwapClsSTR*:
assumes *igWlsAllDisj MOD* **and** *igWlsDisj MOD*
and *igWlsAbsIsInBar MOD* **and** *igConsIPresIGWls MOD*
and *igSwapAllIPresIGWlsAll MOD* **and** *igSwapCls MOD*
shows *igSwapClsSTR* (*errMOD MOD*)
using *assms*
unfolding *igWlsAllDisj-def igConsIPresIGWls-def igSwapCls-def*
igSwapAllIPresIGWlsAll-def igSwapClsSTR-def
using
errMOD-igSwapIGVarSTR[*of MOD*]
errMOD-igSwapIGAbsSTR[*of MOD*]
errMOD-igSwapIGOpSTR[*of MOD*]
by *simp*

The strong "subst" clauses are satisfied:

lemma *errMOD-igSubstIGVar1STR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIGVar1 MOD*
shows *igSubstIGVar1STR* (*errMOD MOD*)
using *assms*
by (*simp add*: *igSubstIGVar1STR-def igVarIPresIGWls-def igSubstIGVar1-def*)
(*metis eSubst-simp1 eWls-invert*)

lemma *errMOD-igSubstIGVar2STR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIGVar2 MOD*
shows *igSubstIGVar2STR* (*errMOD MOD*)
using *assms*
by (*simp add*: *igSubstIGVar2STR-def igVarIPresIGWls-def igSubstIGVar2-def*)
(*metis eSubst-simp1 eWls-invert*)

lemma *errMOD-igSubstIGAbsSTR*:
fixes *MOD* :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model*
assumes $*$: *igAbsIPresIGWls MOD* **and** $**$: *igWlsDisj MOD*
and $***$: *igSubstIPresIGWls MOD* **and** $****$: *igSubstIGAbs MOD*
shows *igSubstIGAbsSTR* (*errMOD MOD*)
unfolding *igSubstIGAbsSTR-def* **proof**(*clarify*)
fix *ys xs* ::*'varSort* **and** *y x* ::*'var* **and** *eX eY*
assume *diff*: $xs \neq ys \vee x \neq y$
and *x-fresh-Y*: *eFresh MOD xs x eY*
show *eSubstAbs MOD ys eY y* (*eAbs MOD xs x eX*) =
eAbs MOD xs x (*eSubst MOD ys eY y eX*)
proof(*cases* $eX \neq ERR \wedge eY \neq ERR$)
case *True*
define X **and** Y **where** $X \equiv$ *check eX* **and** $Y \equiv$ *check eY*

hence *eX*: *eX* = *OK X* **and** *eY*: *eY* = *OK Y* **unfolding** *X-def Y-def*
using *True OK-check* **by** *auto*
show *?thesis*
proof(*cases* (∃ *s*. *isInBar* (*xs*,*s*) ∧ *igWls MOD s X*) ∧ *igWls MOD* (*asSort ys*) *Y*)
case *True*
then obtain *s* **where** *xs-s*: *isInBar* (*xs*, *s*) **and** *X*: *igWls MOD s X*
and *Y*: *igWls MOD* (*asSort ys*) *Y* **by** *auto*
hence *igWlsAbs MOD* (*xs*,*s*) (*igAbs MOD xs x X*)
using ∗ **unfolding** *igAbsIPresIGWls-def* **by** *simp*
moreover have *igWls MOD s* (*igSubst MOD ys Y y X*)
using *X Y* ∗∗∗ **unfolding** *igSubstIPresIGWls-def* **by** *simp*
moreover have *igFresh MOD xs x Y*
using *x-fresh-Y Y* **unfolding** *eY* **by** *simp*
ultimately show *?thesis* **unfolding** *eX eY*
using *xs-s X Y* **apply** *simp*
using *x-fresh-Y diff* ∗∗∗∗ **unfolding** *igSubstIGAbs-def* **by** *fastforce*
next
case *False*
show *?thesis*
proof(*cases* (*EX s*. *igWls MOD s X*) ∧ *igWls MOD* (*asSort ys*) *Y*)
case *True*
then obtain *s* **where** *X*: *igWls MOD s X* **and** *Y*: *igWls MOD* (*asSort ys*) *Y* **by** *auto*
hence *2*: ~ *isInBar* (*xs*,*s*) **using** *False* **by** (*auto simp*: *eX eY*)
let *?Xsb* = *igSubst MOD ys Y y X*
have *Xsb*: *igWls MOD s ?Xsb*
using *Y X* ∗∗∗ **unfolding** *igSubstIPresIGWls-def* **by** *auto*
{fix *s'* **assume** *3*: *isInBar* (*xs*,*s'*) **and** *igWls MOD s' ?Xsb*
hence *s* = *s'* **using** *Xsb* ∗∗ **unfolding** *igWlsDisj-def* **by** *auto*
hence *False* **using** *2 3* **by** (*simp add*: *eX eY*)
}
thus *?thesis* **using** *False Y eAbs-simp2 X eX eY* **by** *fastforce*
qed(*auto simp add*: *eX eY*)
qed
qed *auto*
qed

lemma *errMOD-igSubstIGOpSTR*:
assumes *igWlsAbsIsInBar MOD*
and *igVarIPresIGWls MOD* **and** *igOpIPresIGWls MOD*
and *igSubstIPresIGWls MOD* **and** *igSubstAbsIPresIGWlsAbs MOD*
and *igWlsDisj MOD* **and** *igWlsAbsDisj MOD*
and *igSubstIGOp MOD*
shows *igSubstIGOpSTR* (*errMOD MOD*)
proof−
have *0*: *igSubstInpIPresIGWlsInp MOD* ∧ *igSubstBinpIPresIGWlsBinp MOD*
using ‹*igSubstIPresIGWls MOD*› ‹*igSubstAbsIPresIGWlsAbs MOD*›
imp-igSubstInpIPresIGWlsInp imp-igSubstBinpIPresIGWlsBinp **by** *auto*

have *igSubstIPresIGWlsSTR* (*errMOD MOD*) ∧ *igSubstAbsIPresIGWlsAbsSTR* (*errMOD MOD*)
using *assms errMOD-igSubstIPresIGWlsSTR errMOD-igSubstAbsIPresIGWlsAbsSTR* **by** *auto*
hence *1*: *igSubstInpIPresIGWlsInpSTR* (*errMOD MOD*) ∧
igSubstBinpIPresIGWlsBinpSTR (*errMOD MOD*)
using *imp-igSubstInpIPresIGWlsInpSTR imp-igSubstBinpIPresIGWlsBinpSTR* **by** *auto*
show *?thesis*
unfolding *igSubstIGOpSTR-def* **proof** *safe*
fix *ys*::′*varSort* **and** *y y1* ::′*var* **and** *delta einp ebinp*
let *?Left* = *eSubst MOD ys* (*eVar MOD ys y1*) *y* (*eOp MOD delta einp ebinp*)
let *?einpsb* = *eSubstInp MOD ys* (*eVar MOD ys y1*) *y einp*
let *?ebinpsb* = *eSubstBinp MOD ys* (*eVar MOD ys y1*) *y ebinp*
let *?Right* = *eOp MOD delta ?einpsb ?ebinpsb*
show *?Left* = *?Right*
proof(*cases liftAll* (λ*eX*. *eX* ≠ *ERR*) *einp* ∧ *liftAll* (λ*eA*. *eA* ≠ *ERR*) *ebinp*)
case *True*
moreover obtain *inp binp* **where**
inp = *checkI einp* **and** *binp* = *checkI ebinp* **by** *blast*
ultimately have *einp*: *einp* = *OKI inp* *ebinp* = *OKI binp* **by** *auto*
have *igWls-y1*: *igWls MOD* (*asSort ys*) (*igVar MOD ys y1*)
using ‹*igVarIPresIGWls MOD*› **unfolding** *igVarIPresIGWls-def* **by** *simp*
show *?thesis*
proof(*cases igWlsInp MOD delta inp* ∧ *igWlsBinp MOD delta binp*)
case *False*
hence *?Left* = *ERR* **unfolding** *einp* **by** *auto*
have ¬ (*eWlsInp MOD delta einp* ∧ *eWlsBinp MOD delta ebinp*)
unfolding *einp* **using** *False* **by** *simp*
hence *2*: ¬ (*eWlsInp MOD delta ?einpsb* ∧ *eWlsBinp MOD delta ?ebinpsb*)
using *igWls-y1 1*
unfolding *igSubstInpIPresIGWlsInpSTR-def igSubstBinpIPresIGWlsBinpSTR-def* **by** *simp*
{**fix** *X* **assume** *?Right* = *OK X*
then obtain *inpsb binpsb* **where**
?einpsb = *OKI inpsb* **and** *?ebinpsb* = *OKI binpsb*
and *igWlsInp MOD delta inpsb* **and** *igWlsBinp MOD delta binpsb*
and *X* = *igOp MOD delta inpsb binpsb*
using *eOp-invert*[*of MOD delta ?einpsb ?ebinpsb X*] **by** *auto*
hence *False* **using** *2* **by** *auto*
}
hence *?Right* = *ERR* **by** (*cases ?Right*, *auto*)
with ‹*?Left* = *ERR*› **show** *?thesis* **by** *simp*
next
case *True*
moreover have *igWls MOD* (*stOf delta*) (*igOp MOD delta inp binp*)
using *True* ‹*igOpIPresIGWls MOD*› **unfolding** *igOpIPresIGWls-def* **by** *simp*
moreover

have *igWlsInp MOD delta* (*igSubstInp MOD ys* (*igVar MOD ys y1*) *y inp*) ∧
igWlsBinp MOD delta (*igSubstBinp MOD ys* (*igVar MOD ys y1*) *y binp*)
using *0* **unfolding** *igSubstInpIPresIGWlsInp-def igSubstBinpIPresIGWls-Binp-def*
using *True igWls-y1* **by** *simp*
ultimately show *?thesis*
using ‹*igSubstIGOp MOD*› *igWls-y1* **unfolding** *einp igSubstIGOp-def* **by** *auto*
qed
qed *auto*
next
fix *ys*::'*varSort* **and** *y* ::'*var* **and** *eY delta einp ebinp*
assume *eY*: *eWls MOD* (*asSort ys*) *eY*
let *?Left* = *eSubst MOD ys eY y* (*eOp MOD delta einp ebinp*)
let *?einpsb* = *eSubstInp MOD ys eY y einp*
let *?ebinpsb* = *eSubstBinp MOD ys eY y ebinp*
let *?Right* = *eOp MOD delta ?einpsb ?ebinpsb*
from *eY* **obtain** *Y* **where** *eY-def*: *eY* = *OK Y*
and *Y*: *igWls MOD* (*asSort ys*) *Y* **using** *eWls-invert*[*of MOD asSort ys eY*] **by** *auto*
show *?Left* = *?Right*
proof(*cases liftAll* (λ*eX*. *eX* ≠ *ERR*) *einp* ∧ *liftAll* (λ*eA*. *eA* ≠ *ERR*) *ebinp*)
case *True*
moreover obtain *inp binp* **where**
inp = *checkI einp* **and** *binp* = *checkI ebinp* **by** *blast*
ultimately have *einp*: *einp* = *OKI inp* *ebinp* = *OKI binp* **by** *auto*
show *?thesis*
proof(*cases igWlsInp MOD delta inp* ∧ *igWlsBinp MOD delta binp*)
case *False*
hence *?Left* = *ERR* **unfolding** *einp* **by** *auto*
have ¬ (*eWlsInp MOD delta einp* ∧ *eWlsBinp MOD delta ebinp*)
unfolding *einp* **using** *False* **by** *simp*
hence *2*: ¬ (*eWlsInp MOD delta ?einpsb* ∧ *eWlsBinp MOD delta ?ebinpsb*)
unfolding *eY-def* **using** *Y 1*
unfolding *igSubstInpIPresIGWlsInpSTR-def igSubstBinpIPresIGWlsBinpSTR-def* **by** *simp*
{**fix** *X* **assume** *?Right* = *OK X*
then obtain *inpsb binpsb*
where *?einpsb* = *OKI inpsb* **and** *?ebinpsb* = *OKI binpsb*
and *igWlsInp MOD delta inpsb* **and** *igWlsBinp MOD delta binpsb*
and *X* = *igOp MOD delta inpsb binpsb*
using *eOp-invert*[*of MOD delta ?einpsb ?ebinpsb X*] **by** *auto*
hence *False* **using** *2* **by** *auto*
}
hence *?Right* = *ERR* **by** (*cases ?Right*, *auto*)
with ‹*?Left* = *ERR*› **show** *?thesis* **by** *simp*
next

case *True*
moreover have *igWls MOD* (*stOf delta*) (*igOp MOD delta inp binp*)
using *True* ‹*igOpIPresIGWls MOD*› **unfolding** *igOpIPresIGWls-def* **by** *simp*
moreover
have *igWlsInp MOD delta* (*igSubstInp MOD ys Y y inp*) $\wedge$
igWlsBinp MOD delta (*igSubstBinp MOD ys Y y binp*)
using *0* **unfolding** *igSubstInpIPresIGWlsInp-def igSubstBinpIPresIGWls-Binp-def*
using *True Y* **by** *simp*
ultimately show *?thesis* **unfolding** *einp eY-def*
using ‹*igSubstIGOp MOD*› *Y* **unfolding** *igSubstIGOp-def* **by** *auto*
qed
qed *auto*
qed
qed

lemma *errMOD-igSubstClsSTR*:
assumes *igWlsAllDisj MOD* **and** *igConsIPresIGWls MOD*
and *igWlsAbsIsInBar MOD*
and *igSubstAllIPresIGWlsAll MOD* **and** *igSubstCls MOD*
shows *igSubstClsSTR* (*errMOD MOD*)
using *assms*
unfolding *igWlsAllDisj-def igConsIPresIGWls-def igSubstCls-def*
igSubstAllIPresIGWlsAll-def igSubstClsSTR-def
using
errMOD-igSubstIGVar1STR[*of MOD*] *errMOD-igSubstIGVar2STR*[*of MOD*]
errMOD-igSubstIGAbsSTR[*of MOD*]
errMOD-igSubstIGOpSTR[*of MOD*]
by *simp*

Strong swap-based congruence for abstractions holds:

lemma *errMOD-igAbsCongSSTR*:
assumes *igSwapIPresIGWls MOD* **and** *igWlsDisj MOD* **and** *igAbsCongS MOD*
shows *igAbsCongSSTR* (*errMOD MOD*)
unfolding *igAbsCongSSTR-def* **proof**(*clarify*)
fix *xs* ::*'varSort* **and** *x x' y* ::*'var* **and** *eX eX'*
assume ∗: *eFresh MOD xs y eX* **and** ∗∗: *eFresh MOD xs y eX'*
and ∗∗∗: *eSwap MOD xs y x eX* = *eSwap MOD xs y x' eX'*
let *?phi* = λ*eX*. *eX* = *ERR* $\vee$ ($\exists$ *X*. *eX* = *OK X* $\wedge$ ($\forall$ *s*. $\neg$ *igWls MOD s X*))
have *1*: *?phi eX* = *?phi eX'*
proof
assume *?phi eX*
{**fix** *X' s'* **assume** *eX'* = *OK X'* $\wedge$ ($\exists$ *s*. *igWls MOD s X'*)
hence *ERR* = *OK* (*igSwap MOD xs y x' X'*) **using** ‹*?phi eX*› ∗∗∗ **by** *auto*
}
thus *?phi eX'* **by**(*cases eX'*, *auto*)
next
assume *?phi eX'*

{**fix** *X* **assume** *eX* = *OK X* ∧ (∃ *s*. *igWls MOD s X*)
hence *ERR* = *OK* (*igSwap MOD xs y x X*) **using** ‹*?phi eX′*› ∗∗∗ **by** *auto*
}
thus *?phi eX* **by**(*cases eX*, *auto*)
qed
show *eAbs MOD xs x eX* = *eAbs MOD xs x′ eX′*
proof(*cases ?phi eX*)
case *True*
thus *?thesis* **using** *1* **by** *auto*
next
case *False*
then obtain *s X* **where** *eX*: *eX* = *OK X* **and** *X-wls*: *igWls MOD s X* **by**(*cases eX*, *auto*)
obtain *s′ X′* **where** *eX′*: *eX′* = *OK X′* **and** *X′-wls*: *igWls MOD s′ X′*
using ‹¬ *?phi eX*› *1* **by**(*cases eX′*) *auto*
hence *igSwap MOD xs y x X* = *igSwap MOD xs y x′ X′*
using *eX X-wls* ∗∗∗ **by** *auto*
moreover have *igWls MOD s* (*igSwap MOD xs y x X*)
using *X-wls* ‹*igSwapIPresIGWls MOD*› **unfolding** *igSwapIPresIGWls-def* **by** *simp*
moreover have *igWls MOD s′* (*igSwap MOD xs y x′ X′*)
using *X′-wls* ‹*igSwapIPresIGWls MOD*› **unfolding** *igSwapIPresIGWls-def* **by** *simp*
ultimately have *s′* = *s* **using** ‹*igWlsDisj MOD*› **unfolding** *igWlsDisj-def* **by** *auto*
show *?thesis*
proof (*cases isInBar* (*xs*,*s*))
case *True*
have *igFresh MOD xs y X* **using** ∗ *X-wls* **unfolding** *eX* **by** *simp*
moreover have *igFresh MOD xs y X′* **using** ∗∗ *X′-wls* **unfolding** *eX′* **by** *simp*
moreover have *igSwap MOD xs y x X* = *igSwap MOD xs y x′ X′*
using ∗∗∗ *X-wls X′-wls* **unfolding** *eX eX′* **by** *simp*
ultimately show *?thesis*
unfolding *eX eX′*
using *X-wls X′-wls* **unfolding** ‹*s′* = *s*›
using ‹*igAbsCongS MOD*› *True* **unfolding** *igAbsCongS-def*
by (*metis FixSyn.eCons-simps*(*2*) *FixSyn-axioms*)
next
case *False*
{**fix** *s″* **assume** *xs-s″*: *isInBar* (*xs*,*s″*) **and** *igWls MOD s″ X*
hence *s* = *s″* **using** *X-wls* ‹*igWlsDisj MOD*› **unfolding** *igWlsDisj-def* **by** *auto*
hence *False* **using** *False xs-s″* **by** *simp*
}
moreover
{**fix** *s″* **assume** *xs-s″*: *isInBar* (*xs*,*s″*) **and** *igWls MOD s″ X′*
hence *s* = *s″* **using** *X′-wls* ‹*igWlsDisj MOD*› **unfolding** *igWlsDisj-def* ‹*s′* = *s*› **by** *auto*

hence *False* **using** *False xs-s''* **by** *simp*
}
ultimately show *?thesis*
using *eX eX' X-wls X'-wls* **unfolding** ‹*s' = s*› **by** *fastforce*
qed
qed
qed

The renaming clause for abstractions holds:

lemma *errMOD-igAbsRenSTR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIPresIGWls MOD*
and *igWlsDisj MOD* **and** *igAbsRen MOD*
shows *igAbsRenSTR* (*errMOD MOD*)
using *assms* **unfolding** *igAbsRenSTR-def* **apply** *clarify*
subgoal for *xs y x eX*
apply(*cases eX*)
subgoal by *auto*
subgoal for *X*
apply(*cases EX s. isInBar* (*xs*,*s*) ∧ *igWls MOD s X*)
subgoal by (*auto simp*: *igVarIPresIGWls-def igSubstIPresIGWls-def igAbsRen-def*)

subgoal using *assms* **by** (*simp add*: *igVarIPresIGWls-def igSubstIPresIG-Wls-def igAbsRen-def igWlsDisj-def*)
(*metis eAbs-simp2 eAbs-simp3 eSubst-simp1 eSubst-simp3*) **. . .**

Strong subst-based congruence for abstractions holds:

corollary *errMOD-igAbsCongUSTR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIPresIGWls MOD*
and *igWlsDisj MOD* **and** *igAbsRen MOD*
shows *igAbsCongUSTR* (*errMOD MOD*)
using *assms errMOD-igAbsRenSTR igAbsRenSTR-imp-igAbsCongUSTR* **by** *auto*

The error model is a strongly well-sorted fresh-swap model:

lemma *errMOD-iwlsFSwSTR*:
fixes *MOD* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*) *model*
assumes *iwlsFSw MOD*
shows *iwlsFSwSTR* (*errMOD MOD*)
using *assms* **unfolding** *iwlsFSw-def iwlsFSwSTR-def*
using *errMOD-igWlsAllDisj*[*of MOD*]
errMOD-igWlsAbsIsInBar[*of MOD*]
errMOD-igConsIPresIGWlsSTR[*of MOD*]
errMOD-igSwapAllIPresIGWlsAllSTR[*of MOD*]
errMOD-igFreshClsSTR[*of MOD*] *errMOD-igSwapClsSTR*[*of MOD*]
errMOD-igAbsCongSSTR[*of MOD*]
apply *simp*
unfolding *igSwapAllIPresIGWlsAll-def igWlsAllDisj-defs* **by** *simp*

The error model is a strongly well-sorted fresh-subst model:

lemma *errMOD-iwlsFSbSwTR*:

fixes *MOD* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*) *model*
assumes *iwlsFSb MOD*
shows *iwlsFSbSwTR* (*errMOD MOD*)
using *assms* **unfolding** *iwlsFSb-def iwlsFSbSwTR-def*
using *errMOD-igWlsAllDisj*[*of MOD*]
errMOD-igWlsAbsIsInBar[*of MOD*]
errMOD-igConsIPresIGWlsSTR[*of MOD*]
errMOD-igSubstAllIPresIGWlsAllSTR[*of MOD*]
errMOD-igFreshClsSTR[*of MOD*] *errMOD-igSubstClsSTR*[*of MOD*]
errMOD-igAbsRenSTR[*of MOD*]
by (*simp add*: *igConsIPresIGWls-def igSubstAllIPresIGWlsAll-def igWlsAllDisj-defs*)

8.4.7 The natural morhpism from an error model to its original model

This morphism is igiven by the "check" functions.

Preservation of the domains:

lemma *check-ipresIGWls*:
ipresIGWls check (*errMOD MOD*) *MOD*
unfolding *ipresIGWls-def* **apply** *clarify*
subgoal for - *X* **by**(*cases X*) *auto* **.**

lemma *check-ipresIGWlsAbs*:
ipresIGWlsAbs check (*errMOD MOD*) *MOD*
unfolding *ipresIGWlsAbs-def* **apply** *clarify*
subgoal for - - *A* **by**(*cases A*) *auto* **.**

lemma *check-ipresIGWlsAll*:
ipresIGWlsAll check check (*errMOD MOD*) *MOD*
unfolding *ipresIGWlsAll-def*
using *check-ipresIGWls check-ipresIGWlsAbs* **by** *auto*

Preservation of the operations:

lemma *check-ipresIGVar*:
ipresIGVar check (*errMOD MOD*) *MOD*
unfolding *ipresIGVar-def* **by** *simp*

lemma *check-ipresIGAbs*:
ipresIGAbs check check (*errMOD MOD*) *MOD*
unfolding *ipresIGAbs-def* **apply** *clarify*
subgoal for - - - *X* **by**(*cases X*) *auto* **.**

lemma *check-ipresIGOp*:
ipresIGOp check check (*errMOD MOD*) *MOD*
unfolding *ipresIGOp-def* **proof** *clarify*
  fix *delta einp ebinp*
  assume *eWlsInp MOD delta einp* **and** *eWlsBinp MOD delta ebinp*
  then obtain *inp binp* **where**

igWlsInp MOD delta inp **and** *igWlsBinp MOD delta binp*
and *einp* = *OKI inp* **and** *ebinp* = *OKI binp*
using *eWlsInp-invert eWlsBinp-invert* **by** *blast*
hence *check* (*eOp MOD delta einp ebinp*) =
igOp MOD delta (*checkI einp*) (*checkI ebinp*) **by** *simp*
thus *check* (*eOp MOD delta einp ebinp*) =
igOp MOD delta (*lift check einp*) (*lift check ebinp*)
unfolding *checkI-def* **.**
qed

lemma *check-ipresIGCons*:
ipresIGCons check check (*errMOD MOD*) *MOD*
unfolding *ipresIGCons-def*
using
check-ipresIGVar
check-ipresIGAbs
check-ipresIGOp
by *auto*

lemma *check-ipresIGFresh*:
ipresIGFresh check (*errMOD MOD*) *MOD*
unfolding *ipresIGFresh-def* **apply** *clarify*
subgoal for - - - *X* **by**(*cases X*) *auto* **.**

lemma *check-ipresIGFreshAbs*:
ipresIGFreshAbs check (*errMOD MOD*) *MOD*
unfolding *ipresIGFreshAbs-def* **apply** *clarify*
subgoal for - - - - *A* **by**(*cases A*) *auto* **.**

lemma *check-ipresIGFreshAll*:
ipresIGFreshAll check check (*errMOD MOD*) *MOD*
unfolding *ipresIGFreshAll-def*
using *check-ipresIGFresh check-ipresIGFreshAbs* **by** *auto*

lemma *check-ipresIGSwap*:
ipresIGSwap check (*errMOD MOD*) *MOD*
unfolding *ipresIGSwap-def* **apply** *clarify*
subgoal for - - - - *X* **by**(*cases X*) *auto* **.**

lemma *check-ipresIGSwapAbs*:
ipresIGSwapAbs check (*errMOD MOD*) *MOD*
unfolding *ipresIGSwapAbs-def* **apply** *clarify*
subgoal for - - - - - *A* **by**(*cases A*) *auto* **.**

lemma *check-ipresIGSwapAll*:
ipresIGSwapAll check check (*errMOD MOD*) *MOD*
unfolding *ipresIGSwapAll-def*
using *check-ipresIGSwap check-ipresIGSwapAbs* **by** *auto*

lemma *check-ipresIGSubst*:
ipresIGSubst check (*errMOD MOD*) *MOD*
unfolding *ipresIGSubst-def* **apply** *clarify*
subgoal for - *Y* - - *X* **by** (*cases X*, *simp*, *cases Y*) *auto* **.**

lemma *check-ipresIGSubstAbs*:
ipresIGSubstAbs check check (*errMOD MOD*) *MOD*
unfolding *ipresIGSubstAbs-def* **apply** *clarify*
subgoal for - *Y* - - - *A* **by** (*cases A*, *simp*, *cases Y*) *auto* **.**

lemma *check-ipresIGSubstAll*:
ipresIGSubstAll check check (*errMOD MOD*) *MOD*
unfolding *ipresIGSubstAll-def*
using *check-ipresIGSubst check-ipresIGSubstAbs* **by** *auto*

“check" is a fresh-swap morphism:

lemma *check-FSwImorph*:
FSwImorph check check (*errMOD MOD*) *MOD*
unfolding *FSwImorph-def*
using *check-ipresIGWlsAll check-ipresIGCons*
check-ipresIGFreshAll check-ipresIGSwapAll **by** *auto*

“check" is a fresh-subst morphism:

lemma *check-FSbImorph*:
FSbImorph check check (*errMOD MOD*) *MOD*
unfolding *FSbImorph-def*
using *check-ipresIGWlsAll check-ipresIGCons*
check-ipresIGFreshAll check-ipresIGSubstAll **by** *auto*

8.5 Initiality of the models of terms

We show that terms form initial models in all the considered kinds. The desired initial morphism will be the composition of “check" with the factorization of the standard (absolute-initial) function from quasi-terms, “qInit", to alpha-equivalence. “qInit" preserving alpha-equivalence (in an unsorted fashion) was the main reason for introducing error models.

declare *qItem-simps*[*simp*]
declare *qItem-versus-item-simps*[*simp*]
declare *good-item-simps*[*simp*]

8.5.1 The initial map from quasi-terms to a strong model

definition
aux-qInit-ignoreFirst ::
(*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*)*model* ∗
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm* +
(*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*)*model* ∗
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs* ⇒

$('index,'bindex,'varSort,'var,'opSym)qTermItem$
where
aux-qInit-ignoreFirst K =
(*case K of Inl* (*MOD*,*qX*) ⇒ *termIn qX*
|*Inr* (*MOD*,*qA*) ⇒ *absIn qA*)

lemma *qTermLess-ingoreFirst-wf*:
wf (*inv-image qTermLess aux-qInit-ignoreFirst*)
using *qTermLess-wf wf-inv-image* **by** *auto*

function
qInit :: $('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs)model \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qTerm \Rightarrow 'gTerm$
and
qInitAbs :: $('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs)model \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)qAbs \Rightarrow 'gAbs$
where
qInit MOD (*qVar xs x*) = *igVar MOD xs x*
|
qInit MOD (*qOp delta qinp qbinp*) =
igOp MOD delta (*lift* (*qInit MOD*) *qinp*) (*lift* (*qInitAbs MOD*) *qbinp*)
|
qInitAbs MOD (*qAbs xs x qX*) = *igAbs MOD xs x* (*qInit MOD qX*)
by(*pat-completeness*) *auto*
termination
apply(*relation inv-image qTermLess aux-qInit-ignoreFirst*)
apply(*simp add*: *qTermLess-ingoreFirst-wf*)
by(*auto simp*: *qTermLess-def aux-qInit-ignoreFirst-def*)

lemma *qFreshAll-igFreshAll-qInitAll*:
assumes *igFreshClsSTR MOD*
shows
(*qFresh ys y qX* ⟶ *igFresh MOD ys y* (*qInit MOD qX*)) ∧
(*qFreshAbs ys y qA* ⟶ *igFreshAbs MOD ys y* (*qInitAbs MOD qA*))
apply(*induct rule*: *qTerm-rawInduct*)
using *assms*
by (*auto simp*: *igFreshClsSTR-def igFreshIGVar-def qFreshInp-def qFreshBinp-def liftAll-lift-comp*
liftAll-def igFreshInp-def igFreshBinp-def lift-def igFreshIGAbs1STR-def igFreshI-GAbs2STR-def igFreshIGOpSTR-def
split: *option.splits*)

corollary *iwlsFSwSTR-qFreshAll-igFreshAll-qInitAll*:
assumes *iwlsFSwSTR MOD*
shows
(*qFresh ys y qX* ⟶ *igFresh MOD ys y* (*qInit MOD qX*)) ∧
(*qFreshAbs ys y qA* ⟶ *igFreshAbs MOD ys y* (*qInitAbs MOD qA*))
using *assms* **unfolding** *iwlsFSwSTR-def* **by**(*simp add*: *qFreshAll-igFreshAll-qInitAll*)

corollary *iwlsFSbSwTR-qFreshAll-igFreshAll-qInitAll*:
assumes *iwlsFSbSwTR MOD*
shows
($qFresh\ ys\ y\ qX \longrightarrow igFresh\ MOD\ ys\ y\ (qInit\ MOD\ qX)) \wedge$
$(qFreshAbs\ ys\ y\ qA \longrightarrow igFreshAbs\ MOD\ ys\ y\ (qInitAbs\ MOD\ qA))$
using *assms* **unfolding** *iwlsFSbSwTR-def* **by**(*simp add*: *qFreshAll-igFreshAll-qInitAll*)

lemma *qSwapAll-igSwapAll-qInitAll*:
assumes *igSwapClsSTR MOD*
shows
qInit MOD (*qX* #[[*z1* ∧ *z2*]]-*zs*) = *igSwap MOD zs z1 z2* (*qInit MOD qX*) ∧
qInitAbs MOD (*qA* $[[*z1* ∧ *z2*]]-*zs*) = *igSwapAbs MOD zs z1 z2* (*qInitAbs MOD qA*)
proof(*induction rule*: *qTerm-rawInduct*)
  case (*Var xs x*)
  then show *?case* **using** *assms* **unfolding** *igSwapClsSTR-def igSwapIGVar-def* **by** *simp*
next
  case (*Op delta qinp qbinp*)
  hence *lift* (*qInit MOD*) (*qSwapInp zs z1 z2 qinp*) =
    igSwapInp MOD zs z1 z2 (*lift* (*qInit MOD*) *qinp*) ∧
    lift (*qInitAbs MOD*) (*qSwapBinp zs z1 z2 qbinp*) =
    igSwapBinp MOD zs z1 z2 (*lift* (*qInitAbs MOD*) *qbinp*)
  using *Op.IH* **by** (*auto simp*: *qSwapInp-def qSwapBinp-def igSwapInp-def lift-def liftAll-def*
  igSwapBinp-def iwlsFSwSTR-def igSwapClsSTR-def igSwapIGOpSTR-def
  split: *option.splits*)
  thus *?case*
  using *assms* **unfolding** *iwlsFSwSTR-def igSwapClsSTR-def igSwapIGOpSTR-def* **by** *simp*
next
  case (*Abs xs x X*)
  then show *?case* **using** *assms* **unfolding** *igSwapClsSTR-def igSwapIGAbsSTR-def* **by** *simp*
qed

corollary *iwlsFSwSTR-qSwapAll-igSwapAll-qInitAll*:
assumes *wls*: *iwlsFSwSTR MOD*
shows
qInit MOD (*qX* #[[*z1* ∧ *z2*]]-*zs*) = *igSwap MOD zs z1 z2* (*qInit MOD qX*) ∧
qInitAbs MOD (*qA* $[[*z1* ∧ *z2*]]-*zs*) = *igSwapAbs MOD zs z1 z2* (*qInitAbs MOD qA*)
using *assms* **unfolding** *iwlsFSwSTR-def* **by**(*simp add*: *qSwapAll-igSwapAll-qInitAll*)

lemma *qSwapAll-igSubstAll-qInitAll*:
fixes *qX*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
  qA::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
assumes ∗: *igSubstClsSTR MOD* **and** *igFreshClsSTR MOD*
and *igAbsRenSTR MOD*

shows
(*qGood qX* ⟶
(∀ *ys y1 y*.
qAFresh ys y1 qX ⟶
qInit MOD (*qX* #[[*y1* ∧ *y*]]-*ys*) = *igSubst MOD ys* (*igVar MOD ys y1*) *y* (*qInit MOD qX*)))
∧
(*qGoodAbs qA* ⟶
(∀ *ys y1 y*.
qAFreshAbs ys y1 qA ⟶
qInitAbs MOD (*qA* $[[*y1* ∧ *y*]]-*ys*) = *igSubstAbs MOD ys* (*igVar MOD ys y1*) *y* (*qInitAbs MOD qA*)))
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
show *?case* **apply** *safe*
subgoal for *ys y1 y* **using** ∗
by (*cases ys* = *xs* ∧ *y* = *x*)
(*auto simp*: *igSubstClsSTR-defs igSubstIGVar2STR-def igSubstClsSTR-defs igSubstIGVar1STR-def*).
next
let *?h* = *qInit MOD* **let** *?hA* = *qInitAbs MOD*
case (*Op delta qinp qbinp*)
then show *?case* **proof** *safe*
fix *ys y1 y*
assume ∗∗∗: *qAFresh ys y1* (*qOp delta qinp qbinp*)
have *lift ?h* (*qSwapInp ys y1 y qinp*) =
igSubstInp MOD ys (*igVar MOD ys y1*) *y* (*lift ?h qinp*) ∧
lift ?hA (*qSwapBinp ys y1 y qbinp*) =
igSubstBinp MOD ys (*igVar MOD ys y1*) *y* (*lift ?hA qbinp*)
using *Op.IH* ∗∗∗
by (*auto simp*: *qSwapInp-def igSubstInp-def qSwapBinp-def igSubstBinp-def lift-def liftAll-def split*: *option.splits*)
thus *qInit MOD* (*qOp delta qinp qbinp* #[[*y1* ∧ *y*]]-*ys*) =
igSubst MOD ys (*igVar MOD ys y1*) *y* (*qInit MOD* (*qOp delta qinp qbinp*))
using *assms* **unfolding** *iwlsFSwSTR-def igSubstClsSTR-defs igSubstIGOp-STR-def* **by** *simp*
qed
next
let *?h* = *qInit MOD* **let** *?hA* = *qInitAbs MOD*
case (*Abs xs x qX*)
show *?case* **proof** *safe*
fix *ys y1 y*
let *?xy1y* = *x* @*xs*[*y1* ∧ *y*]-*ys* **let** *?y1* = *igVar MOD ys y1*
assume *qAFreshAbs ys y1* (*qAbs xs x qX*)
hence *y1-fresh*: *ys* = *xs* ⟶ *y1* ≠ *x* *qAFresh ys y1 qX* **by** *auto*
hence *1*: *qFresh ys y1 qX* **using** *qAFresh-imp-qFresh* **by** *auto*
hence *y1-fresh-qX*: *igFresh MOD ys y1* (*?h qX*)
using *assms* **unfolding** *igSubstClsSTR-def*
by(*simp add*: *qFreshAll-igFreshAll-qInitAll*)

obtain *x1* **where** *x1-fresh*: *x1* ∉ {*y*,*y1*} *qFresh xs x1 qX* *qAFresh xs x1 qX*
using *obtain-qFresh*[*of* {*y*,*y1*} {*qX*}] **using** *Abs* **by** *blast*
hence [*simp*]: *igFresh MOD xs x1* (*?h qX*)
using *assms* **by**(*simp add*: *qFreshAll-igFreshAll-qInitAll*)
let *?x1* = *igVar MOD xs x1* **let** *?x1y1y* = *x1* @*xs*[*y1* ∧ *y*]-*ys*
let *?qX-x1x* = *qX* #[[*x1* ∧ *x*]]-*xs* **let** *?qX-x1x-y1y* = *?qX-x1x* #[[*y1* ∧ *y*]]-*ys*
let *?qX-y1y* = *qX* #[[*y1* ∧ *y*]]-*ys* **let** *?qX-y1y-x1-xy1y* = *?qX-y1y* #[[*x1* ∧ *?xy1y*]]-*xs*
let *?qX-y1y-x1y1y-xy1y* = *?qX-y1y* #[[*?x1y1y* ∧ *?xy1y*]]-*xs*
have [*simp*]: *qAFresh ys y1 ?qX-x1x*
using *y1-fresh x1-fresh* **by**(*auto simp add*: *qSwap-preserves-qAFresh-distinct*)
have [*simp*]: *qAFresh xs x1 ?qX-y1y*
using *y1-fresh x1-fresh* **by**(*auto simp add*: *qSwap-preserves-qAFresh-distinct*)
hence *qFresh xs x1 ?qX-y1y* **by** (*simp add*: *qAFresh-imp-qFresh*)
hence [*simp*]: *igFresh MOD xs x1* (*?h ?qX-y1y*)
using *assms* **by**(*simp add*: *qFreshAll-igFreshAll-qInitAll*)
have [*simp*]: *igFresh MOD xs x1 ?y1*
using *x1-fresh assms* **unfolding** *igFreshClsSTR-def igFreshIGVar-def* **by** *simp*
have *x1-def*: *x1* = *?x1y1y* **using** *x1-fresh* **by** *simp*

have *?hA* ((*qAbs xs x qX*) $[[*y1* ∧ *y*]]-*ys*) = *igAbs MOD xs ?xy1y* (*?h ?qX-y1y*) **by** *simp*
also have ... = *igAbs MOD xs x1* (*igSubst MOD xs ?x1 ?xy1y* (*?h ?qX-y1y*))
using *assms* **unfolding** *igAbsRenSTR-def* **by** *simp*
also have *igSubst MOD xs ?x1 ?xy1y* (*?h ?qX-y1y*) = *?h* (*?qX-y1y-x1-xy1y*)
using *y1-fresh Abs.IH*[*of ?qX-y1y*] **by**(*simp add*: *qSwap-qSwapped*)
also have *?qX-y1y-x1-xy1y* = *?qX-y1y-x1y1y-xy1y* **using** *x1-def* **by** *simp*
also have ... = *?qX-x1x-y1y* **apply**(*rule sym*) **by**(*rule qSwap-compose*)
also have *?h ?qX-x1x-y1y* = *igSubst MOD ys ?y1 y* (*?h ?qX-x1x*)
using *Abs.IH*[*of ?qX-x1x*] **by**(*simp add*: *qSwap-qSwapped*)
also have
igAbs MOD xs x1 (*igSubst MOD ys ?y1 y* (*?h ?qX-x1x*)) =
igSubstAbs MOD ys ?y1 y (*igAbs MOD xs x1* (*?h* (*?qX-x1x*)))
using *assms* **unfolding** *igSubstClsSTR-def igSubstIGAbsSTR-def*
using *x1-fresh y1-fresh* **by** *simp*
also have *?h* (*?qX-x1x*) = *igSubst MOD xs ?x1 x* (*?h qX*)
using *Abs.IH*[*of qX*] *x1-fresh* **by**(*simp add*: *qSwapped.Refl*)
also have
igAbs MOD xs x1 (*igSubst MOD xs ?x1 x* (*?h qX*)) =
igAbs MOD xs x (*?h qX*)
using *assms* **unfolding** *igAbsRenSTR-def* **by** *simp*
also have *igAbs MOD xs x* (*?h qX*) = *?hA* (*qAbs xs x qX*)
using *assms* **by** *simp*
finally show *?hA* ((*qAbs xs x qX*) $[[*y1* ∧ *y*]]-*ys*) =
igSubstAbs MOD ys ?y1 y (*?hA* (*qAbs xs x qX*)) **.**
qed
qed

lemma *iwlsFSbSwTR-qSwapAll-igSubstAll-qInitAll*:
assumes *wls*: *iwlsFSbSwTR MOD*
shows
(*qGood qX* ⟶
  qAFresh ys y1 qX ⟶
  qInit MOD (*qX* #[[*y1* ∧ *y*]]-*ys*) = *igSubst MOD ys* (*igVar MOD ys y1*) *y* (*qInit MOD qX*))
 ∧
 (*qGoodAbs qA* ⟶
  qAFreshAbs ys y1 qA ⟶
  qInitAbs MOD (*qA* $[[*y1* ∧ *y*]]-*ys*) = *igSubstAbs MOD ys* (*igVar MOD ys y1*) *y* (*qInitAbs MOD qA*))
using *assms* **unfolding** *iwlsFSbSwTR-def* **by**(*simp add*: *qSwapAll-igSubstAll-qInitAll*)

lemma *iwlsFSwSTR-alphaAll-qInitAll*:
assumes *iwlsFSwSTR MOD*
shows
(∀ *qX′*. *qX* #= *qX′* ⟶ *qInit MOD qX* = *qInit MOD qX′*) ∧
 (∀ *qA′*. *qA* $= *qA′* ⟶ *qInitAbs MOD qA* = *qInitAbs MOD qA′*)
proof(*induction rule*: *qTerm-induct*)
  case (*Var xs x*)
  then show *?case* **by**(*simp add*: *qVar-alpha-iff*)
next
  case (*Op delta qinp qbinp*)
  show *?case* **proof** *safe*
    fix *qX′*
    assume *qOp delta qinp qbinp* #= *qX′*
    then obtain *qinp′ qbinp′* **where** *qX′*: *qX′* = *qOp delta qinp′ qbinp′*
    and ∗: *sameDom qinp qinp′* ∧ *sameDom qbinp qbinp′*
    and ∗∗: *liftAll2* (λ*qX qX′*. *qX* #= *qX′*) *qinp qinp′* ∧
          liftAll2 (λ*qA qA′*. *qA* $= *qA′*) *qbinp qbinp′*
    using *qOp-alpha-iff*[*of delta qinp qbinp qX′*] **by** *auto*
    hence *lift* (*qInit MOD*) *qinp* = *lift* (*qInit MOD*) *qinp′*
    by (*smt* (*verit*) *Op.IH*(*1*) *liftAll2-def liftAll2-lift-ext liftAll-def*)
    moreover have *lift* (*qInitAbs MOD*) *qbinp* = *lift* (*qInitAbs MOD*) *qbinp′*
    by (*smt* (*verit*) ∗ ∗∗ *Op.IH*(*2*) *liftAll2-def liftAll2-lift-ext liftAll-def*)
    ultimately
    show *qInit MOD* (*qOp delta qinp qbinp*) = *qInit MOD qX′* **unfolding** *qX′* **by** *simp*
  qed
next
  case (*Abs xs x qX*)
  show *?case* **proof** *safe*
    fix *qA′*
    assume *qAbs xs x qX* $= *qA′*
    then obtain *x′ y qX′* **where** *qA′*: *qA′* = *qAbs xs x′ qX′*
    and *y-not*: *y* ∉ {*x*, *x′*} **and** *qAFresh xs y qX* *qAFresh xs y qX′*
    and *alpha*: (*qX* #[[*y* ∧ *x*]]-*xs*) #= (*qX′* #[[*y* ∧ *x′*]]-*xs*)
    using *qAbs-alphaAbs-iff*[*of xs x qX qA′*] **by** *auto*

hence *y-fresh*: *qFresh xs y qX* ∧ *qFresh xs y qX'* **using** *qAFresh-imp-qFresh* **by** *auto*
have (*qX*, *qX* #[[*y* ∧ *x*]]-*xs*) ∈ *qSwapped* **using** *qSwap-qSwapped* **by** *fastforce*
hence *qInit MOD* (*qX* #[[*y* ∧ *x*]]-*xs*) = *qInit MOD* (*qX'* #[[*y* ∧ *x'*]]-*xs*)
using *Abs.IH alpha* **by** *simp*
hence *igSwap MOD xs y x* (*qInit MOD qX*) = *igSwap MOD xs y x'* (*qInit MOD qX'*)
using *assms* **by**(*auto simp*: *iwlsFSwSTR-qSwapAll-igSwapAll-qInitAll*)
moreover have *igFresh MOD xs y* (*qInit MOD qX*) ∧ *igFresh MOD xs y* (*qInit MOD qX'*)
using *y-fresh assms* **by**(*auto simp add*: *iwlsFSwSTR-qFreshAll-igFreshAll-qInitAll*)
ultimately have *igAbs MOD xs x* (*qInit MOD qX*) = *igAbs MOD xs x'* (*qInit MOD qX'*)
using *y-not assms* **unfolding** *iwlsFSwSTR-def igAbsCongSSTR-def*
apply *clarify* **by** (*erule allE*[*of - xs*], *erule allE*[*of - x*]) *blast*
thus *qInitAbs MOD* (*qAbs xs x qX*) = *qInitAbs MOD qA'* **unfolding** *qA'* **by** *simp*
qed
qed

corollary *iwlsFSwSTR-qInit-respectsP-alpha*:
assumes *iwlsFSwSTR MOD* **shows** (*qInit MOD*) *respectsP alpha*
unfolding *congruentP-def* **using** *assms iwlsFSwSTR-alphaAll-qInitAll* **by** *blast*

corollary *iwlsFSwSTR-qInitAbs-respectsP-alphaAbs*:
assumes *iwlsFSwSTR MOD* **shows** (*qInitAbs MOD*) *respectsP alphaAbs*
unfolding *congruentP-def* **using** *assms iwlsFSwSTR-alphaAll-qInitAll* **by** *blast*

lemma *iwlsFSbSwTR-alphaAll-qInitAll*:
fixes *qX*::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
qA::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs*
assumes *iwlsFSbSwTR MOD*
shows
(*qGood qX* ⟶ (∀ *qX'*. *qX* #= *qX'* ⟶ *qInit MOD qX* = *qInit MOD qX'*)) ∧
(*qGoodAbs qA* ⟶ (∀ *qA'*. *qA* $= *qA'* ⟶ *qInitAbs MOD qA* = *qInitAbs MOD qA'*))
proof(*induction rule*: *qGood-qTerm-induct*)
case (*Var xs x*)
then show *?case* **by**(*simp add*: *qVar-alpha-iff*)
next
case (*Op delta qinp qbinp*)
show *?case* **proof** *safe*
fix *qX'*
assume *qOp delta qinp qbinp* #= *qX'*
then obtain *qinp' qbinp'* **where** *qX'*: *qX'* = *qOp delta qinp' qbinp'*
and ∗: *sameDom qinp qinp'* ∧ *sameDom qbinp qbinp'*
and ∗∗: *liftAll2* (λ*qX qX'*. *qX* #= *qX'*) *qinp qinp'* ∧
liftAll2 (λ*qA qA'*. *qA* $= *qA'*) *qbinp qbinp'*
using *qOp-alpha-iff*[*of delta qinp qbinp qX'*] **by** *auto*

have *lift* (*qInit MOD*) *qinp* = *lift* (*qInit MOD*) *qinp′*
using ∗ ∗∗ *Op.IH*(*1*) **by** (*simp add*: *lift-def liftAll2-def liftAll-def sameDom-def fun-eq-iff split*: *option.splits*) (*metis option.exhaust*)
moreover
have *lift* (*qInitAbs MOD*) *qbinp* = *lift* (*qInitAbs MOD*) *qbinp′*
using ∗ ∗∗ *Op.IH*(*2*) **by** (*simp add*: *lift-def liftAll2-def liftAll-def sameDom-def fun-eq-iff split*: *option.splits*) (*metis option.exhaust*)
ultimately
show *qInit MOD* (*qOp delta qinp qbinp*) = *qInit MOD qX′*
unfolding *qX′* **by** *simp*
qed
next
case (*Abs xs x qX*)
show *?case* **proof** *safe*
fix *qA′*
assume *qAbs xs x qX* $= *qA′*
then obtain *x′ y qX′* **where** *qA′*: *qA′* = *qAbs xs x′ qX′*
and *y-not*: $y \notin \{x, x'\}$ **and** *y-afresh*: *qAFresh xs y qX qAFresh xs y qX′*
and *alpha*: (*qX* #[[*y* ∧ *x*]]-*xs*) #= (*qX′* #[[*y* ∧ *x′*]]-*xs*)
using *qAbs-alphaAbs-iff*[*of xs x qX qA′*] **by** *auto*
hence *y-fresh*: *qFresh xs y qX* ∧ *qFresh xs y qX′* **using** *qAFresh-imp-qFresh* **by** *auto*
have *qX′*: *qGood qX′* **using** *alpha Abs* **by**(*simp add*: *alpha-qSwap-preserves-qGood1*)
have (*qX*, *qX* #[[*y* ∧ *x*]]-*xs*) ∈ *qSwapped* **using** *qSwap-qSwapped* **by** *fastforce*
hence *qInit MOD* (*qX* #[[*y* ∧ *x*]]-*xs*) = *qInit MOD* (*qX′* #[[*y* ∧ *x′*]]-*xs*)
using *Abs.IH alpha* **by** *simp*
moreover have *qInit MOD* (*qX* #[[*y* ∧ *x*]]-*xs*) = *igSubst MOD xs* (*igVar MOD xs y*) *x* (*qInit MOD qX*)
using *Abs y-afresh assms* **by**(*simp add*: *iwlsFSbSwTR-qSwapAll-igSubstAll-qInitAll*)
moreover have *qInit MOD* (*qX′* #[[*y* ∧ *x′*]]-*xs*) = *igSubst MOD xs* (*igVar MOD xs y*) *x′* (*qInit MOD qX′*)
using *qX′ y-afresh assms* **by**(*simp add*: *iwlsFSbSwTR-qSwapAll-igSubstAll-qInitAll*)
ultimately
have *igSubst MOD xs* (*igVar MOD xs y*) *x* (*qInit MOD qX*) =
igSubst MOD xs (*igVar MOD xs y*) *x′* (*qInit MOD qX′*)
by *simp*
moreover have *igFresh MOD xs y* (*qInit MOD qX*) ∧ *igFresh MOD xs y* (*qInit MOD qX′*)
using *y-fresh assms* **by**(*auto simp add*: *iwlsFSbSwTR-qFreshAll-igFreshAll-qInitAll*)
moreover have *igAbsCongUSTR MOD*
using *assms* **unfolding** *iwlsFSbSwTR-def* **using** *igAbsRenSTR-imp-igAbsCongUSTR* **by** *auto*
ultimately have *igAbs MOD xs x* (*qInit MOD qX*) = *igAbs MOD xs x′* (*qInit MOD qX′*)
using *y-not* **unfolding** *igAbsCongUSTR-def* **apply** *clarify*
by (*erule allE*[*of - xs*], *erule allE*[*of - x*]) *blast*
thus *qInitAbs MOD* (*qAbs xs x qX*) = *qInitAbs MOD qA′* **unfolding** *qA′* **by** *simp*
qed

qed

corollary *iwlsFSbSwTR-qInit-respectsP-alphaGood*:
assumes *iwlsFSbSwTR MOD*
shows (*qInit MOD*) *respectsP alphaGood*
unfolding *congruentP-def alphaGood-def*
using *assms iwlsFSbSwTR-alphaAll-qInitAll* **by** *fastforce*

corollary *iwlsFSbSwTR-qInitAbs-respectsP-alphaAbsGood*:
assumes *iwlsFSbSwTR MOD*
shows (*qInitAbs MOD*) *respectsP alphaAbsGood*
unfolding *congruentP-def alphaAbsGood-def*
using *assms iwlsFSbSwTR-alphaAll-qInitAll* **by** *auto*

8.5.2 The initial morphism (iteration map) from the term model to any strong model

This morphism has the same definition for fresh-swap and fresh-subst strong models

definition *iterSTR* **where**
iterSTR MOD == *univ* (*qInit MOD*)

definition *iterAbsSTR* **where**
iterAbsSTR MOD == *univ* (*qInitAbs MOD*)

lemma *iwlsFSwSTR-iterSTR-ipresVar*:
assumes *iwlsFSwSTR MOD*
shows *ipresVar* (*iterSTR MOD*) *MOD*
using *assms* **by**(*simp add*: *ipresVar-def Var-def iterSTR-def iwlsFSwSTR-qInit-respectsP-alpha*)

lemma *iwlsFSbSwTR-iterSTR-ipresVar*:
assumes *iwlsFSbSwTR MOD*
shows *ipresVar* (*iterSTR MOD*) *MOD*
using *assms* **by** (*simp add*: *ipresVar-def Var-def iterSTR-def iwlsFSbSwTR-qInit-respectsP-alphaGood*)

lemma *iwlsFSwSTR-iterSTR-ipresAbs*:
assumes *iwlsFSwSTR MOD*
shows *ipresAbs* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresAbs-def* **proof** *clarify*
  fix *xs x s X* **assume** *X*: *wls s X*
  hence *qGood* (*pick X*) **by**(*simp add*: *good-imp-qGood-pick*)
  hence *1*: *qGoodAbs* (*qAbs xs x* (*pick X*)) **by** *simp*
  have *iterAbsSTR MOD* (*Abs xs x X*) = *univ* (*qInitAbs MOD*) (*asAbs* (*qAbs xs x*
(*pick X*)))
  using *X* **unfolding** *Abs-def iterAbsSTR-def* **by** *simp*
  also have ... = *qInitAbs MOD* (*qAbs xs x* (*pick X*))
  using *assms 1* **by**(*simp add*: *iwlsFSwSTR-qInitAbs-respectsP-alphaAbs*)
  also have ... = *igAbs MOD xs x* (*qInit MOD* (*pick X*)) **by** *simp*
  also have ... = *igAbs MOD xs x* (*iterSTR MOD X*) **unfolding** *iterSTR-def*

unfolding *univ-def pick-def* **..**
finally show *iterAbsSTR MOD* (*Abs xs x X*) = *igAbs MOD xs x* (*iterSTR MOD X*) **.**
qed

lemma *iwlsFSbSwTR-iterSTR-ipresAbs*:
assumes *iwlsFSbSwTR MOD*
shows *ipresAbs* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresAbs-def* **proof** *clarify*
fix *xs x s X* **assume** *X*: *wls s X*
hence *qGood* (*pick X*) **by**(*simp add*: *good-imp-qGood-pick*)
hence *1*: *qGoodAbs* (*qAbs xs x* (*pick X*)) **by** *simp*
have *iterAbsSTR MOD* (*Abs xs x X*) = *univ* (*qInitAbs MOD*) (*asAbs* (*qAbs xs x* (*pick X*)))
using *X* **unfolding** *Abs-def iterAbsSTR-def* **by** *simp*
also have ... = *qInitAbs MOD* (*qAbs xs x* (*pick X*))
using *assms 1* **by**(*simp add*: *iwlsFSbSwTR-qInitAbs-respectsP-alphaAbsGood*)
also have ... = *igAbs MOD xs x* (*qInit MOD* (*pick X*)) **by** *simp*
also have ... = *igAbs MOD xs x* (*iterSTR MOD X*) **unfolding** *iterSTR-def univ-def*
unfolding *univ-def pick-def* **..**
finally show *iterAbsSTR MOD* (*Abs xs x X*) = *igAbs MOD xs x* (*iterSTR MOD X*) **.**
qed

lemma *iwlsFSwSTR-iterSTR-ipresOp*:
assumes *iwlsFSwSTR MOD*
shows *ipresOp* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresOp-def* **proof** *clarify*
fix *delta inp binp*
assume *inp*: *wlsInp delta inp wlsBinp delta binp*
hence *qGoodInp* (*pickInp inp*) ∧ *qGoodBinp* (*pickBinp binp*)
by(*simp add*: *goodInp-imp-qGoodInp-pickInp goodBinp-imp-qGoodBinp-pickBinp*)
hence *1*: *qGood* (*qOp delta* (*pickInp inp*) (*pickBinp binp*)) **by** *simp*
have *iterSTR MOD* (*Op delta inp binp*) =
univ (*qInit MOD*) (*asTerm* (*qOp delta* (*pickInp inp*) (*pickBinp binp*)))
using *inp* **unfolding** *Op-def iterSTR-def* **by** *simp*
moreover have ... = *qInit MOD* (*qOp delta* (*pickInp inp*) (*pickBinp binp*))
using *assms 1* **by**(*simp add*: *iwlsFSwSTR-qInit-respectsP-alpha*)
moreover have ... = *igOp MOD delta* (*lift* (*qInit MOD*) (*pickInp inp*))
(*lift* (*qInitAbs MOD*) (*pickBinp binp*)) **by** *auto*
moreover
have *lift* (*qInit MOD*) (*pickInp inp*) = *lift* (*iterSTR MOD*) *inp* ∧
lift (*qInitAbs MOD*) (*pickBinp binp*) = *lift* (*iterAbsSTR MOD*) *binp*
unfolding *pickInp-def pickBinp-def iterSTR-def iterAbsSTR-def*
lift-comp univ-def[*abs-def*] *comp-def*
unfolding *univ-def pick-def* **by** *simp*
ultimately
show *iterSTR MOD* (*Op delta inp binp*) =

igOp MOD delta (*lift* (*iterSTR MOD*) *inp*) (*lift* (*iterAbsSTR MOD*) *binp*)
by *simp*
qed

lemma *iwlsFSbSwTR-iterSTR-ipresOp*:
assumes *iwlsFSbSwTR MOD*
shows *ipresOp* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresOp-def* **proof** *clarify*
fix *delta inp binp*
assume *inp*: *wlsInp delta inp* *wlsBinp delta binp*
hence *qGoodInp* (*pickInp inp*) ∧ *qGoodBinp* (*pickBinp binp*)
by(*simp add*: *goodInp-imp-qGoodInp-pickInp goodBinp-imp-qGoodBinp-pickBinp*)
hence *1*: *qGood* (*qOp delta* (*pickInp inp*) (*pickBinp binp*)) **by** *simp*
have *iterSTR MOD* (*Op delta inp binp*) =
univ (*qInit MOD*) (*asTerm* (*qOp delta* (*pickInp inp*) (*pickBinp binp*)))
using *inp* **unfolding** *Op-def iterSTR-def* **by** *simp*
moreover have ... = *qInit MOD* (*qOp delta* (*pickInp inp*) (*pickBinp binp*))
using *assms 1* **by**(*simp add*: *iwlsFSbSwTR-qInit-respectsP-alphaGood*)
moreover have ... = *igOp MOD delta* (*lift* (*qInit MOD*) (*pickInp inp*))
(*lift* (*qInitAbs MOD*) (*pickBinp binp*)) **by** *simp*
moreover have *lift* (*qInit MOD*) (*pickInp inp*) = *lift* (*iterSTR MOD*) *inp* ∧
lift (*qInitAbs MOD*) (*pickBinp binp*) = *lift* (*iterAbsSTR MOD*) *binp*
unfolding *pickInp-def pickBinp-def iterSTR-def iterAbsSTR-def*
lift-comp univ-def[*abs-def*] *comp-def*
unfolding *univ-def pick-def* **by** *simp*
ultimately
show *iterSTR MOD* (*Op delta inp binp*) =
igOp MOD delta (*lift* (*iterSTR MOD*) *inp*) (*lift* (*iterAbsSTR MOD*) *binp*)
by *simp*
qed

lemma *iwlsFSwSTR-iterSTR-ipresCons*:
assumes *iwlsFSwSTR MOD*
shows *ipresCons* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresCons-def* **using** *assms*
iwlsFSwSTR-iterSTR-ipresVar
iwlsFSwSTR-iterSTR-ipresAbs
iwlsFSwSTR-iterSTR-ipresOp **by** *auto*

lemma *iwlsFSbSwTR-iterSTR-ipresCons*:
assumes *iwlsFSbSwTR MOD*
shows *ipresCons* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
unfolding *ipresCons-def* **using** *assms*
iwlsFSbSwTR-iterSTR-ipresVar
iwlsFSbSwTR-iterSTR-ipresAbs
iwlsFSbSwTR-iterSTR-ipresOp **by** *auto*

lemma *iwlsFSwSTR-iterSTR-termFSwImorph*:
assumes *iwlsFSwSTR MOD*

shows *termFSwImorph* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
using *assms* **by** (*auto simp*: *iwlsFSwSTR-termFSwImorph-iff intro*: *iwlsFSwSTR-iterSTR-ipresCons*)

corollary *iterSTR-termFSwImorph-errMOD*:
assumes *iwlsFSw MOD*
shows
termFSwImorph (*iterSTR* (*errMOD MOD*))
(*iterAbsSTR* (*errMOD MOD*))
(*errMOD MOD*)
using *assms errMOD-iwlsFSwSTR iwlsFSwSTR-iterSTR-termFSwImorph* **by** *auto*

lemma *iwlsFSbSwTR-iterSTR-termFSbImorph*:
assumes *iwlsFSbSwTR MOD*
shows *termFSbImorph* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
using *assms* **by** (*auto simp*: *iwlsFSbSwTR-termFSbImorph-iff intro*: *iwlsFSbSwTR-iterSTR-ipresCons*)

corollary *iterSTR-termFSbImorph-errMOD*:
assumes *iwlsFSb MOD*
shows
termFSbImorph (*iterSTR* (*errMOD MOD*))
(*iterAbsSTR* (*errMOD MOD*))
(*errMOD MOD*)
using *assms errMOD-iwlsFSbSwTR iwlsFSbSwTR-iterSTR-termFSbImorph* **by** *auto*

declare *qItem-simps*[*simp del*]
declare *qItem-versus-item-simps*[*simp del*]
declare *good-item-simps*[*simp del*]

8.5.3 The initial morhpism (iteration map) from the term model to any model

Again, this morphism has the same definition for fresh-swap and fresh-subst models, as well as (of course) for fresh-swap-subst and fresh-subst-swap models. (Remember that there is no such thing as "fresh-subst-swap" morhpism – we use the notion of "fresh-swap-subst" morphism.)

Existence of the morphism:

definition *iter* **where**
iter MOD == *check o* (*iterSTR* (*errMOD MOD*))

definition *iterAbs* **where**
iterAbs MOD == *check o* (*iterAbsSTR* (*errMOD MOD*))

theorem *iwlsFSw-iterAll-termFSwImorph*:
iwlsFSw MOD $\Longrightarrow$ *termFSwImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
using *iterSTR-termFSwImorph-errMOD check-FSwImorph*
by (*auto simp*: *iter-def iterAbs-def intro*: *comp-termFSwImorph*)

theorem *iwlsFSb-iterAll-termFSbImorph*:
iwlsFSb MOD ⟹ *termFSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
using *iterSTR-termFSbImorph-errMOD check-FSbImorph*
by (*auto simp*: *iter-def iterAbs-def intro*: *comp-termFSbImorph*)

theorem *iwlsFSwSb-iterAll-termFSwSbImorph*:
iwlsFSwSb MOD ⟹ *termFSwSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
using *iwlsFSw-iterAll-termFSwImorph*
by (*auto simp*: *iwlsFSwSb-termFSwSbImorph-iff iwlsFSwSb-def termFSwImorph-def*)

theorem *iwlsFSbSw-iterAll-termFSwSbImorph*:
iwlsFSbSw MOD ⟹ *termFSwSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
using *iwlsFSb-iterAll-termFSbImorph*
by (*auto simp*: *iwlsFSbSw-termFSwSbImorph-iff iwlsFSbSw-def termFSbImorph-def*)

Uniqueness of the morphism

In fact, already a presumptive construct-preserving map has to be unique:

lemma *ipresCons-unique*:
assumes *ipresCons f fA MOD* **and** *ipresCons ig igA MOD*
shows
(*wls s X* ⟶ *f X* = *ig X*) ∧
(*wlsAbs* (*us*,*s'*) *A* ⟶ *fA A* = *igA A*)
proof(*induction rule*: *wls-rawInduct*)
  case (*Var xs x*)
  then show *?case* **using** *assms* **unfolding** *ipresCons-def ipresVar-def* **by** *simp*
next
  case (*Op delta inp binp*)
  hence *lift f inp* = *lift ig inp* ∧ *lift fA binp* = *lift igA binp*
  using *assms*
  apply(*simp add*: *lift-def liftAll2-def sameDom-def fun-eq-iff wlsInp-iff wlsBinp-iff split*: *option.splits*)
  using *not-None-eq* **by** (*metis surj-pair*)
  thus *f* (*Op delta inp binp*) = *ig* (*Op delta inp binp*)
  using *assms* **unfolding** *ipresCons-def ipresOp-def* **by** (*simp add*: *Op.IH*)
next
  case (*Abs s xs x X*)
  then show *?case* **using** *assms* **unfolding** *ipresCons-def ipresAbs-def* **apply** *clarify*
  by (*erule allE*[*of - xs*], *erule allE*[*of - x*]) *fastforce*
qed

theorem *iwlsFSw-iterAll-unique-ipresCons*:
assumes *iwlsFSw MOD* **and** *ipresCons h hA MOD*
shows
(*wls s X* ⟶ *h X* = *iter MOD X*) ∧
(*wlsAbs* (*us*,*s'*) *A* ⟶ *hA A* = *iterAbs MOD A*)
using *assms iwlsFSw-iterAll-termFSwImorph*
by (*auto simp*: *termFSwImorph-def intro*!: *ipresCons-unique*)

theorem *iwlsFSb-iterAll-unique-ipresCons*:
assumes *iwlsFSb MOD* **and** *ipresCons h hA MOD*
shows
$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$
using *assms iwlsFSb-iterAll-termFSbImorph*
by (*auto simp*: *termFSbImorph-def intro*!: *ipresCons-unique*)

theorem *iwlsFSwSb-iterAll-unique-ipresCons*:
assumes *iwlsFSwSb MOD* **and** *ipresCons h hA MOD*
shows
$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$
using *assms* **unfolding** *iwlsFSwSb-def*
using *iwlsFSw-iterAll-unique-ipresCons* **by** *blast*

theorem *iwlsFSbSw-iterAll-unique-ipresCons*:
assumes ∗: *iwlsFSbSw MOD* **and** ∗∗: *ipresCons h hA MOD*
shows
$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$
using *assms* **unfolding** *iwlsFSbSw-def*
using *iwlsFSb-iterAll-unique-ipresCons* **by** *blast*

lemmas *iteration-simps* =
input-igSwap-igSubst-None
termMOD-simps
error-model-simps

declare *iteration-simps* [*simp del*]

end

end

9 Interpretation of syntax in semantic domains

theory *Semantic-Domains* **imports** *Iteration*
begin

In this section, we employ our iteration principle to obtain interpretation of syntax in semantic domains via valuations. A bonus from our Horn-theoretic approach is the built-in commutation of the interpretation with substitution versus valuation update, a property known in the literature as the “substitution lemma".

9.1 Semantic domains and valuations

Semantic domains are for binding signatures what algebras are for standard algebraic signatures. They fix carrier sets for each sort, and interpret each operation symbol as an operation on these sets [6] of corresponding arity, where:
- non-binding arguments are treated as usual (first-order) arguments;
- binding arguments are treated as second-order (functional) arguments. [7]

In particular, for the untyped and simply-typed λ-calculi, the semantic domains become the well-known (set-theoretic) Henkin models.

We use terminology and notation according to our general methodology employed so far: the inhabitants of semantic domains are referred to as "semantic items"; we prefix the reference to semantic items with an "s": sX, sA, etc. This convention also applies to the operations on semantic domains: "sAbs", "sOp", etc.

We eventually show that the function spaces consisting of maps from valuations to semantic items form models; in other words, these maps can be viewed as "generalized items"; we use for them term-like notations "X", "A", etc. (as we did in the theory that dealt with iteration).

9.1.1 Definitions:

datatype *('varSort,'sTerm)sAbs = sAbs 'varSort 'sTerm ⇒ 'sTerm*

record *('index,'bindex,'varSort,'sort,'opSym,'sTerm)semDom =*
sWls :: 'sort ⇒ 'sTerm ⇒ bool
sDummy :: 'sort ⇒ 'sTerm
sOp :: 'opSym ⇒ ('index,'sTerm)input ⇒ ('bindex,('varSort,'sTerm)sAbs)input ⇒'sTerm

The type of valuations:

type-synonym *('varSort,'var,'sTerm)val = 'varSort ⇒ 'var ⇒ 'sTerm*

context *FixSyn*
begin

fun *sWlsAbs* **where**
sWlsAbs SEM (xs,s) (sAbs xs' sF) =

[6] To match the Isabelle type system, we model (as usual) the family of carrier sets as a "well-sortedness" predicate taking sorts and semantic items from a given (initially unsorted) universe into booleans, and require the operations, considered on the unsorted universe, to preserve well-sortedness.

[7] In other words, syntactic bindings are captured semantically as functional bindings.

$(isInBar\ (xs,s) \wedge xs = xs' \wedge$
$(\forall\ sX.\ if\ sWls\ SEM\ (asSort\ xs)\ sX$
$then\ sWls\ SEM\ s\ (sF\ sX)$
$else\ sF\ sX = sDummy\ SEM\ s))$

definition *sWlsInp* **where**
$sWlsInp\ SEM\ delta\ sinp \equiv$
$wlsOpS\ delta \wedge sameDom\ (arOf\ delta)\ sinp \wedge liftAll2\ (sWls\ SEM)\ (arOf\ delta)\ sinp$

definition *sWlsBinp* **where**
$sWlsBinp\ SEM\ delta\ sbinp \equiv$
$wlsOpS\ delta \wedge sameDom\ (barOf\ delta)\ sbinp \wedge liftAll2\ (sWlsAbs\ SEM)\ (barOf\ delta)\ sbinp$

definition *sWlsNE* **where**
$sWlsNE\ SEM \equiv$
$\forall\ s.\ \exists\ sX.\ sWls\ SEM\ s\ sX$

definition *sWlsDisj* **where**
$sWlsDisj\ SEM \equiv$
$\forall\ s\ s'\ sX.\ sWls\ SEM\ s\ sX \wedge sWls\ SEM\ s'\ sX \longrightarrow s = s'$

definition *sOpPrSWls* **where**
$sOpPrSWls\ SEM \equiv$
$\forall\ delta\ sinp\ sbinp.$
$sWlsInp\ SEM\ delta\ sinp \wedge sWlsBinp\ SEM\ delta\ sbinp$
$\longrightarrow sWls\ SEM\ (stOf\ delta)\ (sOp\ SEM\ delta\ sinp\ sbinp)$

The notion of a “well-sorted" (better read as “well-structured") semantic domain: [8]

definition *wlsSEM* **where**
$wlsSEM\ SEM \equiv$
$sWlsNE\ SEM \wedge sWlsDisj\ SEM \wedge sOpPrSWls\ SEM$

The preperties described in the next 4 definitions turn out to be consequences of the well-structuredness of the semantic domain:

definition *sWlsAbsNE* **where**
$sWlsAbsNE\ SEM \equiv$
$\forall\ us\ s.\ isInBar\ (us,s) \longrightarrow (\exists\ sA.\ sWlsAbs\ SEM\ (us,s)\ sA)$

definition *sWlsAbsDisj* **where**
$sWlsAbsDisj\ SEM \equiv$
$\forall\ us\ s\ us'\ s'\ sA.$
$isInBar\ (us,s) \wedge isInBar\ (us',s') \wedge sWlsAbs\ SEM\ (us,s)\ sA \wedge sWlsAbs\ SEM\ (us',s')\ sA$

[8] As usual in Isabelle, we first define the “raw" version, and then “fix" it with a well-structuredness predicate.

$\longrightarrow us = us' \wedge s = s'$

The notion of two valuations being equal everywhere but on a given variable:

definition *eqBut* **where**
eqBut val val' xs x ≡
∀ *ys y*. (*ys* = *xs* ∧ *y* = *x*) ∨ *val ys y* = *val' ys y*

definition *updVal* ::
(*'varSort*,*'var*,*'sTerm*)*val* ⇒
'var ⇒ *'sTerm* ⇒ *'varSort* ⇒
(*'varSort*,*'var*,*'sTerm*)*val* (‹- '(- := -')'--› *200*)
where
(*val* (*x* := *sX*)-*xs*) ≡
λ *ys y*. (*if ys* = *xs* ∧ *y* = *x then sX else val ys y*)

definition *swapVal* ::
'varSort ⇒ *'var* ⇒ *'var* ⇒ (*'varSort*,*'var*,*'sTerm*)*val* ⇒
(*'varSort*,*'var*,*'sTerm*)*val*
where
swapVal zs z1 z2 val ≡ λ*xs x*. *val xs* (*x* @*xs*[*z1* ∧ *z2*]-*zs*)

abbreviation *swapVal-abbrev* (‹- ^[- ∧ -]'--› *200*) **where**
val ^[*z1* ∧ *z2*]-*zs* ≡ *swapVal zs z1 z2 val*

definition *sWlsVal* **where**
sWlsVal SEM val ≡
∀ *ys y*. *sWls SEM* (*asSort ys*) (*val ys y*)

definition *sWlsValNE* ::
(*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'sTerm*)*semDom* ⇒ *'var* ⇒ *bool*
where
sWlsValNE SEM x ≡ ∃ (*val* :: (*'varSort*,*'var*,*'sTerm*)*val*). *sWlsVal SEM val*

9.1.2 Basic facts

lemma *sWlsNE-imp-sWlsAbsNE*:
assumes *sWlsNE SEM*
shows *sWlsAbsNE SEM*
unfolding *sWlsAbsNE-def* **proof** *clarify*
  fix *xs s*
  obtain *sY* **where** *sWls SEM s sY*
  using *assms* **unfolding** *sWlsNE-def* **by** *auto*
  moreover assume *isInBar* (*xs*,*s*)
  ultimately
  have *sWlsAbs SEM* (*xs*,*s*) (*sAbs xs* (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
                                        then sY
                                        else sDummy SEM s)) **by** *simp*

thus ∃ *sA*. *sWlsAbs SEM* (*xs*,*s*) *sA* **by** *blast*
qed

lemma *sWlsDisj-imp-sWlsAbsDisj*:
sWlsDisj SEM ⟹ *sWlsNE SEM* ⟹ *sWlsAbsDisj SEM*
by (*simp add*: *sWlsAbsDisj-def sWlsNE-def sWlsDisj-def*)
(*smt* (*verit*) *prod.inject sAbs.inject sWlsAbs.elims*(*2*))

lemma *sWlsNE-imp-sWlsValNE*:
sWlsNE SEM ⟹ *sWlsValNE SEM x*
by (*auto simp*: *sWlsNE-def sWlsValNE-def sWlsVal-def*
intro!: *exI someI-ex*[*of* (λ *sY*. *sWls SEM* (*asSort* -) *sY*)])

theorem *updVal-simp*[*simp*]:
(*val* (*x* := *sX*)-*xs*) *ys y* = (*if ys* = *xs* ∧ *y* = *x then sX else val ys y*)
unfolding *updVal-def* **by** *simp*

theorem *updVal-over*[*simp*]:
((*val* (*x* := *sX*)-*xs*) (*x* := *sX'*)-*xs*) = (*val* (*x* := *sX'*)-*xs*)
unfolding *updVal-def* **by** *fastforce*

theorem *updVal-commute*:
assumes *xs* ≠ *ys* ∨ *x* ≠ *y*
shows ((*val* (*x* := *sX*)-*xs*) (*y* := *sY*)-*ys*) = ((*val* (*y* := *sY*)-*ys*) (*x* := *sX*)-*xs*)
using *assms* **unfolding** *updVal-def* **by** *fastforce*

theorem *updVal-preserves-sWls*[*simp*]:
assumes *sWls SEM* (*asSort xs*) *sX* **and** *sWlsVal SEM val*
shows *sWlsVal SEM* (*val* (*x* := *sX*)-*xs*)
using *assms* **unfolding** *sWlsVal-def* **by** *auto*

lemmas *updVal-simps* = *updVal-simp updVal-over updVal-preserves-sWls*

theorem *swapVal-ident*[*simp*]: (*val* ^[*x* ∧ *x*]-*xs*) = *val*
unfolding *swapVal-def* **by** *auto*

theorem *swapVal-compose*:
((*val* ^[*x* ∧ *y*]-*zs*) ^[*x'* ∧ *y'*]-*zs'*) =
((*val* ^[*x'* @*zs'*[*x* ∧ *y*]-*zs* ∧ *y'* @*zs'*[*x* ∧ *y*]-*zs*]-*zs'*) ^[*x* ∧ *y*]-*zs*)
unfolding *swapVal-def* **by** (*metis sw-compose*)

theorem *swapVal-commute*:
zs ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {} ⟹
((*val* ^[*x* ∧ *y*]-*zs*) ^[*x'* ∧ *y'*]-*zs'*) = ((*val* ^[*x'* ∧ *y'*]-*zs'*) ^[*x* ∧ *y*]-*zs*)
using *swapVal-compose*[*of zs' x' y' zs x y val*] **by**(*simp add*: *sw-def*)

lemma *swapVal-involutive*[*simp*]: ((*val* ^[*x* ∧ *y*]-*zs*) ^[*x* ∧ *y*]-*zs*) = *val*
unfolding *swapVal-def* **by** *auto*

lemma *swapVal-sym*: $(val\ ^\frown[x \wedge y]\text{-}zs) = (val\ ^\frown[y \wedge x]\text{-}zs)$
unfolding *swapVal-def* **by**(*auto simp add*: *sw-sym*)

lemma *swapVal-preserves-sWls1*:
assumes *sWlsVal SEM val*
shows *sWlsVal SEM* $(val\ ^\frown[z1 \wedge z2]\text{-}zs)$
using *assms* **unfolding** *sWlsVal-def swapVal-def* **by** *simp*

theorem *swapVal-preserves-sWls*[*simp*]:
sWlsVal SEM $(val\ ^\frown[z1 \wedge z2]\text{-}zs)$ = *sWlsVal SEM val*
using *swapVal-preserves-sWls1*[*of - - zs z1 z2*] **by** *fastforce*

lemmas *swapVal-simps* = *swapVal-ident swapVal-involutive swapVal-preserves-sWls*

lemma *updVal-swapVal*:
$((val\ (x := sX)\text{-}xs)\ ^\frown[y1 \wedge y2]\text{-}ys) =$
$((val\ ^\frown[y1 \wedge y2]\text{-}ys)\ ((x\ @xs[y1 \wedge y2]\text{-}ys) := sX)\text{-}xs)$
unfolding *swapVal-def* **by** *fastforce*

lemma *updVal-preserves-eqBut*:
assumes *eqBut val val′ ys y*
shows *eqBut* $(val\ (x := sX)\text{-}xs)\ (val'\ (x := sX)\text{-}xs)$ *ys y*
using *assms* **unfolding** *eqBut-def updVal-def* **by** *auto*

lemma *updVal-eqBut-eq*:
assumes *eqBut val val′ ys y*
shows $(val\ (y := sY)\text{-}ys) = (val'\ (y := sY)\text{-}ys)$
using *assms* **unfolding** *eqBut-def* **by** *fastforce*

lemma *swapVal-preserves-eqBut*:
assumes *eqBut val val′ xs x*
shows *eqBut* $(val\ ^\frown[z1 \wedge z2]\text{-}zs)\ (val'\ ^\frown[z1 \wedge z2]\text{-}zs)$ *xs* $(x\ @xs[z1 \wedge z2]\text{-}zs)$
using *assms* **unfolding** *eqBut-def swapVal-def* **by** *force*

9.2 Interpretation maps

An interpretation map, of syntax in a semantic domain, is the usual one w.r.t. valuations. Here we state its compostionality conditions (including the “substitution lemma"), and later we prove the existence of a map satisfying these conditions.

9.2.1 Definitions

Below, prefix “pr" means “preserves".

definition *prWls* **where**
prWls g SEM ≡ ∀ *s X val*.
  wls s X ∧ *sWlsVal SEM val*
  ⟶ *sWls SEM s* (*g X val*)

definition *prWlsAbs* **where**
prWlsAbs gA SEM ≡ ∀ *us s A val.*
wlsAbs (*us*,*s*) *A* ∧ *sWlsVal SEM val*
⟶ *sWlsAbs SEM* (*us*,*s*) (*gA A val*)

definition *prWlsAll* **where**
prWlsAll g gA SEM ≡ *prWls g SEM* ∧ *prWlsAbs gA SEM*

definition *prVar* **where**
prVar g SEM ≡ ∀ *xs x val.*
sWlsVal SEM val ⟶ *g* (*Var xs x*) *val* = *val xs x*

definition *prAbs* **where**
prAbs g gA SEM ≡ ∀ *xs s x X val.*
isInBar (*xs*,*s*) ∧ *wls s X* ∧ *sWlsVal SEM val*
⟶
gA (*Abs xs x X*) *val* =
sAbs xs (λ*sX. if sWls SEM* (*asSort xs*) *sX then g X* (*val* (*x* := *sX*)-*xs*)
else sDummy SEM s)

definition *prOp* **where**
prOp g gA SEM ≡ ∀ *delta inp binp val.*
wlsInp delta inp ∧ *wlsBinp delta binp* ∧ *sWlsVal SEM val*
⟶
g (*Op delta inp binp*) *val* =
sOp SEM delta (*lift* (λ*X. g X val*) *inp*)
(*lift* (λ*A. gA A val*) *binp*)

definition *prCons* **where**
prCons g gA SEM ≡ *prVar g SEM* ∧ *prAbs g gA SEM* ∧ *prOp g gA SEM*

definition *prFresh* **where**
prFresh g SEM ≡ ∀ *ys y s X val val′.*
wls s X ∧ *fresh ys y X* ∧
sWlsVal SEM val ∧ *sWlsVal SEM val′* ∧ *eqBut val val′ ys y*
⟶ *g X val* = *g X val′*

definition *prFreshAbs* **where**
prFreshAbs gA SEM ≡ ∀ *ys y us s A val val′.*
wlsAbs (*us*,*s*) *A* ∧ *freshAbs ys y A* ∧
sWlsVal SEM val ∧ *sWlsVal SEM val′* ∧ *eqBut val val′ ys y*
⟶ *gA A val* = *gA A val′*

definition *prFreshAll* **where**
prFreshAll g gA SEM ≡ *prFresh g SEM* ∧ *prFreshAbs gA SEM*

definition *prSwap* **where**
prSwap g SEM ≡ ∀ *zs z1 z2 s X val.*

wls s X ∧ *sWlsVal SEM val*
⟶
g (*X* #[*z1* ∧ *z2*]-*zs*) *val* =
g X (*val* ^[*z1* ∧ *z2*]-*zs*)

definition *prSwapAbs* **where**
prSwapAbs gA SEM ≡ ∀ *zs z1 z2 us s A val*.
wlsAbs (*us*,*s*) *A* ∧ *sWlsVal SEM val*
⟶
gA (*A* $[*z1* ∧ *z2*]-*zs*) *val* =
gA A (*val* ^[*z1* ∧ *z2*]-*zs*)

definition *prSwapAll* **where**
prSwapAll g gA SEM ≡ *prSwap g SEM* ∧ *prSwapAbs gA SEM*

definition *prSubst* **where**
prSubst g SEM ≡ ∀ *ys Y y s X val*.
wls (*asSort ys*) *Y* ∧ *wls s X*
∧ *sWlsVal SEM val*
⟶
g (*X* #[*Y* / *y*]-*ys*) *val* =
g X (*val* (*y* := *g Y val*)-*ys*)

definition *prSubstAbs* **where**
prSubstAbs g gA SEM ≡ ∀ *ys Y y us s A val*.
wls (*asSort ys*) *Y* ∧ *wlsAbs* (*us*,*s*) *A*
∧ *sWlsVal SEM val*
⟶
gA (*A* $[*Y* / *y*]-*ys*) *val* =
gA A (*val* (*y* := *g Y val*)-*ys*)

definition *prSubstAll* **where**
prSubstAll g gA SEM ≡ *prSubst g SEM* ∧ *prSubstAbs g gA SEM*

definition *compInt* **where**
compInt g gA SEM ≡ *prWlsAll g gA SEM* ∧ *prCons g gA SEM* ∧
prFreshAll g gA SEM ∧ *prSwapAll g gA SEM* ∧ *prSubstAll g gA SEM*

9.2.2 Extension of domain preservation to inputs

lemma *prWls-wlsInp*:
assumes *wlsInp delta inp* **and** *prWls g SEM* **and** *sWlsVal SEM val*
shows *sWlsInp SEM delta* (*lift* (λ *X*. *g X val*) *inp*)
using *assms* **unfolding** *sWlsInp-def wlsInp-iff liftAll2-def lift-def prWls-def*
by (*auto simp add*: *option.case-eq-if sameDom-def*)

lemma *prWlsAbs-wlsBinp*:
assumes *wlsBinp delta binp* **and** *prWlsAbs gA SEM* **and** *sWlsVal SEM val*
shows *sWlsBinp SEM delta* (*lift* (λ *A*. *gA A val*) *binp*)

using *assms* **unfolding** *sWlsBinp-def wlsBinp-iff liftAll2-def lift-def prWlsAbs-def*
by (*auto simp add*: *option.case-eq-if sameDom-def*)

end

9.3 The iterative model associated to a semantic domain

"asIMOD SEM" stands for "SEM (regarded) as a model". [9] The associated model is built essentially as follows:
- Its carrier sets consist of functions from valuations to semantic items.
- The construct operations (i.e., those corresponding to the syntactic constructs indicated in the given binding signature) are lifted componentwise from those of the semantic domain "SEM" (also taking into account the higher-order nature of of the semantic counterparts of abstractions).
- For a map from valuations to items (terms or abstractions), freshness of a variable "x" is defined as being oblivious what the argument valuation returns for "x".
- Swapping is defined componentwise, by two iterations of the notion of swapping the returned value of a function.
- Substitution of a semantic term "Y" for a variable "y" is a semantic term "X" is defined to map each valuation "val" to the application of "X" to ["val" updated at "y" with whatever "Y" returns for "val"].

Note that:
- The construct operations definitions are determined by the desired clauses of the standard notion of interpreting syntax in a semantic domains.
- Substitution and freshness are defined having in mind the (again standard) facts of the interpretation commuting with substitution versus valuation update and the interpretation being oblivious to the valuation of fresh variables.

9.3.1 Definition and basic facts

The next two types of "generalized items" are used to build models from semantic domains: [10]

type-synonym $('varSort,'var,'sTerm)\ gTerm = ('varSort,'var,'sTerm)val \Rightarrow 'sTerm$

type-synonym $('varSort,'var,'sTerm)\ gAbs = ('varSort,'var,'sTerm)val \Rightarrow ('varSort,'sTerm)sAbs$

context *FixSyn*
begin

[9] We use the word "model" as introduced in the theory "Models-and-Recursion".
[10] Recall that "generalized items" inhabit models.

definition *asIMOD* ::
$('index,'bindex,'varSort,'sort,'opSym,'sTerm)semDom \Rightarrow$
$('index,'bindex,'varSort,'sort,'opSym,'var,$
$('varSort,'var,'sTerm)gTerm,$
$('varSort,'var,'sTerm)gAbs)model$
where
asIMOD SEM ≡
(|*igWls* = λ*s X*. ∀ *val*. (*sWlsVal SEM val* ∨ *X val* = *undefined*) ∧
(*sWlsVal SEM val* ⟶ *sWls SEM s* (*X val*)),
igWlsAbs = λ(*xs*,*s*) *A*. ∀ *val*. (*sWlsVal SEM val* ∨ *A val* = *undefined*) ∧
(*sWlsVal SEM val* ⟶ *sWlsAbs SEM* (*xs*,*s*) (*A val*)),
igVar = λ*ys y*. λ*val*. *if sWlsVal SEM val then val ys y else undefined*,
igAbs =
λ*xs x X*. λ*val*. *if sWlsVal SEM val*
then sAbs xs (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
then X (*val* (*x* := *sX*)-*xs*)
else sDummy SEM (*SOME s*. *sWls SEM s* (*X*
val)))
else undefined,
igOp = λ*delta inp binp*. λ*val*.
if sWlsVal SEM val then sOp SEM delta (*lift* (λ*X*. *X val*) *inp*)
(*lift* (λ*A*. *A val*) *binp*)
else undefined,
igFresh =
λ*ys y X*. ∀ *val val'*. *sWlsVal SEM val* ∧ *sWlsVal SEM val'* ∧ *eqBut val val' ys y*
⟶ *X val* = *X val'*,
igFreshAbs =
λ*ys y A*. ∀ *val val'*. *sWlsVal SEM val* ∧ *sWlsVal SEM val'* ∧ *eqBut val val' ys y*
⟶ *A val* = *A val'*,
igSwap = λ*zs z1 z2 X*. λ*val*. *if sWlsVal SEM val then X* (*val* ^[*z1* ∧ *z2*]-*zs*)
else undefined,
igSwapAbs = λ*zs z1 z2 A*. λ*val*. *if sWlsVal SEM val then A* (*val* ^[*z1* ∧ *z2*]-*zs*)
else undefined,
igSubst = λ*ys Y y X*. λ*val*. *if sWlsVal SEM val then X* (*val* (*y* := *Y val*)-*ys*)
else undefined,
igSubstAbs = λ*ys Y y A*. λ*val*. *if sWlsVal SEM val then A* (*val* (*y* := *Y val*)-*ys*)
else undefined|)

Next we state, as usual, the direct definitions of the operators and relations of associated model, freeing ourselves from having to go through the "asIMOD" definition each time we reason about them.

lemma *asIMOD-igWls*:
igWls (*asIMOD SEM*) *s X* ⟷
(∀ *val*. (*sWlsVal SEM val* ∨ *X val* = *undefined*) ∧
(*sWlsVal SEM val* ⟶ *sWls SEM s* (*X val*)))
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igWlsAbs*:
igWlsAbs (*asIMOD SEM*) (*us*,*s*) *A* ⟷

(∀ *val*. (*sWlsVal SEM val* ∨ *A val* = *undefined*) ∧
(*sWlsVal SEM val* ⟶ *sWlsAbs SEM* (*us*,*s*) (*A val*)))
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igOp*:
igOp (*asIMOD SEM*) *delta inp binp* =
(λ*val*. *if sWlsVal SEM val then sOp SEM delta* (*lift* (λ*X*. *X val*) *inp*)
(*lift* (λ*A*. *A val*) *binp*)
else undefined)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igVar*:
igVar (*asIMOD SEM*) *ys y* = (λ*val*. *if sWlsVal SEM val then val ys y else undefined*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igAbs*:
igAbs (*asIMOD SEM*) *xs x X* =
(λ*val*. *if sWlsVal SEM val then sAbs xs* (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
then X (*val* (*x* := *sX*)-*xs*)
else sDummy SEM (*SOME s*. *sWls SEM s* (*X val*)))
else undefined)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igAbs2*:
fixes *SEM* :: ('*index*,'*bindex*,'*varSort*,'*sort*,'*opSym*,'*sTerm*)*semDom*
assumes ∗: *sWlsDisj SEM* **and** ∗∗: *igWls* (*asIMOD SEM*) *s X*
shows *igAbs* (*asIMOD SEM*) *xs x X* =
(λ*val*. *if sWlsVal SEM val then sAbs xs* (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
then X (*val* (*x* := *sX*)-*xs*)
else sDummy SEM s)
else undefined)
proof−
{**fix** *val* :: ('*varSort*,'*var*,'*sTerm*)*val* **assume** *val*: *sWlsVal SEM val*
hence *Xval*: *sWls SEM s* (*X val*)
using ∗∗ **unfolding** *asIMOD-igWls* **by** *simp*
hence (*SOME s*. *sWls SEM s* (*X val*)) = *s*
using *Xval* ∗ **unfolding** *sWlsDisj-def* **by** *auto*
}
thus *?thesis* **unfolding** *asIMOD-igAbs* **by** *fastforce*
qed

lemma *asIMOD-igFresh*:
igFresh (*asIMOD SEM*) *ys y X* =
(∀ *val val*′. *sWlsVal SEM val* ∧ *sWlsVal SEM val*′ ∧ *eqBut val val*′ *ys y*
⟶ *X val* = *X val*′)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igFreshAbs*:
igFreshAbs (*asIMOD SEM*) *ys y A* =
(∀ *val val'*. *sWlsVal SEM val* ∧ *sWlsVal SEM val'* ∧ *eqBut val val' ys y*
⟶ *A val* = *A val'*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igSwap*:
igSwap (*asIMOD SEM*) *zs z1 z2 X* =
(λ*val*. *if sWlsVal SEM val then X* (*val* ^[*z1* ∧ *z2*]-*zs*) *else undefined*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igSwapAbs*:
igSwapAbs (*asIMOD SEM*) *zs z1 z2 A* =
(λ*val*. *if sWlsVal SEM val then A* (*val* ^[*z1* ∧ *z2*]-*zs*) *else undefined*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igSubst*:
igSubst (*asIMOD SEM*) *ys Y y X* =
(λ*val*. *if sWlsVal SEM val then X* (*val* (*y* := *Y val*)-*ys*) *else undefined*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igSubstAbs*:
igSubstAbs (*asIMOD SEM*) *ys Y y A* =
(λ*val*. *if sWlsVal SEM val then A* (*val* (*y* := *Y val*)-*ys*) *else undefined*)
unfolding *asIMOD-def* **by** *simp*

lemma *asIMOD-igWlsInp*:
assumes *sWlsNE SEM*
shows
igWlsInp (*asIMOD SEM*) *delta inp* ⟷
((∀ *val*. *liftAll* (λ*X*. *sWlsVal SEM val* ∨ *X val* = *undefined*) *inp*) ∧
(∀ *val*. *sWlsVal SEM val* ⟶ *sWlsInp SEM delta* (*lift* (λ*X*. *X val*) *inp*)))
using *assms* **apply** *safe*
subgoal by (*simp add*: *asIMOD-igWls liftAll-def liftAll2-def igWlsInp-def sameDom-def split*: *option.splits*) (*metis option.distinct*(*1*) *option.exhaust*)
subgoal by (*simp add*: *igWlsInp-def asIMOD-igWls liftAll-def liftAll2-def lift-def sWlsInp-def sameDom-def split*: *option.splits*)
subgoal by (*simp add*:*igWlsInp-def asIMOD-igWls liftAll-def liftAll2-def lift-def sWlsInp-def sameDom-def split*: *option.splits*)
(*metis* (*no-types*) *option.distinct*(*1*) *sWlsNE-imp-sWlsValNE sWlsValNE-def*) **.**

lemma *asIMOD-igSwapInp*:
sWlsVal SEM val ⟹
lift (λ*X*. *X val*) (*igSwapInp* (*asIMOD SEM*) *zs z1 z2 inp*) =
lift (λ*X*. *X* (*swapVal zs z1 z2 val*)) *inp*
by (*auto simp*: *igSwapInp-def asIMOD-igSwap lift-def split*: *option.splits*)

lemma *asIMOD-igSubstInp*:
sWlsVal SEM val ⟹

lift (*λX. X val*) (*igSubstInp* (*asIMOD SEM*) *ys Y y inp*) =
lift (*λX. X* (*val* (*y* := *Y val*)-*ys*)) *inp*
by (*auto simp*: *igSubstInp-def asIMOD-igSubst lift-def split*: *option.splits*)

lemma *asIMOD-igWlsBinp*:
assumes *sWlsNE SEM*
shows
igWlsBinp (*asIMOD SEM*) *delta binp* =
((∀ *val. liftAll* (*λX. sWlsVal SEM val* ∨ *X val* = *undefined*) *binp*) ∧
(∀ *val. sWlsVal SEM val* ⟶ *sWlsBinp SEM delta* (*lift* (*λX. X val*) *binp*)))
using *assms* **apply** *safe*
subgoal by (*simp add*: *asIMOD-igWlsAbs liftAll-def liftAll2-def igWlsBinp-def sameDom-def split*: *option.splits*)
(*metis option.distinct*(*1*) *option.exhaust surj-pair*)
subgoal by (*simp add*: *igWlsBinp-def asIMOD-igWlsAbs liftAll-def liftAll2-def lift-def sWlsBinp-def sameDom-def split*: *option.splits*)
subgoal by (*simp add*:*igWlsBinp-def asIMOD-igWlsAbs liftAll-def liftAll2-def lift-def sWlsBinp-def sameDom-def split*: *option.splits*)
(*metis* (*no-types*) *old.prod.exhaust option.distinct*(*1*) *option.exhaust sWlsNE-imp-sWlsValNE sWlsValNE-def*) **.**

lemma *asIMOD-igSwapBinp*:
sWlsVal SEM val ⟹
lift (*λA. A val*) (*igSwapBinp* (*asIMOD SEM*) *zs z1 z2 binp*) =
lift (*λA. A* (*swapVal zs z1 z2 val*)) *binp*
by (*auto simp*: *igSwapBinp-def asIMOD-igSwapAbs lift-def split*: *option.splits*)

lemma *asIMOD-igSubstBinp*:
sWlsVal SEM val ⟹
lift (*λA. A val*) (*igSubstBinp* (*asIMOD SEM*) *ys Y y binp*) =
lift (*λA. A* (*val* (*y* := *Y val*)-*ys*)) *binp*
by (*auto simp*: *igSubstBinp-def asIMOD-igSubstAbs lift-def split*: *option.splits*)

9.3.2 The associated model is well-structured

That is to say: it is a fresh-swap-subst and fresh-subst-swap model (hence of course also a fresh-swap and fresh-subst) model.

Domain disjointness:

lemma *asIMOD-igWlsDisj*:
sWlsNE SEM ⟹ *sWlsDisj SEM* ⟹ *igWlsDisj* (*asIMOD SEM*)
using *sWlsNE-imp-sWlsValNE*
by (*fastforce simp*: *igWlsDisj-def asIMOD-igWls sWlsValNE-def sWlsDisj-def*)

lemma *asIMOD-igWlsAbsDisj*:
sWlsNE SEM ⟹ *sWlsDisj SEM* ⟹ *igWlsAbsDisj* (*asIMOD SEM*)
using *sWlsNE-imp-sWlsValNE sWlsDisj-imp-sWlsAbsDisj*
by (*fastforce simp*: *igWlsAbsDisj-def asIMOD-igWlsAbs sWlsAbsDisj-def sWlsValNE-def*)

lemma *asIMOD-igWlsAllDisj*:
sWlsNE SEM $\Longrightarrow$ *sWlsDisj SEM* $\Longrightarrow$ *igWlsAllDisj* (*asIMOD SEM*)
unfolding *igWlsAllDisj-def* **using** *asIMOD-igWlsDisj asIMOD-igWlsAbsDisj* **by** *auto*

Only "bound arit" abstraction domains are inhabited:

lemma *asIMOD-igWlsAbsIsInBar*:
sWlsNE SEM $\Longrightarrow$ *igWlsAbsIsInBar* (*asIMOD SEM*)
using *sWlsNE-imp-sWlsValNE*
by (*auto simp*: *sWlsValNE-def igWlsAbsIsInBar-def asIMOD-igWlsAbs*
split: *option.splits elim*: *sWlsAbs.elims*(*2*))

Domain preservation by the operators

The constructs preserve the domains:

lemma *asIMOD-igVarIPresIGWls*: *igVarIPresIGWls* (*asIMOD SEM*)
unfolding *igVarIPresIGWls-def asIMOD-igWls asIMOD-igVar sWlsVal-def* **by** *simp*

lemma *asIMOD-igAbsIPresIGWls*:
sWlsDisj SEM $\Longrightarrow$ *igAbsIPresIGWls* (*asIMOD SEM*)
unfolding *igAbsIPresIGWls-def asIMOD-igWlsAbs* **apply** *clarify*
subgoal for - - - - *val*
unfolding *asIMOD-igAbs2* **by** (*cases sWlsVal SEM val*) (*auto simp*: *asIMOD-igWls*)
.

lemma *asIMOD-igOpIPresIGWls*:
sOpPrSWls SEM $\Longrightarrow$ *sWlsNE SEM* $\Longrightarrow$ *igOpIPresIGWls* (*asIMOD SEM*)
using *asIMOD-igWlsInp asIMOD-igWlsBinp*
by (*fastforce simp*: *igOpIPresIGWls-def asIMOD-igWls asIMOD-igOp sOpPrSWls-def*)

lemma *asIMOD-igConsIPresIGWls*:
wlsSEM SEM $\Longrightarrow$ *igConsIPresIGWls* (*asIMOD SEM*)
unfolding *igConsIPresIGWls-def wlsSEM-def*
using *asIMOD-igVarIPresIGWls asIMOD-igAbsIPresIGWls asIMOD-igOpIPresIGWls*
by *auto*

Swap preserves the domains:

lemma *asIMOD-igSwapIPresIGWls*: *igSwapIPresIGWls* (*asIMOD SEM*)
unfolding *igSwapIPresIGWls-def asIMOD-igSwap asIMOD-igWls* **by** *auto*

lemma *asIMOD-igSwapAbsIPresIGWlsAbs*: *igSwapAbsIPresIGWlsAbs* (*asIMOD SEM*)
unfolding *igSwapAbsIPresIGWlsAbs-def asIMOD-igSwapAbs asIMOD-igWlsAbs* **by** *auto*

lemma *asIMOD-igSwapAllIPresIGWlsAll*: *igSwapAllIPresIGWlsAll* (*asIMOD SEM*)
unfolding *igSwapAllIPresIGWlsAll-def*
using *asIMOD-igSwapIPresIGWls asIMOD-igSwapAbsIPresIGWlsAbs* **by** *auto*

Subst preserves the domains:

lemma *asIMOD-igSubstIPresIGWls*: *igSubstIPresIGWls* (*asIMOD SEM*)
unfolding *igSubstIPresIGWls-def asIMOD-igSubst asIMOD-igWls* **by** *simp*

lemma *asIMOD-igSubstAbsIPresIGWlsAbs*: *igSubstAbsIPresIGWlsAbs* (*asIMOD SEM*)
unfolding *igSubstAbsIPresIGWlsAbs-def asIMOD-igSubstAbs asIMOD-igWls asIMOD-igWlsAbs* **by** *simp*

lemma *asIMOD-igSubstAllIPresIGWlsAll*: *igSubstAllIPresIGWlsAll* (*asIMOD SEM*)
unfolding *igSubstAllIPresIGWlsAll-def*
using *asIMOD-igSubstIPresIGWls asIMOD-igSubstAbsIPresIGWlsAbs* **by** *auto*

The clauses for fresh hold:

lemma *asIMOD-igFreshIGVar*: *igFreshIGVar* (*asIMOD SEM*)
unfolding *igFreshIGVar-def asIMOD-igFresh asIMOD-igVar eqBut-def* **by** *force*

lemma *asIMOD-igFreshIGAbs1*:
sWlsDisj SEM $\Longrightarrow$ *igFreshIGAbs1* (*asIMOD SEM*)
by(*fastforce simp*: *igFreshIGAbs1-def asIMOD-igFresh asIMOD-igFreshAbs asIMOD-igAbs2 updVal-eqBut-eq*)

lemma *asIMOD-igFreshIGAbs2*:
sWlsDisj SEM $\Longrightarrow$ *igFreshIGAbs2* (*asIMOD SEM*)
by(*fastforce simp*: *igFreshIGAbs2-def asIMOD-igFresh asIMOD-igFreshAbs asIMOD-igAbs2 updVal-preserves-eqBut*)

lemma *asIMOD-igFreshIGOp*:
fixes *SEM* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'sTerm*)*semDom*
shows *igFreshIGOp* (*asIMOD SEM*)
unfolding *igFreshIGOp-def* **proof** *clarify*
  fix *ys y delta* **and** *inp* :: (*'index*, (*'varSort*,*'var*,*'sTerm*)*gTerm*)*input*
  and *binp* :: (*'bindex*, (*'varSort*,*'var*,*'sTerm*)*gAbs*)*input*
  assume *inp-fresh*: *igFreshInp* (*asIMOD SEM*) *ys y inp*
          igFreshBinp (*asIMOD SEM*) *ys y binp*
  show *igFresh* (*asIMOD SEM*) *ys y* (*igOp* (*asIMOD SEM*) *delta inp binp*)
  unfolding *asIMOD-igFresh asIMOD-igOp* **proof** *safe*
    fix *val val'*
    let *?sinp* = *lift* (λ*X*. *X val*) *inp* **let** *?sinp'* = *lift* (λ*X*. *X val'*) *inp*
    let *?sbinp* = *lift* (λ*A*. *A val*) *binp* **let** *?sbinp'* = *lift* (λ*A*. *A val'*) *binp*
    assume *wls*: *sWlsVal SEM val sWlsVal SEM val'* **and** *eqBut val val' ys y*
    hence *?sinp* = *?sinp'* ∧ *?sbinp* = *?sbinp'*
    using *inp-fresh*
    by (*auto simp*: *lift-def igFreshInp-def igFreshBinp-def errMOD-def liftAll-def asIMOD-igFresh asIMOD-igFreshAbs split*: *option.splits*)
    then show (*if sWlsVal SEM val then sOp SEM delta* (*lift* (λ*X*. *X val*) *inp*) (*lift* (λ*A*. *A val*) *binp*)
      else undefined) =
     (*if sWlsVal SEM val' then sOp SEM delta* (*lift* (λ*X*. *X val'*) *inp*) (*lift* (λ*A*.

A val') *binp*)
else undefined) **using** *wls* **by** *auto*
qed
qed

lemma *asIMOD-igFreshCls*:
assumes *sWlsDisj SEM*
shows *igFreshCls* (*asIMOD SEM*)
using *assms* **unfolding** *igFreshCls-def*
using *asIMOD-igFreshIGVar asIMOD-igFreshIGAbs1 asIMOD-igFreshIGAbs2 asI-MOD-igFreshIGOp* **by** *auto*

The clauses for swap hold:

lemma *asIMOD-igSwapIGVar*: *igSwapIGVar* (*asIMOD SEM*)
unfolding *igSwapIGVar-def* **apply** *clarsimp* **apply**(*rule ext*)
unfolding *asIMOD-igSwap asIMOD-igVar* **apply** *clarsimp*
unfolding *swapVal-def* **by** *simp*

lemma *asIMOD-igSwapIGAbs*: *igSwapIGAbs* (*asIMOD SEM*)
by (*fastforce simp*: *igSwapIGAbs-def asIMOD-igSwap asIMOD-igSwapAbs asIMOD-igAbs updVal-swapVal*)

lemma *asIMOD-igSwapIGOp*: *igSwapIGOp* (*asIMOD SEM*)
by (*auto simp*: *igSwapIGOp-def asIMOD-igSwap asIMOD-igOp asIMOD-igSwapInp asIMOD-igSwapBinp*)

lemma *asIMOD-igSwapCls*: *igSwapCls* (*asIMOD SEM*)
unfolding *igSwapCls-def* **using** *asIMOD-igSwapIGVar asIMOD-igSwapIGAbs asI-MOD-igSwapIGOp* **by** *auto*

The clauses for subst hold:

lemma *asIMOD-igSubstIGVar1*: *igSubstIGVar1* (*asIMOD SEM*)
by (*auto simp*: *igSubstIGVar1-def asIMOD-igSubst asIMOD-igVar asIMOD-igWls*)

lemma *asIMOD-igSubstIGVar2*: *igSubstIGVar2* (*asIMOD SEM*)
by (*fastforce simp*: *igSubstIGVar2-def asIMOD-igSubst asIMOD-igVar asIMOD-igWls*)

lemma *asIMOD-igSubstIGAbs*: *igSubstIGAbs* (*asIMOD SEM*)
unfolding *igSubstIGAbs-def* **proof**(*clarify*, *rule ext*)
fix *ys y Y xs x s X val*
assume *Y*: *igWls* (*asIMOD SEM*) (*asSort ys*) *Y*
and *X*: *igWls* (*asIMOD SEM*) *s X* **and** *x-diff-y*: $xs \neq ys \vee x \neq y$
and *x-fresh-Y*: *igFresh* (*asIMOD SEM*) *xs x Y*
show *igSubstAbs* (*asIMOD SEM*) *ys Y y* (*igAbs* (*asIMOD SEM*) *xs x X*) *val* =
igAbs (*asIMOD SEM*) *xs x* (*igSubst* (*asIMOD SEM*) *ys Y y X*) *val*
proof(*cases sWlsVal SEM val*)
case *False*
thus *?thesis* **unfolding** *asIMOD-igSubst asIMOD-igSubstAbs asIMOD-igAbs* **by** *simp*

next
case *True*
hence *Yval*: *sWls SEM* (*asSort ys*) (*Y val*)
using *Y* **unfolding** *asIMOD-igWls* **by** *simp*
{**fix** *sX* **assume** *sX*: *sWls SEM* (*asSort xs*) *sX*
let *?val-x* = *val* (*x* := *sX*)-*xs*
have *sWlsVal SEM ?val-x* **using** *True sX* **by** *simp*
moreover have *eqBut ?val-x val xs x*
unfolding *eqBut-def updVal-def* **by** *simp*
ultimately have *1*: *Y ?val-x* = *Y val*
using *True x-fresh-Y* **unfolding** *asIMOD-igFresh* **by** *simp*
let *?Left* = *X* ((*val* (*y* := *Y val*)-*ys*) (*x* := *sX*)-*xs*)
let *?Riight* = *X* (*?val-x* (*y* := *Y ?val-x*)-*ys*)
have *?Left* = *X* (*?val-x* (*y* := *Y val*)-*ys*)
using *x-diff-y* **by**(*auto simp add*: *updVal-commute*)
also have ... = *?Riight* **using** *1* **by** *simp*
finally have *?Left* = *?Riight* **.**
}
thus *?thesis* **using** *True Yval* **by**(*auto simp*: *asIMOD-igSubst asIMOD-igSubstAbs asIMOD-igAbs*)
qed
qed

lemma *asIMOD-igSubstIGOp*: *igSubstIGOp* (*asIMOD SEM*)
unfolding *igSubstIGOp-def* **proof**(*clarify*,*rule ext*)
fix *ys y Y delta inp binp val*
assume *Y*: *igWls* (*asIMOD SEM*) (*asSort ys*) *Y*
and *inp*: *igWlsInp* (*asIMOD SEM*) *delta inp*
and *binp*: *igWlsBinp* (*asIMOD SEM*) *delta binp*
define *inpsb binpsb* **where**
inpsb-def: *inpsb* ≡ *igSubstInp* (*asIMOD SEM*) *ys Y y inp*
binpsb ≡ *igSubstBinp* (*asIMOD SEM*) *ys Y y binp*
note *inpsb-rev* = *inpsb-def*[*symmetric*]
let *?sinpsb* = *lift* (λ*X*. *X* (*val* (*y* := *Y val*)-*ys*)) *inp*
let *?sbinpsb* = *lift* (λ*A*. *A* (*val* (*y* := *Y val*)-*ys*)) *binp*
show *igSubst* (*asIMOD SEM*) *ys Y y* (*igOp* (*asIMOD SEM*) *delta inp binp*) *val* =
igOp (*asIMOD SEM*) *delta* (*igSubstInp* (*asIMOD SEM*) *ys Y y inp*)
(*igSubstBinp* (*asIMOD SEM*) *ys Y y binp*) *val*
unfolding *inpsb-rev* **unfolding** *asIMOD-igSubst asIMOD-igOp* **unfolding** *inpsb-def*

apply(*simp add*: *asIMOD-igSubstInp asIMOD-igSubstBinp*)
using *Y* **unfolding** *asIMOD-def* **by** *auto*
qed

lemma *asIMOD-igSubstCls*: *igSubstCls* (*asIMOD SEM*)
unfolding *igSubstCls-def*
using *asIMOD-igSubstIGVar1 asIMOD-igSubstIGVar2 asIMOD-igSubstIGAbs asIMOD-igSubstIGOp* **by** *auto*

The fresh-swap-based congruence clause holds:

lemma *updVal-swapVal-eqBut*: *eqBut* (*val* (*x* := *sX*)-*xs*) ((*val* (*y* := *sX*)-*xs*) ^[*y* ∧ *x*]-*xs*) *xs y*
by (*simp add*: *updVal-def swapVal-def eqBut-def sw-def*)

lemma *asIMOD-igAbsCongS*: *sWlsDisj SEM* ⟹ *igAbsCongS* (*asIMOD SEM*)
unfolding *igAbsCongS-def asIMOD-igFresh asIMOD-igSwap asIMOD-igAbs2*
apply *safe* **apply** (*simp add*: *asIMOD-igAbs2*)
by (*rule ext*) (*metis* (*opaque-lifting*) *updVal-swapVal-eqBut swapVal-preserves-sWls updVal-preserves-sWls*)

The abstraction-renaming clause holds:

lemma *asIMOD-igAbs3*:
assumes *sWlsDisj SEM* **and** *igWls* (*asIMOD SEM*) *s X*
shows
igAbs (*asIMOD SEM*) *xs y* (*igSubst* (*asIMOD SEM*) *xs* (*igVar* (*asIMOD SEM*) *xs y*) *x X*) =
(λ*val*. *if sWlsVal SEM val*
then sAbs xs (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
then (*igSubst* (*asIMOD SEM*) *xs* (*igVar* (*asIMOD SEM*) *xs y*) *x X*) (*val* (*y* := *sX*)-*xs*)
else sDummy SEM s)
else undefined)
using *assms asIMOD-igVarIPresIGWls asIMOD-igSubstIPresIGWls*
unfolding *igVarIPresIGWls-def igSubstIPresIGWls-def*
by (*fastforce intro*!: *asIMOD-igAbs2*)

lemma *asIMOD-igAbsRen*:
sWlsDisj SEM ⟹ *igAbsRen* (*asIMOD SEM*)
unfolding *igAbsRen-def asIMOD-igFresh asIMOD-igSwap* **apply** *safe*
by (*simp add*: *asIMOD-igAbs2 asIMOD-igAbs3*)
(*auto intro*!: *ext simp*: *asIMOD-igAbs2 asIMOD-igAbs3 eqBut-def asIMOD-igSubst asIMOD-igVar*)

The associated model forms well-structured models of all 4 kinds:

lemma *asIMOD-wlsFSw*:
assumes *wlsSEM SEM*
shows *iwlsFSw* (*asIMOD SEM*)
using *assms* **unfolding** *wlsSEM-def iwlsFSw-def*
using *assms asIMOD-igWlsAllDisj asIMOD-igWlsAbsIsInBar asIMOD-igConsIPresIGWls asIMOD-igSwapAllIPresIGWlsAll asIMOD-igFreshCls asIMOD-igSwapCls asIMOD-igAbsCongS*
by *auto*

lemma *asIMOD-wlsFSb*:
assumes *wlsSEM SEM*
shows *iwlsFSb* (*asIMOD SEM*)
using *assms* **unfolding** *wlsSEM-def iwlsFSb-def*
using *assms asIMOD-igWlsAllDisj asIMOD-igWlsAbsIsInBar*

asIMOD-igConsIPresIGWls[*of SEM*] *asIMOD-igSubstAllIPresIGWlsAll*
asIMOD-igFreshCls *asIMOD-igSubstCls* *asIMOD-igAbsRen*
by *auto*

lemma *asIMOD-wlsFSwSb*: *wlsSEM SEM* $\Longrightarrow$ *iwlsFSwSb* (*asIMOD SEM*)
unfolding *iwlsFSwSb-def*
using *asIMOD-wlsFSw asIMOD-igSubstAllIPresIGWlsAll asIMOD-igSubstCls* **by** *auto*

lemma *asIMOD-wlsFSbSw*: *wlsSEM SEM* $\Longrightarrow$ *iwlsFSbSw* (*asIMOD SEM*)
unfolding *iwlsFSbSw-def*
using *asIMOD-wlsFSb asIMOD-igSwapAllIPresIGWlsAll asIMOD-igSwapCls* **by** *auto*

9.4 The semantic interpretation

The well-definedness of the semantic interpretation, as well as its associated substitution lemma and non-dependence of fresh variables, are the end products of this theory.

Note that in order to establish these results either fresh-subst-swap or fresh-swap-subst aligebras would do the job, and, moreover, if we did not care about swapping, fresh-subst aligebras would do the job. Therefore, our exhaustive study of the model from previous section had a deigree of redundancy w.r.t. to our main igoal – we pursued it however in order to better illustrate the rich structure laying under the apparent paucity of the notion of a semantic domain. Next, we choose to employ fresh-subst-swap aligebras to establish the required results. (Recall however that either aligebraic route we take, the initial morphism turns out to be the same function.)

definition *semInt* **where** *semInt SEM* $\equiv$ *iter* (*asIMOD SEM*)

definition *semIntAbs* **where** *semIntAbs SEM* $\equiv$ *iterAbs* (*asIMOD SEM*)

lemma *semIntAll-termFSwSbImorph*:
wlsSEM SEM $\Longrightarrow$
termFSwSbImorph (*semInt SEM*) (*semIntAbs SEM*) (*asIMOD SEM*)
unfolding *semInt-def semInt-def semIntAbs-def*
using *asIMOD-wlsFSbSw iwlsFSbSw-iterAll-termFSwSbImorph* **by** *auto*

lemma *semInt-prWls*:
wlsSEM SEM $\Longrightarrow$ *prWls* (*semInt SEM*) *SEM*
unfolding *prWls-def* **using** *semIntAll-termFSwSbImorph*
unfolding *termFSwSbImorph-def termFSwImorph-def ipresWlsAll-def ipresWls-def asIMOD-igWls* **by** *auto*

lemma *semIntAbs-prWlsAbs*:
wlsSEM SEM $\Longrightarrow$ *prWlsAbs* (*semIntAbs SEM*) *SEM*
unfolding *prWlsAbs-def* **using** *semIntAll-termFSwSbImorph*

unfolding *termFSwSbImorph-def termFSwImorph-def ipresWlsAll-def ipresWlsAbs-def asIMOD-igWlsAbs* **by** *blast*

lemma *semIntAll-prWlsAll*:
wlsSEM SEM ⟹ *prWlsAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
unfolding *prWlsAll-def* **by**(*simp add*: *semInt-prWls semIntAbs-prWlsAbs*)

lemma *semInt-prVar*:
wlsSEM SEM ⟹ *prVar* (*semInt SEM*) *SEM*
using *semIntAll-termFSwSbImorph*
unfolding *prVar-def termFSwSbImorph-def termFSwImorph-def ipresCons-def ipresVar-def asIMOD-igVar*
by *fastforce*

lemma *semIntAll-prAbs*:
fixes *SEM* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'sTerm*)*semDom*
assumes *wlsSEM SEM*
shows *prAbs* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
proof−
  {**fix** *xs s x X* **and** *val* :: (*'varSort*,*'var*,*'sTerm*)*val*
   assume *xs-s*: *isInBar* (*xs*,*s*) **and** *X*: *wls s X*
   and *val*: *sWlsVal SEM val*
   let *?L* = *semIntAbs SEM* (*Abs xs x X*)
   let *?R* = λ *val*. *sAbs xs* (λ*sX*. *if sWls SEM* (*asSort xs*) *sX*
                              then semInt SEM X (*val* (*x* := *sX*)-*xs*)
                              else sDummy SEM s)
   have *?L* = *igAbs* (*asIMOD SEM*) *xs x* (*semInt SEM X*)
   using *xs-s X assms semIntAll-termFSwSbImorph*[*of SEM*]
  unfolding *termFSwSbImorph-def termFSwImorph-def ipresCons-def ipresAbs-def*
by *auto*
   moreover
   {**have** *prWls* (*semInt SEM*) *SEM* **using** *assms semInt-prWls* **by** *auto*
    hence *1*: *sWls SEM s* (*semInt SEM X val*)
    using *val X* **unfolding** *prWls-def* **by** *simp*
    hence (*SOME s*. *sWls SEM s* (*semInt SEM X val*)) = *s*
    using *1 assms* **unfolding** *wlsSEM-def sWlsDisj-def* **by** *auto*
    hence *igAbs* (*asIMOD SEM*) *xs x* (*semInt SEM X*) *val* = *?R val*
    unfolding *asIMOD-igAbs* **using** *val* **by** *fastforce*
   }
   ultimately have *?L val* = *?R val* **by** *simp*
  }
  thus *?thesis* **unfolding** *prAbs-def* **by** *auto*
qed

lemma *semIntAll-prOp*:
assumes *wlsSEM SEM*
shows *prOp* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prOp-def termFSwSbImorph-def termFSwImorph-def ipresCons-def ipresOp-def*

asIMOD-igOp lift-comp comp-def **by** *fastforce*

lemma *semIntAll-prCons*:
assumes *wlsSEM SEM*
shows *prCons* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms* **unfolding** *prCons-def* **by**(*simp add*: *semInt-prVar semIntAll-prAbs semIntAll-prOp*)

lemma *semInt-prFresh*:
assumes *wlsSEM SEM*
shows *prFresh* (*semInt SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prFresh-def termFSwSbImorph-def termFSwImorph-def ipresFreshAll-def ipresFresh-def*
asIMOD-igFresh **by** *fastforce*

lemma *semIntAbs-prFreshAbs*:
assumes *wlsSEM SEM*
shows *prFreshAbs* (*semIntAbs SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prFreshAbs-def termFSwSbImorph-def termFSwImorph-def ipresFreshAll-def ipresFreshAbs-def*
asIMOD-igFreshAbs **by** *fastforce*

lemma *semIntAll-prFreshAll*:
assumes *wlsSEM SEM*
shows *prFreshAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms* **unfolding** *prFreshAll-def* **by**(*simp add*: *semInt-prFresh semIntAbs-prFreshAbs*)

lemma *semInt-prSwap*:
assumes *wlsSEM SEM*
shows *prSwap* (*semInt SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prSwap-def termFSwSbImorph-def termFSwImorph-def ipresSwapAll-def ipresSwap-def*
asIMOD-igSwap **by** *fastforce*

lemma *semIntAbs-prSwapAbs*:
assumes *wlsSEM SEM*
shows *prSwapAbs* (*semIntAbs SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prSwapAbs-def termFSwSbImorph-def termFSwImorph-def ipresSwapAll-def ipresSwapAbs-def*
asIMOD-igSwapAbs **by** *fastforce*

lemma *semIntAll-prSwapAll*:
assumes *wlsSEM SEM*
shows *prSwapAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms* **unfolding** *prSwapAll-def* **by**(*simp add*: *semInt-prSwap semIntAbs-prSwapAbs*)

lemma *semInt-prSubst*:
assumes *wlsSEM SEM*
shows *prSubst* (*semInt SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prSubst-def termFSwSbImorph-def termFSwImorph-def ipresSubstAll-def ipresSubst-def*
asIMOD-igSubst **by** *fastforce*

lemma *semIntAbs-prSubstAbs*:
assumes *wlsSEM SEM*
shows *prSubstAbs* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms semIntAll-termFSwSbImorph*
unfolding *prSubstAbs-def termFSwSbImorph-def termFSwImorph-def ipresSubstAll-def ipresSubstAbs-def*
asIMOD-igSubstAbs **by** *fastforce*

lemma *semIntAll-prSubstAll*:
assumes *wlsSEM SEM*
shows *prSubstAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms* **unfolding** *prSubstAll-def* **by**(*simp add*: *semInt-prSubst semIntAbs-prSubstAbs*)

theorem *semIntAll-compInt*:
assumes *wlsSEM SEM*
shows *compInt* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
using *assms* **unfolding** *compInt-def*
by(*simp add*: *semIntAll-prWlsAll semIntAll-prCons semIntAll-prFreshAll semIntAll-prSwapAll semIntAll-prSubstAll*)

lemmas *semDom-simps* = *updVal-simps swapVal-simps*

end

end

10 General Recursion

theory *Recursion* **imports** *Iteration*
begin

The initiality theorems from the previous section support iteration principles. Next we extend the results to general recursion. The difference between general recursion and iteration is that the former also considers the (source) “items" (terms and abstractions), and not only the (target) generalized items, appear in the recursive clauses.

(Here is an example illustrating the above difference for the standard case of natural numbers:
- Given a number n, the operator “add-n" can be defined by iteration:

— “add-n 0 = n",
— “add-n (Suc m) = Suc (add-n m)".

Notice that, in right-hand side of the recursive clause, “m" is not used “directly", but only via “add-n" – this makes the definition iterative. By contrast, the following definition of predecessor is trivial form of recursion (namely, case analysis), but is *not* iteration:
— “pred 0 = 0",
— “pred (Suc n) = n".)

We achieve our desired extension by augmenting the notion of model and then essentially inferring recursion (as customary) from [iteration having as target the product between the term model and the original model].

As a matter of notation: remember we are using for generalized items the same meta-variables as for “items" (terms and abstractions). But now the model operators will take both items and generalized items. We shall prime the meta-variables for items (as in X', A', etc).

10.1 Raw models

record *('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs)model* =
gWls :: *'sort* ⇒ *'gTerm* ⇒ *bool*
gWlsAbs :: *'varSort* × *'sort* ⇒ *'gAbs* ⇒ *bool*

gVar :: *'varSort* ⇒ *'var* ⇒ *'gTerm*
gAbs ::
'varSort ⇒ *'var* ⇒
('index,'bindex,'varSort,'var,'opSym)term ⇒ *'gTerm* ⇒
'gAbs
gOp ::
'opSym ⇒
('index,('index,'bindex,'varSort,'var,'opSym)term)input ⇒ *('index,'gTerm)input*
⇒
('bindex,('index,'bindex,'varSort,'var,'opSym)abs)input ⇒ *('bindex,'gAbs)input*
⇒
'gTerm

gFresh ::
'varSort ⇒ *'var* ⇒ *('index,'bindex,'varSort,'var,'opSym)term* ⇒ *'gTerm* ⇒ *bool*
gFreshAbs ::
'varSort ⇒ *'var* ⇒ *('index,'bindex,'varSort,'var,'opSym)abs* ⇒ *'gAbs* ⇒ *bool*

gSwap ::
'varSort ⇒ *'var* ⇒ *'var* ⇒
('index,'bindex,'varSort,'var,'opSym)term ⇒ *'gTerm* ⇒
'gTerm
gSwapAbs ::
'varSort ⇒ *'var* ⇒ *'var* ⇒
('index,'bindex,'varSort,'var,'opSym)abs ⇒ *'gAbs* ⇒

'gAbs

gSubst ::
'varSort ⇒
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* ⇒ *'gTerm* ⇒
'var ⇒
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* ⇒ *'gTerm* ⇒
'gTerm
gSubstAbs ::
'varSort ⇒
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* ⇒ *'gTerm* ⇒
'var ⇒
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs* ⇒ *'gAbs* ⇒
'gAbs

10.2 Well-sorted models of various kinds

Lifting the model operations to inputs

definition *gFreshInp* **where**
gFreshInp MOD ys y inp' inp ≡ *liftAll2* (*gFresh MOD ys y*) *inp' inp*

definition *gFreshBinp* **where**
gFreshBinp MOD ys y binp' binp ≡ *liftAll2* (*gFreshAbs MOD ys y*) *binp' binp*

definition *gSwapInp* **where**
gSwapInp MOD zs z1 z2 inp' inp ≡ *lift2* (*gSwap MOD zs z1 z2*) *inp' inp*

definition *gSwapBinp* **where**
gSwapBinp MOD zs z1 z2 binp' binp ≡ *lift2* (*gSwapAbs MOD zs z1 z2*) *binp' binp*

definition *gSubstInp* **where**
gSubstInp MOD ys Y' Y y inp' inp ≡ *lift2* (*gSubst MOD ys Y' Y y*) *inp' inp*

definition *gSubstBinp* **where**
gSubstBinp MOD ys Y' Y y binp' binp ≡ *lift2* (*gSubstAbs MOD ys Y' Y y*) *binp' binp*

context *FixSyn*
begin

definition *gWlsInp* **where**
gWlsInp MOD delta inp ≡
wlsOpS delta ∧ *sameDom* (*arOf delta*) *inp* ∧ *liftAll2* (*gWls MOD*) (*arOf delta*) *inp*

lemmas *gWlsInp-defs* = *gWlsInp-def sameDom-def liftAll2-def*

definition *gWlsBinp* **where**

gWlsBinp MOD delta binp ≡
wlsOpS delta ∧ *sameDom* (*barOf delta*) *binp* ∧ *liftAll2* (*gWlsAbs MOD*) (*barOf delta*) *binp*

lemmas *gWlsBinp-defs* = *gWlsBinp-def sameDom-def liftAll2-def*

Basic properties of the lifted model operations

. for free inputs:

lemma *sameDom-swapInp-gSwapInp*[*simp*]:
assumes *wlsInp delta inp′* **and** *gWlsInp MOD delta inp*
shows *sameDom* (*swapInp zs z1 z2 inp′*) (*gSwapInp MOD zs z1 z2 inp′ inp*)
using *assms* **by**(*simp add*: *wlsInp-iff gWlsInp-def swapInp-def gSwapInp-def liftAll2-def lift-def lift2-def sameDom-def split*: *option.splits*)

lemma *sameDom-substInp-gSubstInp*[*simp*]:
assumes *wlsInp delta inp′* **and** *gWlsInp MOD delta inp*
shows *sameDom* (*substInp ys Y′ y inp′*) (*gSubstInp MOD ys Y′ Y y inp′ inp*)
using *assms* **by**(*simp add*: *wlsInp-iff gWlsInp-def substInp-def2 gSubstInp-def liftAll2-def lift-def lift2-def sameDom-def split*: *option.splits*)

. for bound inputs:

lemma *sameDom-swapBinp-gSwapBinp*[*simp*]:
assumes *wlsBinp delta binp′* **and** *gWlsBinp MOD delta binp*
shows *sameDom* (*swapBinp zs z1 z2 binp′*) (*gSwapBinp MOD zs z1 z2 binp′ binp*)
using *assms* **by**(*simp add*: *wlsBinp-iff gWlsBinp-def swapBinp-def gSwapBinp-def liftAll2-def lift-def lift2-def sameDom-def split*: *option.splits*)

lemma *sameDom-substBinp-gSubstBinp*[*simp*]:
assumes *wlsBinp delta binp′* **and** *gWlsBinp MOD delta binp*
shows *sameDom* (*substBinp ys Y′ y binp′*) (*gSubstBinp MOD ys Y′ Y y binp′ binp*)
using *assms* **by**(*simp add*: *wlsBinp-iff gWlsBinp-def substBinp-def2 gSubstBinp-def liftAll2-def lift-def lift2-def sameDom-def split*: *option.splits*)

lemmas *sameDom-gInput-simps* =
sameDom-swapInp-gSwapInp sameDom-substInp-gSubstInp
sameDom-swapBinp-gSwapBinp sameDom-substBinp-gSubstBinp

Domain disjointness:

definition *gWlsDisj* **where**
gWlsDisj MOD ≡ ∀ *s s′ X*. *gWls MOD s X* ∧ *gWls MOD s′ X* ⟶ *s* = *s′*

definition *gWlsAbsDisj* **where**
gWlsAbsDisj MOD ≡ ∀ *xs s xs′ s′ A*.
isInBar (*xs*,*s*) ∧ *isInBar* (*xs′*,*s′*) ∧
gWlsAbs MOD (*xs*,*s*) *A* ∧ *gWlsAbs MOD* (*xs′*,*s′*) *A*
⟶ *xs* = *xs′* ∧ *s* = *s′*

definition *gWlsAllDisj* **where**
gWlsAllDisj MOD ≡ *gWlsDisj MOD* ∧ *gWlsAbsDisj MOD*

lemmas *gWlsAllDisj-defs* =
gWlsAllDisj-def gWlsDisj-def gWlsAbsDisj-def

Abstraction domains inhabited only within bound arities:

definition *gWlsAbsIsInBar* **where**
gWlsAbsIsInBar MOD ≡ ∀ *us s A*. *gWlsAbs MOD* (*us*,*s*) *A* ⟶ *isInBar* (*us*,*s*)

Domain preservation by the operators

The constructs preserve the domains:

definition *gVarPresGWls* **where**
gVarPresGWls MOD ≡ ∀ *xs x*. *gWls MOD* (*asSort xs*) (*gVar MOD xs x*)

definition *gAbsPresGWls* **where**
gAbsPresGWls MOD ≡ ∀ *xs s x X′ X*.
isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
gWlsAbs MOD (*xs*,*s*) (*gAbs MOD xs x X′ X*)

definition *gOpPresGWls* **where**
gOpPresGWls MOD ≡ ∀ *delta inp′ inp binp′ binp*.
wlsInp delta inp′ ∧ *gWlsInp MOD delta inp* ∧ *wlsBinp delta binp′* ∧ *gWlsBinp MOD delta binp*
⟶ *gWls MOD* (*stOf delta*) (*gOp MOD delta inp′ inp binp′ binp*)

definition *gConsPresGWls* **where**
gConsPresGWls MOD ≡ *gVarPresGWls MOD* ∧ *gAbsPresGWls MOD* ∧ *gOpPresGWls MOD*

lemmas *gConsPresGWls-defs* = *gConsPresGWls-def*
gVarPresGWls-def gAbsPresGWls-def gOpPresGWls-def

“swap" preserves the domains:

definition *gSwapPresGWls* **where**
gSwapPresGWls MOD ≡ ∀ *zs z1 z2 s X′ X*.
wls s X′ ∧ *gWls MOD s X* ⟶
gWls MOD s (*gSwap MOD zs z1 z2 X′ X*)

definition *gSwapAbsPresGWlsAbs* **where**
gSwapAbsPresGWlsAbs MOD ≡ ∀ *zs z1 z2 us s A′ A*.
isInBar (*us*,*s*) ∧ *wlsAbs* (*us*,*s*) *A′* ∧ *gWlsAbs MOD* (*us*,*s*) *A* ⟶
gWlsAbs MOD (*us*,*s*) (*gSwapAbs MOD zs z1 z2 A′ A*)

definition *gSwapAllPresGWlsAll* **where**
gSwapAllPresGWlsAll MOD ≡ *gSwapPresGWls MOD* ∧ *gSwapAbsPresGWlsAbs MOD*

lemmas *gSwapAllPresGWlsAll-defs* =
gSwapAllPresGWlsAll-def gSwapPresGWls-def gSwapAbsPresGWlsAbs-def

"subst" preserves the domains:

definition *gSubstPresGWls* **where**
gSubstPresGWls MOD ≡ ∀ *ys Y′ Y y s X′ X*.
wls (*asSort ys*) *Y′* ∧ *gWls MOD* (*asSort ys*) *Y* ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
gWls MOD s (*gSubst MOD ys Y′ Y y X′ X*)

definition *gSubstAbsPresGWlsAbs* **where**
gSubstAbsPresGWlsAbs MOD ≡ ∀ *ys Y′ Y y us s A′ A*.
isInBar (*us*,*s*) ∧
wls (*asSort ys*) *Y′* ∧ *gWls MOD* (*asSort ys*) *Y* ∧ *wlsAbs* (*us*,*s*) *A′* ∧ *gWlsAbs MOD* (*us*,*s*) *A* ⟶
gWlsAbs MOD (*us*,*s*) (*gSubstAbs MOD ys Y′ Y y A′ A*)

definition *gSubstAllPresGWlsAll* **where**
gSubstAllPresGWlsAll MOD ≡ *gSubstPresGWls MOD* ∧ *gSubstAbsPresGWlsAbs MOD*

lemmas *gSubstAllPresGWlsAll-defs* =
gSubstAllPresGWlsAll-def gSubstPresGWls-def gSubstAbsPresGWlsAbs-def

Clauses for fresh:

definition *gFreshGVar* **where**
gFreshGVar MOD ≡ ∀ *ys y xs x*.
(*ys* ≠ *xs* ∨ *y* ≠ *x*) ⟶
gFresh MOD ys y (*Var xs x*) (*gVar MOD xs x*)

definition *gFreshGAbs1* **where**
gFreshGAbs1 MOD ≡ ∀ *ys y s X′ X*.
isInBar (*ys*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
gFreshAbs MOD ys y (*Abs ys y X′*) (*gAbs MOD ys y X′ X*)

definition *gFreshGAbs2* **where**
gFreshGAbs2 MOD ≡ ∀ *ys y xs x s X′ X*.
isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
fresh ys y X′ ∧ *gFresh MOD ys y X′ X* ⟶
gFreshAbs MOD ys y (*Abs xs x X′*) (*gAbs MOD xs x X′ X*)

definition *gFreshGOp* **where**
gFreshGOp MOD ≡ ∀ *ys y delta inp′ inp binp′ binp*.
wlsInp delta inp′ ∧ *gWlsInp MOD delta inp* ∧ *wlsBinp delta binp′* ∧ *gWlsBinp MOD delta binp* ⟶
freshInp ys y inp′ ∧ *gFreshInp MOD ys y inp′ inp* ∧
freshBinp ys y binp′ ∧ *gFreshBinp MOD ys y binp′ binp* ⟶
gFresh MOD ys y (*Op delta inp′ binp′*) (*gOp MOD delta inp′ inp binp′ binp*)

definition *gFreshCls* **where**
gFreshCls MOD ≡ *gFreshGVar MOD* ∧ *gFreshGAbs1 MOD* ∧ *gFreshGAbs2 MOD* ∧ *gFreshGOp MOD*

lemmas *gFreshCls-defs* = *gFreshCls-def*
gFreshGVar-def gFreshGAbs1-def gFreshGAbs2-def gFreshGOp-def

definition *gSwapGVar* **where**
gSwapGVar MOD ≡ ∀ *zs z1 z2 xs x*.
  gSwap MOD zs z1 z2 (*Var xs x*) (*gVar MOD xs x*) =
  gVar MOD xs (*x* @*xs*[*z1* ∧ *z2*]-*zs*)

definition *gSwapGAbs* **where**
gSwapGAbs MOD ≡ ∀ *zs z1 z2 xs x s X′ X*.
  isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
  gSwapAbs MOD zs z1 z2 (*Abs xs x X′*) (*gAbs MOD xs x X′ X*) =
  gAbs MOD xs (*x* @*xs*[*z1* ∧ *z2*]-*zs*) (*X′* #[*z1* ∧ *z2*]-*zs*) (*gSwap MOD zs z1 z2 X′ X*)

definition *gSwapGOp* **where**
gSwapGOp MOD ≡ ∀ *zs z1 z2 delta inp′ inp binp′ binp*.
  wlsInp delta inp′ ∧ *gWlsInp MOD delta inp* ∧ *wlsBinp delta binp′* ∧ *gWlsBinp MOD delta binp* ⟶
  gSwap MOD zs z1 z2 (*Op delta inp′ binp′*) (*gOp MOD delta inp′ inp binp′ binp*) =
  gOp MOD delta
    (*inp′* %[*z1* ∧ *z2*]-*zs*) (*gSwapInp MOD zs z1 z2 inp′ inp*)
    (*binp′* %%[*z1* ∧ *z2*]-*zs*) (*gSwapBinp MOD zs z1 z2 binp′ binp*)

definition *gSwapCls* **where**
gSwapCls MOD ≡ *gSwapGVar MOD* ∧ *gSwapGAbs MOD* ∧ *gSwapGOp MOD*

lemmas *gSwapCls-defs* = *gSwapCls-def*
gSwapGVar-def gSwapGAbs-def gSwapGOp-def

definition *gSubstGVar1* **where**
gSubstGVar1 MOD ≡ ∀ *ys y Y′ Y xs x*.
  wls (*asSort ys*) *Y′* ∧ *gWls MOD* (*asSort ys*) *Y* ⟶
  (*ys* ≠ *xs* ∨ *y* ≠ *x*) ⟶
  gSubst MOD ys Y′ Y y (*Var xs x*) (*gVar MOD xs x*) =
  gVar MOD xs x

definition *gSubstGVar2* **where**
gSubstGVar2 MOD ≡ ∀ *ys y Y′ Y*.
  wls (*asSort ys*) *Y′* ∧ *gWls MOD* (*asSort ys*) *Y* ⟶

$gSubst\ MOD\ ys\ Y'\ Y\ y\ (Var\ ys\ y)\ (gVar\ MOD\ ys\ y) = Y$

definition *gSubstGAbs* **where**
$gSubstGAbs\ MOD \equiv \forall\ ys\ y\ Y'\ Y\ xs\ x\ s\ X'\ X.$
$isInBar\ (xs,s) \wedge$
$wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \wedge$
$wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$
$(xs \neq ys \vee x \neq y) \wedge fresh\ xs\ x\ Y' \wedge gFresh\ MOD\ xs\ x\ Y'\ Y \longrightarrow$
$gSubstAbs\ MOD\ ys\ Y'\ Y\ y\ (Abs\ xs\ x\ X')\ (gAbs\ MOD\ xs\ x\ X'\ X) =$
$gAbs\ MOD\ xs\ x\ (X'\ \#[Y'\ /\ y]\text{-}ys)\ (gSubst\ MOD\ ys\ Y'\ Y\ y\ X'\ X)$

definition *gSubstGOp* **where**
$gSubstGOp\ MOD \equiv \forall\ ys\ y\ Y'\ Y\ delta\ inp'\ inp\ binp'\ binp.$
$wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \wedge$
$wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \wedge$
$wlsBinp\ delta\ binp' \wedge gWlsBinp\ MOD\ delta\ binp \longrightarrow$
$gSubst\ MOD\ ys\ Y'\ Y\ y\ (Op\ delta\ inp'\ binp')\ (gOp\ MOD\ delta\ inp'\ inp\ binp'\ binp) =$
$gOp\ MOD\ delta$
$(inp'\ \%[Y'\ /\ y]\text{-}ys)\ (gSubstInp\ MOD\ ys\ Y'\ Y\ y\ inp'\ inp)$
$(binp'\ \%\%[Y'\ /\ y]\text{-}ys)\ (gSubstBinp\ MOD\ ys\ Y'\ Y\ y\ binp'\ binp)$

definition *gSubstCls* **where**
$gSubstCls\ MOD \equiv gSubstGVar1\ MOD \wedge gSubstGVar2\ MOD \wedge gSubstGAbs\ MOD \wedge gSubstGOp\ MOD$

lemmas *gSubstCls-defs = gSubstCls-def gSubstGVar1-def gSubstGVar2-def gSubstGAbs-def gSubstGOp-def*

definition *gAbsCongS* **where**
$gAbsCongS\ MOD \equiv \forall\ xs\ x\ x2\ y\ s\ X'\ X\ X2'\ X2.$
$isInBar\ (xs,s) \wedge$
$wls\ s\ X' \wedge gWls\ MOD\ s\ X \wedge$
$wls\ s\ X2' \wedge gWls\ MOD\ s\ X2 \longrightarrow$
$fresh\ xs\ y\ X' \wedge gFresh\ MOD\ xs\ y\ X'\ X \wedge$
$fresh\ xs\ y\ X2' \wedge gFresh\ MOD\ xs\ y\ X2'\ X2 \wedge$
$(X'\ \#[y \wedge x]\text{-}xs) = (X2'\ \#[y \wedge x2]\text{-}xs) \longrightarrow$
$gSwap\ MOD\ xs\ y\ x\ X'\ X = gSwap\ MOD\ xs\ y\ x2\ X2'\ X2 \longrightarrow$
$gAbs\ MOD\ xs\ x\ X'\ X = gAbs\ MOD\ xs\ x2\ X2'\ X2$

definition *gAbsRen* **where**

gAbsRen MOD ≡ ∀ *xs y x s X′ X*.
isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
fresh xs y X′ ∧ *gFresh MOD xs y X′ X* ⟶
gAbs MOD xs y (*X′* #[*y* // *x*]-*xs*) (*gSubst MOD xs* (*Var xs y*) (*gVar MOD xs y*) *x X′ X*) =
gAbs MOD xs x X′ X

Well-sorted fresh-swap models:

definition *wlsFSw* **where**
wlsFSw MOD ≡ *gWlsAllDisj MOD* ∧ *gWlsAbsIsInBar MOD* ∧
gConsPresGWls MOD ∧ *gSwapAllPresGWlsAll MOD* ∧
gFreshCls MOD ∧ *gSwapCls MOD* ∧ *gAbsCongS MOD*

lemmas *wlsFSw-defs1* = *wlsFSw-def*
gWlsAllDisj-def gWlsAbsIsInBar-def
gConsPresGWls-def gSwapAllPresGWlsAll-def
gFreshCls-def gSwapCls-def gAbsCongS-def

lemmas *wlsFSw-defs* = *wlsFSw-def*
gWlsAllDisj-defs gWlsAbsIsInBar-def
gConsPresGWls-defs gSwapAllPresGWlsAll-defs
gFreshCls-defs gSwapCls-defs gAbsCongS-def

Well-sorted fresh-subst models:

definition *wlsFSb* **where**
wlsFSb MOD ≡ *gWlsAllDisj MOD* ∧ *gWlsAbsIsInBar MOD* ∧
gConsPresGWls MOD ∧ *gSubstAllPresGWlsAll MOD* ∧
gFreshCls MOD ∧ *gSubstCls MOD* ∧ *gAbsRen MOD*

lemmas *wlsFSb-defs1* = *wlsFSb-def*
gWlsAllDisj-def gWlsAbsIsInBar-def
gConsPresGWls-def gSubstAllPresGWlsAll-def
gFreshCls-def gSubstCls-def gAbsRen-def

lemmas *wlsFSb-defs* = *wlsFSb-def*
gWlsAllDisj-defs gWlsAbsIsInBar-def
gConsPresGWls-defs gSubstAllPresGWlsAll-defs
gFreshCls-defs gSubstCls-defs gAbsRen-def

Well-sorted fresh-swap-subst-models

definition *wlsFSwSb* **where**
wlsFSwSb MOD ≡ *wlsFSw MOD* ∧ *gSubstAllPresGWlsAll MOD* ∧ *gSubstCls MOD*

lemmas *wlsFSwSb-defs1* = *wlsFSwSb-def*
wlsFSw-def gSubstAllPresGWlsAll-def gSubstCls-def

lemmas *wlsFSwSb-defs* = *wlsFSwSb-def*
wlsFSw-def gSubstAllPresGWlsAll-defs gSubstCls-defs

Well-sorted fresh-subst-swap-models

definition *wlsFSbSw* **where**
wlsFSbSw MOD ≡ wlsFSb MOD ∧ gSwapAllPresGWlsAll MOD ∧ gSwapCls MOD

lemmas *wlsFSbSw-defs1 = wlsFSbSw-def*
wlsFSw-def gSwapAllPresGWlsAll-def gSwapCls-def

lemmas *wlsFSbSw-defs = wlsFSbSw-def*
wlsFSw-def gSwapAllPresGWlsAll-defs gSwapCls-defs

Extension of domain preservation (by swap and subst) to inputs:

First for free inputs:

definition *gSwapInpPresGWlsInp* **where**
gSwapInpPresGWlsInp MOD ≡ ∀ zs z1 z2 delta inp′ inp.
wlsInp delta inp′ ∧ gWlsInp MOD delta inp ⟶
gWlsInp MOD delta (gSwapInp MOD zs z1 z2 inp′ inp)

definition *gSubstInpPresGWlsInp* **where**
gSubstInpPresGWlsInp MOD ≡ ∀ ys y Y′ Y delta inp′ inp.
wls (asSort ys) Y′ ∧ gWls MOD (asSort ys) Y ∧
wlsInp delta inp′ ∧ gWlsInp MOD delta inp ⟶
gWlsInp MOD delta (gSubstInp MOD ys Y′ Y y inp′ inp)

lemma *imp-gSwapInpPresGWlsInp*:
gSwapPresGWls MOD ⟹ gSwapInpPresGWlsInp MOD
by (*auto simp*: *lift2-def liftAll2-def sameDom-def wlsInp-iff gWlsInp-def*
gSwapPresGWls-def gSwapInpPresGWlsInp-def gSwapInp-def
split: *option.splits*)

lemma *imp-gSubstInpPresGWlsInp*:
gSubstPresGWls MOD ⟹ gSubstInpPresGWlsInp MOD
by (*auto simp*: *lift2-def liftAll2-def sameDom-def wlsInp-iff gWlsInp-def*
gSubstPresGWls-def gSubstInpPresGWlsInp-def gSubstInp-def
split: *option.splits*)

Then for bound inputs:

definition *gSwapBinpPresGWlsBinp* **where**
gSwapBinpPresGWlsBinp MOD ≡ ∀ zs z1 z2 delta binp′ binp.
wlsBinp delta binp′ ∧ gWlsBinp MOD delta binp ⟶
gWlsBinp MOD delta (gSwapBinp MOD zs z1 z2 binp′ binp)

definition *gSubstBinpPresGWlsBinp* **where**
gSubstBinpPresGWlsBinp MOD ≡ ∀ ys y Y′ Y delta binp′ binp.
wls (asSort ys) Y′ ∧ gWls MOD (asSort ys) Y ∧
wlsBinp delta binp′ ∧ gWlsBinp MOD delta binp ⟶
gWlsBinp MOD delta (gSubstBinp MOD ys Y′ Y y binp′ binp)

lemma *imp-gSwapBinpPresGWlsBinp*:

gSwapAbsPresGWlsAbs MOD ⟹ *gSwapBinpPresGWlsBinp MOD*
by (*auto simp*: *lift2-def liftAll2-def sameDom-def wlsBinp-iff gWlsBinp-def gSwapAbsPresGWlsAbs-def gSwapBinpPresGWlsBinp-def gSwapBinp-def split*: *option.splits*)

lemma *imp-gSubstBinpPresGWlsBinp*:
gSubstAbsPresGWlsAbs MOD ⟹ *gSubstBinpPresGWlsBinp MOD*
by (*auto simp*: *lift2-def liftAll2-def sameDom-def wlsBinp-iff gWlsBinp-def gSubstAbsPresGWlsAbs-def gSubstBinpPresGWlsBinp-def gSubstBinp-def split*: *option.splits*)

10.3 Model morphisms from the term model

definition *presWls* **where**
presWls h MOD ≡ ∀ *s X*. *wls s X* ⟶ *gWls MOD s* (*h X*)

definition *presWlsAbs* **where**
presWlsAbs hA MOD ≡ ∀ *us s A*. *wlsAbs* (*us*,*s*) *A* ⟶ *gWlsAbs MOD* (*us*,*s*) (*hA A*)

definition *presWlsAll* **where**
presWlsAll h hA MOD ≡ *presWls h MOD* ∧ *presWlsAbs hA MOD*

lemmas *presWlsAll-defs* = *presWlsAll-def presWls-def presWlsAbs-def*

definition *presVar* **where**
presVar h MOD ≡ ∀ *xs x*. *h* (*Var xs x*) = *gVar MOD xs x*

definition *presAbs* **where**
presAbs h hA MOD ≡ ∀ *xs x s X*.
  isInBar (*xs*,*s*) ∧ *wls s X* ⟶
  hA (*Abs xs x X*) = *gAbs MOD xs x X* (*h X*)

definition *presOp* **where**
presOp h hA MOD ≡ ∀ *delta inp binp*.
   wlsInp delta inp ∧ *wlsBinp delta binp* ⟶
   h (*Op delta inp binp*) =
   gOp MOD delta inp (*lift h inp*) *binp* (*lift hA binp*)

definition *presCons* **where**
presCons h hA MOD ≡ *presVar h MOD* ∧ *presAbs h hA MOD* ∧ *presOp h hA MOD*

lemmas *presCons-defs* = *presCons-def presVar-def presAbs-def presOp-def*

definition *presFresh* **where**
presFresh h MOD ≡ ∀ *ys y s X*.
   wls s X ⟶

fresh ys y X ⟶ *gFresh MOD ys y X* (*h X*)

definition *presFreshAbs* **where**
presFreshAbs hA MOD ≡ ∀ *ys y us s A*.
wlsAbs (*us*,*s*) *A* ⟶
freshAbs ys y A ⟶ *gFreshAbs MOD ys y A* (*hA A*)

definition *presFreshAll* **where**
presFreshAll h hA MOD ≡ *presFresh h MOD* ∧ *presFreshAbs hA MOD*

lemmas *presFreshAll-defs* = *presFreshAll-def*
presFresh-def presFreshAbs-def

definition *presSwap* **where**
presSwap h MOD ≡ ∀ *zs z1 z2 s X*.
wls s X ⟶
h (*X* #[*z1* ∧ *z2*]-*zs*) = *gSwap MOD zs z1 z2 X* (*h X*)

definition *presSwapAbs* **where**
presSwapAbs hA MOD ≡ ∀ *zs z1 z2 us s A*.
wlsAbs (*us*,*s*) *A* ⟶
hA (*A* $[*z1* ∧ *z2*]-*zs*) = *gSwapAbs MOD zs z1 z2 A* (*hA A*)

definition *presSwapAll* **where**
presSwapAll h hA MOD ≡ *presSwap h MOD* ∧ *presSwapAbs hA MOD*

lemmas *presSwapAll-defs* = *presSwapAll-def*
presSwap-def presSwapAbs-def

definition *presSubst* **where**
presSubst h MOD ≡ ∀ *ys Y y s X*.
wls (*asSort ys*) *Y* ∧ *wls s X* ⟶
h (*subst ys Y y X*) = *gSubst MOD ys Y* (*h Y*) *y X* (*h X*)

definition *presSubstAbs* **where**
presSubstAbs h hA MOD ≡ ∀ *ys Y y us s A*.
wls (*asSort ys*) *Y* ∧ *wlsAbs* (*us*,*s*) *A* ⟶
hA (*A* $[*Y* / *y*]-*ys*) = *gSubstAbs MOD ys Y* (*h Y*) *y A* (*hA A*)

definition *presSubstAll* **where**
presSubstAll h hA MOD ≡ *presSubst h MOD* ∧ *presSubstAbs h hA MOD*

lemmas *presSubstAll-defs* = *presSubstAll-def*
presSubst-def presSubstAbs-def

definition *termFSwMorph* **where**
termFSwMorph h hA MOD ≡ *presWlsAll h hA MOD* ∧ *presCons h hA MOD* ∧
presFreshAll h hA MOD ∧ *presSwapAll h hA MOD*

lemmas *termFSwMorph-defs1* = *termFSwMorph-def*
presWlsAll-def presCons-def presFreshAll-def presSwapAll-def

lemmas *termFSwMorph-defs* = *termFSwMorph-def*
presWlsAll-defs presCons-defs presFreshAll-defs presSwapAll-defs

definition *termFSbMorph* **where**
termFSbMorph h hA MOD ≡ *presWlsAll h hA MOD* ∧ *presCons h hA MOD* ∧
presFreshAll h hA MOD ∧ *presSubstAll h hA MOD*

lemmas *termFSbMorph-defs1* = *termFSbMorph-def*
presWlsAll-def presCons-def presFreshAll-def presSubstAll-def

lemmas *termFSbMorph-defs* = *termFSbMorph-def*
presWlsAll-defs presCons-defs presFreshAll-defs presSubstAll-defs

definition *termFSwSbMorph* **where**
termFSwSbMorph h hA MOD ≡ *termFSwMorph h hA MOD* ∧ *presSubstAll h hA MOD*

lemmas *termFSwSbMorph-defs1* = *termFSwSbMorph-def*
termFSwMorph-def presSubstAll-def

lemmas *termFSwSbMorph-defs* = *termFSwSbMorph-def*
termFSwMorph-defs presSubstAll-defs

Extension of domain preservation (by the morphisms) to inputs

. for free inputs:

lemma *presWls-wlsInp*:
wlsInp delta inp ⟹ *presWls h MOD* ⟹ *gWlsInp MOD delta* (*lift h inp*)
by(*auto simp*: *wlsInp-iff gWlsInp-def lift-def liftAll2-def sameDom-def*
presWls-def split: *option.splits*)

. for bound inputs:

lemma *presWls-wlsBinp*:
wlsBinp delta binp ⟹ *presWlsAbs hA MOD* ⟹ *gWlsBinp MOD delta* (*lift hA binp*)
by(*auto simp*: *wlsBinp-iff gWlsBinp-def lift-def liftAll2-def sameDom-def*
presWlsAbs-def split: *option.splits*)

10.4 From models to iterative models

The transition map:

definition *fromMOD* ::
('*index*,'*bindex*,'*varSort*,'*sort*,'*opSym*,'*var*,'*gTerm*,'*gAbs*) *model*
⇒
('*index*,'*bindex*,'*varSort*,'*sort*,'*opSym*,'*var*,

$('index,'bindex,'varSort,'var,'opSym)term \times 'gTerm,$
$('index,'bindex,'varSort,'var,'opSym)abs \times 'gAbs)$ *Iteration.model*
where
fromMOD MOD ≡
⦇
$igWls = \lambda s\ X'X.\ wls\ s\ (fst\ X'X) \wedge gWls\ MOD\ s\ (snd\ X'X),$
$igWlsAbs = \lambda us\text{-}s\ A'A.\ wlsAbs\ us\text{-}s\ (fst\ A'A) \wedge gWlsAbs\ MOD\ us\text{-}s\ (snd\ A'A),$

$igVar = \lambda xs\ x.\ (Var\ xs\ x,\ gVar\ MOD\ xs\ x),$
$igAbs = \lambda xs\ x\ X'X.\ (Abs\ xs\ x\ (fst\ X'X),\ gAbs\ MOD\ xs\ x\ (fst\ X'X)\ (snd\ X'X)),$
$igOp =$
$\lambda delta\ iinp\ biinp.$
$(Op\ delta\ (lift\ fst\ iinp)\ (lift\ fst\ biinp),$
$gOp\ MOD\ delta$
$(lift\ fst\ iinp)\ (lift\ snd\ iinp)$
$(lift\ fst\ biinp)\ (lift\ snd\ biinp)),$

$igFresh =$
$\lambda ys\ y\ X'X.\ fresh\ ys\ y\ (fst\ X'X) \wedge gFresh\ MOD\ ys\ y\ (fst\ X'X)\ (snd\ X'X),$
$igFreshAbs =$
$\lambda ys\ y\ A'A.\ freshAbs\ ys\ y\ (fst\ A'A) \wedge gFreshAbs\ MOD\ ys\ y\ (fst\ A'A)\ (snd\ A'A),$

$igSwap =$
$\lambda zs\ z1\ z2\ X'X.\ ((fst\ X'X)\ \#[z1 \wedge z2]\text{-}zs,\ gSwap\ MOD\ zs\ z1\ z2\ (fst\ X'X)\ (snd\ X'X)),$
$igSwapAbs =$
$\lambda zs\ z1\ z2\ A'A.\ ((fst\ A'A)\ \$[z1 \wedge z2]\text{-}zs,\ gSwapAbs\ MOD\ zs\ z1\ z2\ (fst\ A'A)\ (snd\ A'A)),$

$igSubst =$
$\lambda ys\ Y'Y\ y\ X'X.$
$((fst\ X'X)\ \#[(fst\ Y'Y)\ /\ y]\text{-}ys,$
$gSubst\ MOD\ ys\ (fst\ Y'Y)\ (snd\ Y'Y)\ y\ (fst\ X'X)\ (snd\ X'X)),$
$igSubstAbs =$
$\lambda ys\ Y'Y\ y\ A'A.$
$((fst\ A'A)\ \$[(fst\ Y'Y)\ /\ y]\text{-}ys,$
$gSubstAbs\ MOD\ ys\ (fst\ Y'Y)\ (snd\ Y'Y)\ y\ (fst\ A'A)\ (snd\ A'A))$
⦈

Basic simplification rules:

lemma *fromMOD-basic-simps*[*simp*]:
$igWls\ (fromMOD\ MOD)\ s\ X'X =$
$(wls\ s\ (fst\ X'X) \wedge gWls\ MOD\ s\ (snd\ X'X))$

$igWlsAbs\ (fromMOD\ MOD)\ us\text{-}s\ A'A =$
$(wlsAbs\ us\text{-}s\ (fst\ A'A) \wedge gWlsAbs\ MOD\ us\text{-}s\ (snd\ A'A))$

$igVar\ (fromMOD\ MOD)\ xs\ x = (Var\ xs\ x,\ gVar\ MOD\ xs\ x)$

igAbs (*fromMOD MOD*) *xs x X′X* = (*Abs xs x* (*fst X′X*), *gAbs MOD xs x* (*fst X′X*) (*snd X′X*))

igOp (*fromMOD MOD*) *delta iinp biinp* =
(*Op delta* (*lift fst iinp*) (*lift fst biinp*),
gOp MOD delta
(*lift fst iinp*) (*lift snd iinp*)
(*lift fst biinp*) (*lift snd biinp*))

igFresh (*fromMOD MOD*) *ys y X′X* =
(*fresh ys y* (*fst X′X*) ∧ *gFresh MOD ys y* (*fst X′X*) (*snd X′X*))

igFreshAbs (*fromMOD MOD*) *ys y A′A* =
(*freshAbs ys y* (*fst A′A*) ∧ *gFreshAbs MOD ys y* (*fst A′A*) (*snd A′A*))

igSwap (*fromMOD MOD*) *zs z1 z2 X′X* =
((*fst X′X*) #[*z1* ∧ *z2*]-*zs*, *gSwap MOD zs z1 z2* (*fst X′X*) (*snd X′X*))

igSwapAbs (*fromMOD MOD*) *zs z1 z2 A′A* =
((*fst A′A*) $[*z1* ∧ *z2*]-*zs*, *gSwapAbs MOD zs z1 z2* (*fst A′A*) (*snd A′A*))

igSubst (*fromMOD MOD*) *ys Y′Y y X′X* =
((*fst X′X*) #[(*fst Y′Y*) / *y*]-*ys*,
gSubst MOD ys (*fst Y′Y*) (*snd Y′Y*) *y* (*fst X′X*) (*snd X′X*))

igSubstAbs (*fromMOD MOD*) *ys Y′Y y A′A* =
((*fst A′A*) $[(*fst Y′Y*) / *y*]-*ys*,
gSubstAbs MOD ys (*fst Y′Y*) (*snd Y′Y*) *y* (*fst A′A*) (*snd A′A*))
unfolding *fromMOD-def* **by** *auto*

Simps for inputs

. for free inputs:

lemma *igWlsInp-fromMOD*[*simp*]:
igWlsInp (*fromMOD MOD*) *delta iinp* ⟷
wlsInp delta (*lift fst iinp*) ∧ *gWlsInp MOD delta* (*lift snd iinp*)
apply (*intro iffI*)
subgoal apply(*simp add*: *liftAll2-def lift-def sameDom-def*
igWlsInp-def wlsInp-iff gWlsInp-def split: *option.splits*) **.**
subgoal
unfolding *liftAll2-def lift-def sameDom-def*
igWlsInp-def wlsInp-iff gWlsInp-def
by *simp* (*metis* (*no-types*, *lifting*) *eq-snd-iff fstI option.case-eq-if*
option.distinct(*1*) *option.simps*(*5*)) **.**

lemma *igFreshInp-fromMOD*[*simp*]:
igFreshInp (*fromMOD MOD*) *ys y iinp* ⟷
freshInp ys y (*lift fst iinp*) ∧ *gFreshInp MOD ys y* (*lift fst iinp*) (*lift snd iinp*)
by (*auto simp*: *igFreshInp-def gFreshInp-def freshInp-def*
liftAll2-def liftAll-def lift-def split: *option.splits*)

lemma *igSwapInp-fromMOD*[*simp*]:
igSwapInp (*fromMOD MOD*) *zs z1 z2 iinp* =
lift2 Pair
(*swapInp zs z1 z2* (*lift fst iinp*))
(*gSwapInp MOD zs z1 z2* (*lift fst iinp*) (*lift snd iinp*))
by(*auto simp*: *igSwapInp-def swapInp-def gSwapInp-def lift-def lift2-def*
split: *option.splits*)

lemma *igSubstInp-fromMOD*[*simp*]:
igSubstInp (*fromMOD MOD*) *ys Y′Y y iinp* =
lift2 Pair
(*substInp ys* (*fst Y′Y*) *y* (*lift fst iinp*))
(*gSubstInp MOD ys* (*fst Y′Y*) (*snd Y′Y*) *y* (*lift fst iinp*) (*lift snd iinp*))
by(*auto simp*: *igSubstInp-def substInp-def2 gSubstInp-def lift-def lift2-def*
split: *option.splits*)

lemmas *input-fromMOD-simps* =
igWlsInp-fromMOD igFreshInp-fromMOD igSwapInp-fromMOD igSubstInp-fromMOD

. for bound inputs:

lemma *igWlsBinp-fromMOD*[*simp*]:
igWlsBinp (*fromMOD MOD*) *delta biinp* ⟷
(*wlsBinp delta* (*lift fst biinp*) ∧ *gWlsBinp MOD delta* (*lift snd biinp*))
apply (*intro iffI*)
subgoal apply(*simp add*: *liftAll2-def lift-def sameDom-def*
igWlsBinp-def wlsBinp-iff gWlsBinp-def split: *option.splits*) **.**
subgoal
unfolding *liftAll2-def lift-def sameDom-def*
igWlsBinp-def wlsBinp-iff gWlsBinp-def
by *simp* (*metis* (*no-types*, *lifting*) *eq-snd-iff fstI option.case-eq-if*
option.distinct(*1*) *option.simps*(*5*)) **.**

lemma *igFreshBinp-fromMOD*[*simp*]:
igFreshBinp (*fromMOD MOD*) *ys y biinp* ⟷
(*freshBinp ys y* (*lift fst biinp*) ∧
gFreshBinp MOD ys y (*lift fst biinp*) (*lift snd biinp*))
by (*auto simp*: *igFreshBinp-def gFreshBinp-def freshBinp-def*
liftAll2-def liftAll-def lift-def split: *option.splits*)

lemma *igSwapBinp-fromMOD*[*simp*]:
igSwapBinp (*fromMOD MOD*) *zs z1 z2 biinp* =
lift2 Pair
(*swapBinp zs z1 z2* (*lift fst biinp*))
(*gSwapBinp MOD zs z1 z2* (*lift fst biinp*) (*lift snd biinp*))
by(*auto simp*: *igSwapBinp-def swapBinp-def gSwapBinp-def lift-def lift2-def*
split: *option.splits*)

lemma *igSubstBinp-fromMOD*[*simp*]:

igSubstBinp (*fromMOD MOD*) *ys Y′Y y biinp* =
lift2 Pair
(*substBinp ys* (*fst Y′Y*) *y* (*lift fst biinp*))
(*gSubstBinp MOD ys* (*fst Y′Y*) (*snd Y′Y*) *y* (*lift fst biinp*) (*lift snd biinp*))
by(*auto simp*: *igSubstBinp-def substBinp-def2 gSubstBinp-def lift-def lift2-def*
split: *option.splits*)

lemmas *binput-fromMOD-simps* =
igWlsBinp-fromMOD igFreshBinp-fromMOD igSwapBinp-fromMOD igSubstBinp-fromMOD

Domain disjointness:

lemma *igWlsDisj-fromMOD*[*simp*]:
gWlsDisj MOD ⟹ *igWlsDisj* (*fromMOD MOD*)
unfolding *igWlsDisj-def gWlsDisj-def* **by** *auto*

lemma *igWlsAbsDisj-fromMOD*[*simp*]:
gWlsAbsDisj MOD ⟹ *igWlsAbsDisj* (*fromMOD MOD*)
unfolding *igWlsAbsDisj-def gWlsAbsDisj-def* **by** *fastforce*

lemma *igWlsAllDisj-fromMOD*[*simp*]:
gWlsAllDisj MOD ⟹ *igWlsAllDisj* (*fromMOD MOD*)
unfolding *igWlsAllDisj-def gWlsAllDisj-def* **by** *fastforce*

lemmas *igWlsAllDisj-fromMOD-simps* =
igWlsDisj-fromMOD igWlsAbsDisj-fromMOD igWlsAllDisj-fromMOD

Abstractions only within IsInBar:

lemma *igWlsAbsIsInBar-fromMOD*[*simp*]:
gWlsAbsIsInBar MOD ⟹ *igWlsAbsIsInBar* (*fromMOD MOD*)
unfolding *gWlsAbsIsInBar-def igWlsAbsIsInBar-def* **by** *simp*

The constructs preserve the domains:

lemma *igVarIPresIGWls-fromMOD*[*simp*]:
gVarPresGWls MOD ⟹ *igVarIPresIGWls* (*fromMOD MOD*)
unfolding *igVarIPresIGWls-def gVarPresGWls-def* **by** *simp*

lemma *igAbsIPresIGWls-fromMOD*[*simp*]:
gAbsPresGWls MOD ⟹ *igAbsIPresIGWls* (*fromMOD MOD*)
unfolding *igAbsIPresIGWls-def gAbsPresGWls-def* **by** *simp*

lemma *igOpIPresIGWls-fromMOD*[*simp*]:
gOpPresGWls MOD ⟹ *igOpIPresIGWls* (*fromMOD MOD*)
unfolding *igOpIPresIGWls-def gOpPresGWls-def* **by** *simp*

lemma *igConsIPresIGWls-fromMOD*[*simp*]:
gConsPresGWls MOD ⟹ *igConsIPresIGWls* (*fromMOD MOD*)
unfolding *igConsIPresIGWls-def gConsPresGWls-def* **by** *simp*

lemmas *igConsIPresIGWls-fromMOD-simps* =

igVarIPresIGWls-fromMOD igAbsIPresIGWls-fromMOD igOpIPresIGWls-fromMOD igConsIPresIGWls-fromMOD

Swap preserves the domains:

lemma *igSwapIPresIGWls-fromMOD*[*simp*]:
gSwapPresGWls MOD ⟹ *igSwapIPresIGWls* (*fromMOD MOD*)
unfolding *igSwapIPresIGWls-def gSwapPresGWls-def* **by** *simp*

lemma *igSwapAbsIPresIGWlsAbs-fromMOD*[*simp*]:
gSwapAbsPresGWlsAbs MOD ⟹ *igSwapAbsIPresIGWlsAbs* (*fromMOD MOD*)
unfolding *igSwapAbsIPresIGWlsAbs-def gSwapAbsPresGWlsAbs-def* **by** *simp*

lemma *igSwapAllIPresIGWlsAll-fromMOD*[*simp*]:
gSwapAllPresGWlsAll MOD ⟹ *igSwapAllIPresIGWlsAll* (*fromMOD MOD*)
unfolding *igSwapAllIPresIGWlsAll-def gSwapAllPresGWlsAll-def* **by** *simp*

lemmas *igSwapAllIPresIGWlsAll-fromMOD-simps* =
igSwapIPresIGWls-fromMOD igSwapAbsIPresIGWlsAbs-fromMOD igSwapAllIPresIG-WlsAll-fromMOD

Subst preserves the domains:

lemma *igSubstIPresIGWls-fromMOD*[*simp*]:
gSubstPresGWls MOD ⟹ *igSubstIPresIGWls* (*fromMOD MOD*)
unfolding *igSubstIPresIGWls-def gSubstPresGWls-def* **by** *simp*

lemma *igSubstAbsIPresIGWlsAbs-fromMOD*[*simp*]:
gSubstAbsPresGWlsAbs MOD ⟹ *igSubstAbsIPresIGWlsAbs* (*fromMOD MOD*)
unfolding *igSubstAbsIPresIGWlsAbs-def gSubstAbsPresGWlsAbs-def* **by** *simp*

lemma *igSubstAllIPresIGWlsAll-fromMOD*[*simp*]:
gSubstAllPresGWlsAll MOD ⟹ *igSubstAllIPresIGWlsAll* (*fromMOD MOD*)
unfolding *igSubstAllIPresIGWlsAll-def gSubstAllPresGWlsAll-def* **by** *simp*

lemmas *igSubstAllIPresIGWlsAll-fromMOD-simps* =
igSubstIPresIGWls-fromMOD igSubstAbsIPresIGWlsAbs-fromMOD igSubstAllIPresIG-WlsAll-fromMOD

The fresh clauses:

lemma *igFreshIGVar-fromMOD*[*simp*]:
gFreshGVar MOD ⟹ *igFreshIGVar* (*fromMOD MOD*)
unfolding *igFreshIGVar-def gFreshGVar-def* **by** *simp*

lemma *igFreshIGAbs1-fromMOD*[*simp*]:
gFreshGAbs1 MOD ⟹ *igFreshIGAbs1* (*fromMOD MOD*)
unfolding *igFreshIGAbs1-def gFreshGAbs1-def* **by** *auto*

lemma *igFreshIGAbs2-fromMOD*[*simp*]:
gFreshGAbs2 MOD ⟹ *igFreshIGAbs2* (*fromMOD MOD*)
unfolding *igFreshIGAbs2-def gFreshGAbs2-def* **by** *auto*

lemma *igFreshIGOp-fromMOD*[*simp*]:
gFreshGOp MOD ⟹ *igFreshIGOp* (*fromMOD MOD*)
unfolding *igFreshIGOp-def gFreshGOp-def* **by** *simp*

lemma *igFreshCls-fromMOD*[*simp*]:
gFreshCls MOD ⟹ *igFreshCls* (*fromMOD MOD*)
unfolding *igFreshCls-def gFreshCls-def* **by** *simp*

lemmas *igFreshCls-fromMOD-simps* =
igFreshIGVar-fromMOD igFreshIGAbs1-fromMOD igFreshIGAbs2-fromMOD
igFreshIGOp-fromMOD igFreshCls-fromMOD

The swap clauses

lemma *igSwapIGVar-fromMOD*[*simp*]:
gSwapGVar MOD ⟹ *igSwapIGVar* (*fromMOD MOD*)
unfolding *igSwapIGVar-def gSwapGVar-def* **by** *simp*

lemma *igSwapIGAbs-fromMOD*[*simp*]:
gSwapGAbs MOD ⟹ *igSwapIGAbs* (*fromMOD MOD*)
unfolding *igSwapIGAbs-def gSwapGAbs-def* **by** *auto*

lemma *igSwapIGOp-fromMOD*[*simp*]:
gSwapGOp MOD ⟹ *igSwapIGOp* (*fromMOD MOD*)
by (*auto simp*: *igSwapIGOp-def gSwapGOp-def lift-lift2*)

lemma *igSwapCls-fromMOD*[*simp*]:
gSwapCls MOD ⟹ *igSwapCls* (*fromMOD MOD*)
unfolding *igSwapCls-def gSwapCls-def* **by** *simp*

lemmas *igSwapCls-fromMOD-simps* =
igSwapIGVar-fromMOD igSwapIGAbs-fromMOD
igSwapIGOp-fromMOD igSwapCls-fromMOD

The subst clauses

lemma *igSubstIGVar1-fromMOD*[*simp*]:
gSubstGVar1 MOD ⟹ *igSubstIGVar1* (*fromMOD MOD*)
unfolding *igSubstIGVar1-def gSubstGVar1-def* **by** *simp*

lemma *igSubstIGVar2-fromMOD*[*simp*]:
gSubstGVar2 MOD ⟹ *igSubstIGVar2* (*fromMOD MOD*)
unfolding *igSubstIGVar2-def gSubstGVar2-def* **by** *simp*

lemma *igSubstIGAbs-fromMOD*[*simp*]:
gSubstGAbs MOD ⟹ *igSubstIGAbs* (*fromMOD MOD*)
unfolding *igSubstIGAbs-def gSubstGAbs-def* **by** *fastforce*+

lemma *igSubstIGOp-fromMOD*[*simp*]:
gSubstGOp MOD ⟹ *igSubstIGOp* (*fromMOD MOD*)

by(*auto simp*: *igSubstIGOp-def gSubstGOp-def lift-lift2*)

lemma *igSubstCls-fromMOD*[*simp*]:
gSubstCls MOD $\Longrightarrow$ *igSubstCls* (*fromMOD MOD*)
unfolding *igSubstCls-def gSubstCls-def* **by** *simp*

lemmas *igSubstCls-fromMOD-simps* =
igSubstIGVar1-fromMOD igSubstIGVar2-fromMOD igSubstIGAbs-fromMOD
igSubstIGOp-fromMOD igSubstCls-fromMOD

Abstraction swapping congruence:

lemma *igAbsCongS-fromMOD*[*simp*]:
assumes *gAbsCongS MOD*
shows *igAbsCongS* (*fromMOD MOD*)
using *assms*
unfolding *igAbsCongS-def gAbsCongS-def*
apply *simp*
apply *clarify*
by (*intro conjI*, *erule wls-Abs-swap-cong*) *blast*+

Abstraction renaming:

lemma *igAbsRen-fromMOD*[*simp*]:
gAbsRen MOD $\Longrightarrow$ *igAbsRen* (*fromMOD MOD*)
unfolding *igAbsRen-def gAbsRen-def vsubst-def* **by** *auto*

Models:

lemma *iwlsFSw-fromMOD*[*simp*]:
wlsFSw MOD $\Longrightarrow$ *iwlsFSw* (*fromMOD MOD*)
unfolding *iwlsFSw-def wlsFSw-def* **by** *simp*

lemma *iwlsFSb-fromMOD*[*simp*]:
wlsFSb MOD $\Longrightarrow$ *iwlsFSb* (*fromMOD MOD*)
unfolding *iwlsFSb-def wlsFSb-def* **by** *simp*

lemma *iwlsFSwSb-fromMOD*[*simp*]:
wlsFSwSb MOD $\Longrightarrow$ *iwlsFSwSb* (*fromMOD MOD*)
unfolding *iwlsFSwSb-def wlsFSwSb-def* **by** *simp*

lemma *iwlsFSbSw-fromMOD*[*simp*]:
wlsFSbSw MOD $\Longrightarrow$ *iwlsFSbSw* (*fromMOD MOD*)
unfolding *iwlsFSbSw-def wlsFSbSw-def* **by** *simp*

lemmas *iwlsModel-fromMOD-simps* =
iwlsFSw-fromMOD iwlsFSb-fromMOD
iwlsFSwSb-fromMOD iwlsFSbSw-fromMOD

lemmas *fromMOD-predicate-simps* =
igWlsAllDisj-fromMOD-simps

igConsIPresIGWls-fromMOD-simps
igSwapAllIPresIGWlsAll-fromMOD-simps
igSubstAllIPresIGWlsAll-fromMOD-simps
igFreshCls-fromMOD-simps
igSwapCls-fromMOD-simps
igSubstCls-fromMOD-simps
igAbsCongS-fromMOD
igAbsRen-fromMOD
iwlsModel-fromMOD-simps

lemmas *fromMOD-simps* =
fromMOD-basic-simps
input-fromMOD-simps
binput-fromMOD-simps
fromMOD-predicate-simps

10.5 The recursion-iteration "identity trick"

Here we show that any construct-preserving map from terms to "fromMOD MOD" is the identity on its first projection – this is the main trick when reducing recursion to iteration.

lemma *ipresCons-fromMOD-fst*:
assumes *ipresCons h hA* (*fromMOD MOD*)
shows (*wls s X* ⟶ *fst* (*h X*) = *X*) ∧ (*wlsAbs* (*us*,*s'*) *A* ⟶ *fst* (*hA A*) = *A*)
proof(*induction rule*: *wls-rawInduct*)
next
  case (*Op delta inp binp*)
  hence *lift* (*fst* ∘ *h*) *inp* = *inp* ∧ *lift* (*fst* ∘ *hA*) *binp* = *binp*
  by (*simp add*: *lift-def fun-eq-iff liftAll2-def*
   wlsInp-iff wlsBinp-iff sameDom-def split: *option.splits*)
  (*metis not-Some-eq old.prod.exhaust*)
  then show *?case*
  using *assms Op* **by** (*auto simp*: *ipresCons-def ipresOp-def lift-comp*)
qed(*insert assms*, *auto simp*: *ipresVar-def ipresCons-def ipresAbs-def*)

lemma *ipresCons-fromMOD-fst-simps*[*simp*]:
⟦*ipresCons h hA* (*fromMOD MOD*); *wls s X*⟧
 ⟹ *fst* (*h X*) = *X*

⟦*ipresCons h hA* (*fromMOD MOD*); *wlsAbs* (*us*,*s'*) *A*⟧
 ⟹ *fst* (*hA A*) = *A*
using *ipresCons-fromMOD-fst* **by** *blast+*

lemma *ipresCons-fromMOD-fst-inp*[*simp*]:
ipresCons h hA (*fromMOD MOD*) ⟹ *wlsInp delta inp* ⟹ *lift* (*fst o h*) *inp* = *inp*
by (*force simp add*: *lift-def fun-eq-iff liftAll2-def*
wlsInp-iff sameDom-def split: *option.splits*)

lemma *ipresCons-fromMOD-fst-binp*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *wlsBinp delta binp* $\Longrightarrow$ *lift* (*fst o hA*) *binp* = *binp*
by (*force simp add*: *lift-def fun-eq-iff liftAll2-def wlsBinp-iff sameDom-def split*: *option.splits*)

lemmas *ipresCons-fromMOD-fst-all-simps* =
ipresCons-fromMOD-fst-simps ipresCons-fromMOD-fst-inp ipresCons-fromMOD-fst-binp

10.6 From iteration morphisms to morphisms

The transition map:

definition *fromIMor* ::
(('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* ⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* × '*gTerm*)
⇒
(('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*term* ⇒ '*gTerm*)
where *fromIMor h* ≡ *snd o h*

definition *fromIMorAbs* ::
(('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* ⇒
('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* × '*gAbs*)
⇒
(('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*abs* ⇒ '*gAbs*)
where *fromIMorAbs hA* ≡ *snd o hA*

Basic simplification rules:

lemma *fromIMor*[*simp*]: *fromIMor h X'* = *snd* (*h X'*)
unfolding *fromIMor-def* **by** *simp*

lemma *fromIMorAbs*[*simp*]: *fromIMorAbs hA A'* = *snd* (*hA A'*)
unfolding *fromIMorAbs-def* **by** *simp*

lemma *fromIMor-snd-inp*[*simp*]:
wlsInp delta inp $\Longrightarrow$ *lift* (*fromIMor h*) *inp* = *lift* (*snd o h*) *inp*
by (*auto simp*: *lift-def split*: *option.splits*)

lemma *fromIMorAbs-snd-binp*[*simp*]:
wlsBinp delta binp $\Longrightarrow$ *lift* (*fromIMorAbs hA*) *binp* = *lift* (*snd o hA*) *binp*
by (*auto simp*: *lift-def split*: *option.splits*)

lemmas *fromIMor-basic-simps* =
fromIMor fromIMorAbs fromIMor-snd-inp fromIMorAbs-snd-binp

Predicate simplification rules

Domain preservation

lemma *presWls-fromIMor*[*simp*]:

ipresWls h (*fromMOD MOD*) $\Longrightarrow$ *presWls* (*fromIMor h*) *MOD*
unfolding *ipresWls-def presWls-def* **by** *simp*

lemma *presWlsAbs-fromIMorAbs*[*simp*]:
ipresWlsAbs hA (*fromMOD MOD*) $\Longrightarrow$ *presWlsAbs* (*fromIMorAbs hA*) *MOD*
unfolding *ipresWlsAbs-def presWlsAbs-def* **by** *simp*

lemma *presWlsAll-fromIMorAll*[*simp*]:
ipresWlsAll h hA (*fromMOD MOD*) $\Longrightarrow$ *presWlsAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *ipresWlsAll-def presWlsAll-def* **by** *simp*

lemmas *presWlsAll-fromIMorAll-simps* =
presWls-fromIMor presWlsAbs-fromIMorAbs presWlsAll-fromIMorAll

Preservation of the constructs

lemma *presVar-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *presVar* (*fromIMor h*) *MOD*
unfolding *ipresCons-def ipresVar-def presVar-def* **by** *simp*

lemma *presAbs-fromIMor*[*simp*]:
assumes *ipresCons h hA* (*fromMOD MOD*)
shows *presAbs* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
using *assms* **unfolding** *ipresCons-def ipresAbs-def presAbs-def*
using *assms* **by** *fastforce*

lemma *presOp-fromIMor*[*simp*]:
assumes *ipresCons h hA* (*fromMOD MOD*)
shows *presOp* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
using *assms* **unfolding** *ipresCons-def ipresOp-def presOp-def*
using *assms* **by** (*auto simp*: *lift-comp*)

lemma *presCons-fromIMor*[*simp*]:
assumes *ipresCons h hA* (*fromMOD MOD*)
shows *presCons* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *ipresCons-def presCons-def* **using** *assms* **by** *simp*

lemmas *presCons-fromIMor-simps* =
presVar-fromIMor presAbs-fromIMor presOp-fromIMor presCons-fromIMor

Preservation of freshness

lemma *presFresh-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresFresh h* (*fromMOD MOD*)
$\Longrightarrow$ *presFresh* (*fromIMor h*) *MOD*
unfolding *ipresFresh-def presFresh-def* **by** *simp*

lemma *presFreshAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresFreshAbs hA* (*fromMOD MOD*)
$\Longrightarrow$ *presFreshAbs* (*fromIMorAbs hA*) *MOD*

unfolding *ipresFreshAbs-def presFreshAbs-def* **by** *simp*

lemma *presFreshAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresFreshAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presFreshAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*

unfolding *ipresFreshAll-def presFreshAll-def* **by** *simp*

lemmas *presFreshAll-fromIMor-simps* =
presFresh-fromIMor presFreshAbs-fromIMor presFreshAll-fromIMor

Preservation of swap

lemma *presSwap-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSwap h* (*fromMOD MOD*)
$\Longrightarrow$ *presSwap* (*fromIMor h*) *MOD*
unfolding *ipresSwap-def presSwap-def* **by** *simp*

lemma *presSwapAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSwapAbs hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSwapAbs* (*fromIMorAbs hA*) *MOD*
unfolding *ipresSwapAbs-def presSwapAbs-def* **by** *simp*

lemma *presSwapAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSwapAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSwapAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *ipresSwapAll-def presSwapAll-def* **by** *simp*

lemmas *presSwapAll-fromIMor-simps* =
presSwap-fromIMor presSwapAbs-fromIMor presSwapAll-fromIMor

Preservation of subst

lemma *presSubst-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSubst h* (*fromMOD MOD*)
$\Longrightarrow$ *presSubst* (*fromIMor h*) *MOD*
unfolding *ipresSubst-def presSubst-def* **by** *auto*

lemma *presSubstAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSubstAbs h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSubstAbs* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *ipresSubstAbs-def presSubstAbs-def* **by** *auto*

lemma *presSubstAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSubstAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSubstAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *ipresSubstAll-def presSubstAll-def* **by** *simp*

lemmas *presSubstAll-fromIMor-simps* =
presSubst-fromIMor presSubstAbs-fromIMor presSubstAll-fromIMor

Morphisms

lemma *fromIMor-termFSwMorph*[*simp*]:
termFSwImorph h hA (*fromMOD MOD*) $\Longrightarrow$ *termFSwMorph* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *termFSwImorph-def termFSwMorph-def* **by** *simp*

lemma *fromIMor-termFSbMorph*[*simp*]:
termFSbImorph h hA (*fromMOD MOD*) $\Longrightarrow$ *termFSbMorph* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
unfolding *termFSbImorph-def termFSbMorph-def* **by** *simp*

lemma *fromIMor-termFSwSbMorph*[*simp*]:
assumes *termFSwSbImorph h hA* (*fromMOD MOD*)
shows *termFSwSbMorph* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
using *assms* **unfolding** *termFSwSbImorph-defs1*
using *assms* **unfolding** *termFSwSbImorph-def termFSwSbMorph-def* **by** *simp*

lemmas *mor-fromIMor-simps* =
fromIMor-termFSwMorph fromIMor-termFSbMorph fromIMor-termFSwSbMorph

lemmas *fromIMor-predicate-simps* =
presCons-fromIMor-simps
presFreshAll-fromIMor-simps
presSwapAll-fromIMor-simps
presSubstAll-fromIMor-simps
mor-fromIMor-simps

lemmas *fromIMor-simps* =
fromIMor-basic-simps fromIMor-predicate-simps

10.7 The recursion theorem

The recursion maps:

definition *rec* **where** *rec MOD* $\equiv$ *fromIMor* (*iter* (*fromMOD MOD*))

definition *recAbs* **where** *recAbs MOD* $\equiv$ *fromIMorAbs* (*iterAbs* (*fromMOD MOD*))

Existence:

theorem *wlsFSw-recAll-termFSwMorph*:
wlsFSw MOD $\Longrightarrow$ *termFSwMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
by (*simp add*: *rec-def recAbs-def iwlsFSw-iterAll-termFSwImorph*)

theorem *wlsFSb-recAll-termFSbMorph*:
wlsFSb MOD $\Longrightarrow$ *termFSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
by (*simp add*: *rec-def recAbs-def iwlsFSb-iterAll-termFSbImorph*)

theorem *wlsFSwSb-recAll-termFSwSbMorph*:
wlsFSwSb MOD $\Longrightarrow$ *termFSwSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
by (*simp add*: *rec-def recAbs-def iwlsFSwSb-iterAll-termFSwSbImorph*)

theorem *wlsFSbSw-recAll-termFSwSbMorph*:
wlsFSbSw MOD $\Longrightarrow$ *termFSwSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
by (*simp add*: *rec-def recAbs-def iwlsFSbSw-iterAll-termFSwSbImorph*)

Uniqueness:

lemma *presCons-unique*:
assumes *presCons f fA MOD* **and** *presCons g gA MOD*
shows (*wls s X* $\longrightarrow$ *f X* = *g X*) $\wedge$ (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *fA A* = *gA A*)
proof(*induction rule*: *wls-rawInduct*)
  case (*Op delta inp binp*)
  hence *lift f inp* = *lift g inp* $\wedge$ *lift fA binp* = *lift gA binp*
  apply(*simp add*: *lift-def wlsInp-iff wlsBinp-iff sameDom-def liftAll2-def fun-eq-iff split*: *option.splits*)
  by (*metis not-Some-eq old.prod.exhaust*)
  then show *?case* **using** *assms Op* **unfolding** *presCons-def presOp-def* **by** *simp*
qed(*insert assms*, *auto simp*: *presVar-def presCons-def presAbs-def*)

theorem *wlsFSw-recAll-unique-presCons*:
assumes *wlsFSw MOD* **and** *presCons h hA MOD*
shows (*wls s X* $\longrightarrow$ *h X* = *rec MOD X*) $\wedge$
  (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *hA A* = *recAbs MOD A*)
using *assms wlsFSw-recAll-termFSwMorph*
by (*intro presCons-unique*) (*auto simp*: *termFSwMorph-def*)

theorem *wlsFSb-recAll-unique-presCons*:
assumes *wlsFSb MOD* **and** *presCons h hA MOD*
shows (*wls s X* $\longrightarrow$ *h X* = *rec MOD X*) $\wedge$
  (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *hA A* = *recAbs MOD A*)
using *assms wlsFSb-recAll-termFSbMorph*
by (*intro presCons-unique*) (*auto simp*: *termFSbMorph-def*)

theorem *wlsFSwSb-recAll-unique-presCons*:
assumes *wlsFSwSb MOD* **and** *presCons h hA MOD*
shows (*wls s X* $\longrightarrow$ *h X* = *rec MOD X*) $\wedge$
  (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *hA A* = *recAbs MOD A*)
using *assms wlsFSw-recAll-unique-presCons* **unfolding** *wlsFSwSb-def* **by** *blast*

theorem *wlsFSbSw-recAll-unique-presCons*:
assumes *wlsFSbSw MOD* **and** *presCons h hA MOD*
shows (*wls s X* $\longrightarrow$ *h X* = *rec MOD X*) $\wedge$
  (*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ *hA A* = *recAbs MOD A*)
using *assms wlsFSb-recAll-unique-presCons* **unfolding** *wlsFSbSw-def* **by** *blast*

10.8 Models that are even “closer" to the term model

We describe various conditions (later referred to as “extra clauses" or “extra conditions") that, when satisfied by models, yield the recursive maps (1) freshness-preserving and/or (2) injective and/or (3) surjective, thus bringing the considered models "closer" to (being isomorphic to) the term model. The extreme case, when all of (1)-(3) above are ensured, means indeed isomorphism to the term model – this is in fact an abstract characterization of the term model.

10.8.1 Relevant predicates on models

The fresh clauses reversed

definition *gFreshGVarRev* **where**
gFreshGVarRev MOD ≡ ∀ *xs y x*.
gFresh MOD xs y (*Var xs x*) (*gVar MOD xs x*) ⟶ *y* ≠ *x*

definition *gFreshGAbsRev* **where**
gFreshGAbsRev MOD ≡ ∀ *ys y xs x s X' X*.
isInBar (*xs*,*s*) ∧ *wls s X'* ∧ *gWls MOD s X* ⟶
gFreshAbs MOD ys y (*Abs xs x X'*) (*gAbs MOD xs x X' X*) ⟶
(*ys* = *xs* ∧ *y* = *x*) ∨ *gFresh MOD ys y X' X*

definition *gFreshGOpRev* **where**
gFreshGOpRev MOD ≡ ∀ *ys y delta inp' inp binp' binp*.
wlsInp delta inp' ∧ *gWlsInp MOD delta inp* ∧ *wlsBinp delta binp'* ∧ *gWlsBinp MOD delta binp* ⟶
gFresh MOD ys y (*Op delta inp' binp'*) (*gOp MOD delta inp' inp binp' binp*) ⟶
gFreshInp MOD ys y inp' inp ∧ *gFreshBinp MOD ys y binp' binp*

definition *gFreshClsRev* **where**
gFreshClsRev MOD ≡ *gFreshGVarRev MOD* ∧ *gFreshGAbsRev MOD* ∧ *gFreshGOpRev MOD*

lemmas *gFreshClsRev-defs* = *gFreshClsRev-def*
gFreshGVarRev-def gFreshGAbsRev-def gFreshGOpRev-def

Injectiveness of the construct operators

definition *gVarInj* **where**
gVarInj MOD ≡ ∀ *xs x y*. *gVar MOD xs x* = *gVar MOD xs y* ⟶ *x* = *y*

definition *gAbsInj* **where**
gAbsInj MOD ≡ ∀ *xs s x X' X X1' X1*.
isInBar (*xs*,*s*) ∧ *wls s X'* ∧ *gWls MOD s X* ∧ *wls s X1'* ∧ *gWls MOD s X1* ∧
gAbs MOD xs x X' X = *gAbs MOD xs x X1' X1*
⟶
X = *X1*

definition *gOpInj* **where**
gOpInj MOD ≡ ∀ *delta delta1 inp' binp' inp binp inp1' binp1' inp1 binp1*.
wlsInp delta inp' ∧ *wlsBinp delta binp'* ∧ *gWlsInp MOD delta inp* ∧ *gWlsBinp MOD delta binp* ∧

wlsInp delta1 inp1′ ∧ *wlsBinp delta1 binp1′* ∧ *gWlsInp MOD delta1 inp1* ∧ *gWlsBinp MOD delta1 binp1* ∧
stOf delta = *stOf delta1* ∧
gOp MOD delta inp′ inp binp′ binp = *gOp MOD delta1 inp1′ inp1 binp1′ binp1*
⟶
delta = *delta1* ∧ *inp* = *inp1* ∧ *binp* = *binp1*

definition *gVarGOpInj* **where**
gVarGOpInj MOD ≡ ∀ *xs x delta inp′ binp′ inp binp.*
wlsInp delta inp′ ∧ *wlsBinp delta binp′* ∧ *gWlsInp MOD delta inp* ∧ *gWlsBinp MOD delta binp* ∧
asSort xs = *stOf delta*
⟶
gVar MOD xs x ≠ *gOp MOD delta inp′ inp binp′ binp*

definition *gConsInj* **where**
gConsInj MOD ≡ *gVarInj MOD* ∧ *gAbsInj MOD* ∧ *gOpInj MOD* ∧ *gVarGOpInj MOD*

lemmas *gConsInj-defs* = *gConsInj-def*
gVarInj-def gAbsInj-def gOpInj-def gVarGOpInj-def

Abstraction renaming for swapping

definition *gAbsRenS* **where**
gAbsRenS MOD ≡ ∀ *xs y x s X′ X.*
isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ⟶
fresh xs y X′ ∧ *gFresh MOD xs y X′ X* ⟶
gAbs MOD xs y (*X′* #[*y* ∧ *x*]-*xs*) (*gSwap MOD xs y x X′ X*) =
gAbs MOD xs x X′ X

Indifference to the general-recursive argument

. This "indifference" property says that the construct operators from the model only depend on the generalized item (i.e., generalized term or abstraction) argument, and *not* on the "item" (i.e., concrete term or abstraction) argument. In other words, the model constructs correspond to *iterative clauses*, and not to the more general notion of "general-recursive" clause.

definition *gAbsIndif* **where**
gAbsIndif MOD ≡ ∀ *xs s x X1′ X2′ X.*
isInBar (*xs*,*s*) ∧ *wls s X1′* ∧ *wls s X2′* ∧ *gWls MOD s X* ⟶
gAbs MOD xs x X1′ X = *gAbs MOD xs x X2′ X*

definition *gOpIndif* **where**
gOpIndif MOD ≡ ∀ *delta inp1′ inp2′ inp binp1′ binp2′ binp.*
wlsInp delta inp1′ ∧ *wlsBinp delta binp1′* ∧ *wlsInp delta inp2′* ∧ *wlsBinp delta binp2′* ∧
gWlsInp MOD delta inp ∧ *gWlsBinp MOD delta binp*
⟶
gOp MOD delta inp1′ inp binp1′ binp = *gOp MOD delta inp2′ inp binp2′ binp*

definition *gConsIndif* **where**
gConsIndif MOD ≡ *gOpIndif MOD* ∧ *gAbsIndif MOD*

lemmas *gConsIndif-defs* = *gConsIndif-def gAbsIndif-def gOpIndif-def*

Inductiveness

. Inductiveness of a model means the satisfaction of a minimal inductive principle ("minimal" in the sense that no fancy swapping or freshness induction-friendly conditions are involved).

definition *gInduct* **where**
gInduct MOD ≡ ∀ *phi phiAbs s X us s′ A*.
(
(∀ *xs x*. *phi* (*asSort xs*) (*gVar MOD xs x*))
∧
(∀ *delta inp′ inp binp′ binp*.
wlsInp delta inp′ ∧ *wlsBinp delta binp′* ∧ *gWlsInp MOD delta inp* ∧ *gWlsBinp MOD delta binp* ∧
liftAll2 phi (*arOf delta*) *inp* ∧ *liftAll2 phiAbs* (*barOf delta*) *binp*
⟶ *phi* (*stOf delta*) (*gOp MOD delta inp′ inp binp′ binp*))
∧
(∀ *xs s x X′ X*.
isInBar (*xs*,*s*) ∧ *wls s X′* ∧ *gWls MOD s X* ∧
phi s X
⟶ *phiAbs* (*xs*,*s*) (*gAbs MOD xs x X′ X*))
)
⟶
(*gWls MOD s X* ⟶ *phi s X*) ∧
(*gWlsAbs MOD* (*us*,*s′*) *A* ⟶ *phiAbs* (*us*,*s′*) *A*)

lemma *gInduct-elim*:
assumes *gInduct MOD* **and**
Var: ⋀ *xs x*. *phi* (*asSort xs*) (*gVar MOD xs x*) **and**
Op:
⋀ *delta inp′ inp binp′ binp*.
⟦*wlsInp delta inp′*; *wlsBinp delta binp′*; *gWlsInp MOD delta inp*; *gWlsBinp MOD delta binp*;
liftAll2 phi (*arOf delta*) *inp*; *liftAll2 phiAbs* (*barOf delta*) *binp*⟧
⟹ *phi* (*stOf delta*) (*gOp MOD delta inp′ inp binp′ binp*) **and**
Abs:
⋀ *xs s x X′ X*.
⟦*isInBar* (*xs*,*s*); *wls s X′*; *gWls MOD s X*; *phi s X*⟧
⟹ *phiAbs* (*xs*,*s*) (*gAbs MOD xs x X′ X*)
shows
(*gWls MOD s X* ⟶ *phi s X*) ∧
(*gWlsAbs MOD* (*us*,*s′*) *A* ⟶ *phiAbs* (*us*,*s′*) *A*)
using *assms* **unfolding** *gInduct-def*
apply(*elim allE*[*of - phi*] *allE*[*of - phiAbs*] *allE*[*of - s*] *allE*[*of - X*])

apply(*elim allE*[*of - us*] *allE*[*of - s'*] *allE*[*of - A*])
by *blast*

10.8.2 Relevant predicates on maps from the term model

Reflection of freshness

definition *reflFresh* **where**
reflFresh h MOD ≡ ∀ *ys y s X*.
wls s X ⟶
gFresh MOD ys y X (*h X*) ⟶ *fresh ys y X*

definition *reflFreshAbs* **where**
reflFreshAbs hA MOD ≡ ∀ *ys y us s A*.
wlsAbs (*us*,*s*) *A* ⟶
gFreshAbs MOD ys y A (*hA A*) ⟶ *freshAbs ys y A*

definition *reflFreshAll* **where**
reflFreshAll h hA MOD ≡ *reflFresh h MOD* ∧ *reflFreshAbs hA MOD*

lemmas *reflFreshAll-defs* = *reflFreshAll-def*
reflFresh-def reflFreshAbs-def

Injectiveness

definition *isInj* **where**
isInj h ≡ ∀ *s X Y*.
wls s X ∧ *wls s Y* ⟶
h X = *h Y* ⟶ *X* = *Y*

definition *isInjAbs* **where**
isInjAbs hA ≡ ∀ *us s A B*.
wlsAbs (*us*,*s*) *A* ∧ *wlsAbs* (*us*,*s*) *B* ⟶
hA A = *hA B* ⟶ *A* = *B*

definition *isInjAll* **where**
isInjAll h hA ≡ *isInj h* ∧ *isInjAbs hA*

lemmas *isInjAll-defs* = *isInjAll-def*
isInj-def isInjAbs-def

Surjectiveness

definition *isSurj* **where**
isSurj h MOD ≡ ∀ *s X*.
gWls MOD s X ⟶
(∃ *X'*. *wls s X'* ∧ *h X'* = *X*)

definition *isSurjAbs* **where**
isSurjAbs hA MOD ≡ ∀ *us s A*.
gWlsAbs MOD (*us*,*s*) *A* ⟶

($\exists$ *A′*. *wlsAbs* (*us*,*s*) *A′* $\wedge$ *hA A′* = *A*)

definition *isSurjAll* **where**
isSurjAll h hA MOD $\equiv$ *isSurj h MOD* $\wedge$ *isSurjAbs hA MOD*

lemmas *isSurjAll-defs* = *isSurjAll-def*
isSurj-def isSurjAbs-def

10.8.3 Criterion for the reflection of freshness

First an auxiliary fact, independent of the type of model:

lemma *gFreshClsRev-recAll-reflFreshAll*:
assumes *pWls*: *presWlsAll* (*rec MOD*) (*recAbs MOD*) *MOD*
and *pCons*: *presCons* (*rec MOD*) (*recAbs MOD*) *MOD*
and *pFresh*: *presFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
and $**$: *gFreshClsRev MOD*
shows *reflFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
proof−
 let *?h* = *rec MOD* **let** *?hA* = *recAbs MOD*
 have *pWlsInps*[*simp*]:
 $\bigwedge$ *delta inp*. *wlsInp delta inp* $\Longrightarrow$ *gWlsInp MOD delta* (*lift ?h inp*)
 $\bigwedge$ *delta binp*. *wlsBinp delta binp* $\Longrightarrow$ *gWlsBinp MOD delta* (*lift ?hA binp*)
 using *pWls presWls-wlsInp presWls-wlsBinp* **unfolding** *presWlsAll-def* **by** *auto*
 {fix *ys y s X us s′ A*
  have
  (*wls s X* $\longrightarrow$ *gFresh MOD ys y X* (*rec MOD X*) $\longrightarrow$ *fresh ys y X*) $\wedge$
   (*wlsAbs* (*us*,*s′*) *A* $\longrightarrow$ *gFreshAbs MOD ys y A* (*recAbs MOD A*) $\longrightarrow$ *freshAbs ys y A*)
  proof(*induction rule*: *wls-induct*)
   case (*Var xs x*)
   then show *?case* **using** *assms*
    by (*fastforce simp*: *presWlsAll-defs presCons-defs gFreshClsRev-def gFreshGVarRev-def*)
  next
   case (*Op delta inp binp*)
   show *?case* **proof** *safe*
    let *?ar* = *arOf delta* **let** *?bar* = *barOf delta* **let** *?st* = *stOf delta*
   let *?linp* = *lift ?h inp* **let** *?lbinp* = *lift ?hA binp*
   assume *gFresh MOD ys y* (*Op delta inp binp*) (*rec MOD* (*Op delta inp binp*))
    hence *gFresh MOD ys y* (*Op delta inp binp*) (*gOp MOD delta inp ?linp binp ?lbinp*)
   using *assms Op* **by** (*simp add*: *presCons-def presOp-def*)
   hence *gFreshInp MOD ys y inp ?linp* $\wedge$ *gFreshBinp MOD ys y binp ?lbinp*
   using *Op* $**$ **by** (*force simp*: *gFreshClsRev-def gFreshGOpRev-def*)
   with *Op* **have** *freshInp*: *freshInp ys y inp* $\wedge$ *freshBinp ys y binp*
    by (*simp add*: *freshInp-def freshBinp-def liftAll-def gFreshInp-def gFreshBinp-def liftAll2-def lift-def*
  sameDom-def wlsInp-iff wlsBinp-iff split: *option.splits*) (*metis eq-snd-iff not-Some-eq*)

thus *fresh ys y* (*Op delta inp binp*) **using** *Op* **by** *auto*
qed
next
case (*Abs s xs x X*)
show *?case* **proof** *safe*
have *hX*: *gWls MOD s* (*?h X*) **using** *Abs pWls* **unfolding** *presWlsAll-defs* **by** *simp*
assume *gFreshAbs MOD ys y* (*Abs xs x X*) (*recAbs MOD* (*Abs xs x X*))
hence *gFreshAbs MOD ys y* (*Abs xs x X*) (*gAbs MOD xs x X* (*rec MOD X*))
using *Abs* **by** (*metis pCons presAbs-def presCons-def*)
moreover have *?hA* (*Abs xs x X*) = *gAbs MOD xs x X* (*?h X*)
using *Abs pCons* **unfolding** *presCons-defs* **by** *blast*
ultimately have *1*: *gFreshAbs MOD ys y* (*Abs xs x X*) (*gAbs MOD xs x X* (*?h X*)) **by** *simp*
show *freshAbs ys y* (*Abs xs x X*)
using *assms hX Abs* ∗∗ **unfolding** *gFreshClsRev-def gFreshGAbsRev-def* **using** *1* **by** *fastforce*
qed
qed
}
thus *?thesis* **unfolding** *reflFreshAll-defs* **by** *blast*
qed

For fresh-swap models

theorem *wlsFSw-recAll-reflFreshAll*:
wlsFSw MOD ⟹ *gFreshClsRev MOD* ⟹ *reflFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
using *wlsFSw-recAll-termFSwMorph*
by (*auto simp*: *termFSwMorph-def intro*: *gFreshClsRev-recAll-reflFreshAll*)

For fresh-subst models

theorem *wlsFSb-recAll-reflFreshAll*:
wlsFSb MOD ⟹ *gFreshClsRev MOD* ⟹ *reflFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
using *wlsFSb-recAll-termFSbMorph*
by (*auto simp*: *termFSbMorph-def intro*: *gFreshClsRev-recAll-reflFreshAll*)

10.8.4 Criterion for the injectiveness of the recursive map

For fresh-swap models

theorem *wlsFSw-recAll-isInjAll*:
assumes ∗: *wlsFSw MOD gAbsRenS MOD* **and** ∗∗: *gConsInj MOD*
shows *isInjAll* (*rec MOD*) (*recAbs MOD*)
proof−
let *?h* = *rec MOD* **let** *?hA* = *recAbs MOD*
have *1*: *termFSwMorph ?h ?hA MOD* **using** ∗ *wlsFSw-recAll-termFSwMorph* **by** *auto*
hence *pWls*: *presWlsAll ?h ?hA MOD*

and *pCons*: *presCons ?h ?hA MOD*
and *pFresh*: *presFreshAll ?h ?hA MOD*
and *pSwap*: *presSwapAll ?h ?hA MOD* **unfolding** *termFSwMorph-def* **by** *auto*
hence *pWlsInps*[*simp*]:
$\bigwedge$ *delta inp. wlsInp delta inp* $\Longrightarrow$ *gWlsInp MOD delta* (*lift ?h inp*)
$\bigwedge$ *delta binp. wlsBinp delta binp* $\Longrightarrow$ *gWlsBinp MOD delta* (*lift ?hA binp*)
using *presWls-wlsInp presWls-wlsBinp* **unfolding** *presWlsAll-def* **by** *auto*
{fix *s X us s′ A*
have
(*wls s X* $\longrightarrow$ ($\forall$ *Y*. *wls s Y* $\wedge$ *rec MOD X* = *rec MOD Y* $\longrightarrow$ *X* = *Y*)) $\wedge$
(*wlsAbs* (*us*,*s′*) *A* $\longrightarrow$ ($\forall$ *B*. *wlsAbs* (*us*,*s′*) *B* $\wedge$ *recAbs MOD A* = *recAbs MOD B* $\longrightarrow$ *A* = *B*))
proof (*induction rule*: *wls-induct*)
case (*Var xs x*)
show *?case* **proof** *clarify*
fix *Y*
assume *eq*: *rec MOD* (*Var xs x*) = *rec MOD Y* **and** *Y*: *wls* (*asSort xs*) *Y*
thus *Var xs x* = *Y*
proof−
{fix *ys y* **assume** *Y-def*: *Y* = *Var ys y* **and** *asSort ys* = *asSort xs*
hence *ys-def*: *ys* = *xs* **by** *simp*
have *rec-y-def*: *rec MOD* (*Var ys y*) = *gVar MOD ys y*
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq* ∗∗ *1* **unfolding** *Y-def rec-y-def gConsInj-def gVarInj-def*
unfolding *ys-def* **by** (*simp add*: *termFSwMorph-defs*)
}
moreover
{fix *delta1 inp1 binp1* **assume** *inp1s*: *wlsInp delta1 inp1 wlsBinp delta1 binp1*
and *Y-def*: *Y* = *Op delta1 inp1 binp1* **and** *st*: *stOf delta1* = *asSort xs*
hence *rec-Op-def*:
rec MOD (*Op delta1 inp1 binp1*) =
gOp MOD delta1 inp1 (*lift ?h inp1*) *binp1* (*lift ?hA binp1*)
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq* ∗∗ **unfolding** *Y-def rec-Op-def gConsInj-def gVarGOpInj-def*
using *inp1s st 1* **by** (*simp add*: *termFSwMorph-defs*)
}
ultimately show *?thesis* **using** *wls-nchotomy*[*of asSort xs Y*] *Y* **by** *blast*
qed
qed
next
case (*Op delta inp binp*)
show *?case* **proof** *clarify*
fix *Y* **assume** *Y*: *wls* (*stOf delta*) *Y*
and *rec MOD* (*Op delta inp binp*) = *rec MOD Y*
hence *eq*: *gOp MOD delta inp* (*lift ?h inp*) *binp* (*lift ?hA binp*) = *?h Y*
using *1 Op* **by** (*simp add*: *termFSwMorph-defs*)

show *Op delta inp binp* = *Y*
proof−
{**fix** *ys y* **assume** *Y-def*: *Y* = *Var ys y* **and** *st*: *asSort ys* = *stOf delta*
have *rec-y-def*: *rec MOD* (*Var ys y*) = *gVar MOD ys y*
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq*[*THEN sym*] ∗∗ **unfolding** *Y-def rec-y-def gConsInj-def gVar-GOpInj-def*
using *Op st* **by** *simp*
}
moreover
{**fix** *delta1 inp1 binp1* **assume** *inp1s*: *wlsInp delta1 inp1 wlsBinp delta1 binp1*
and *Y-def*: *Y* = *Op delta1 inp1 binp1* **and** *st*: *stOf delta1* = *stOf delta*
hence *rec-Op-def*:
rec MOD (*Op delta1 inp1 binp1*) =
gOp MOD delta1 inp1 (*lift ?h inp1*) *binp1* (*lift ?hA binp1*)
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *0*: *delta* = *delta1* ∧ *lift ?h inp* = *lift ?h inp1* ∧ *lift ?hA binp* = *lift ?hA binp1*
using *eq* ∗∗ **unfolding** *Y-def rec-Op-def gConsInj-def gOpInj-def*
using *Op inp1s st* **apply** *clarify*
apply(*erule allE*[*of - delta*]) **apply**(*erule allE*[*of - delta1*]) **by** *force*
hence *delta1-def*: *delta1* = *delta* **by** *simp*
have *1*: *inp* = *inp1*
proof(*rule ext*)
fix *i*
show *inp i* = *inp1 i*
proof(*cases inp i*)
case *None*
hence *lift ?h inp i* = *None* **by**(*simp add*: *lift-None*)
hence *lift ?h inp1 i* = *None* **using** *0* **by** *simp*
thus *?thesis* **unfolding** *None* **by**(*simp add*: *lift-None*)
next
case (*Some X*)
hence *lift ?h inp i* = *Some* (*?h X*) **unfolding** *lift-def* **by** *simp*
hence *lift ?h inp1 i* = *Some* (*?h X*) **using** *0* **by** *simp*
then obtain *Y* **where** *inp1-i*: *inp1 i* = *Some Y* **and** *hXY*: *?h X* = *?h Y*
unfolding *lift-def* **by**(*cases inp1 i*) *auto*
then obtain *s* **where** *ar-i*: *arOf delta i* = *Some s*
using *inp1s* **unfolding** *delta1-def wlsInp-iff sameDom-def*
by (*cases arOf delta i*) *auto*
hence *Y*: *wls s Y*
using *inp1s inp1-i* **unfolding** *delta1-def wlsInp-iff liftAll2-def* **by** *auto*
thus *?thesis*
unfolding *Some inp1-i* **using** *ar-i Some hXY Op.IH* **unfolding** *liftAll2-def* **by** *auto*
qed

qed
have *2*: *binp* = *binp1*
proof(*rule ext*)
fix *i*
show *binp i* = *binp1 i*
proof(*cases binp i*)
case *None*
hence *lift ?hA binp i* = *None* **by**(*simp add*: *lift-None*)
hence *lift ?hA binp1 i* = *None* **using** *0* **by** *simp*
thus *?thesis* **unfolding** *None* **by** (*simp add*: *lift-None*)
next
case (*Some A*)
hence *lift ?hA binp i* = *Some* (*?hA A*) **unfolding** *lift-def* **by** *simp*
hence *lift ?hA binp1 i* = *Some* (*?hA A*) **using** *0* **by** *simp*
then obtain *B* **where** *binp1-i*: *binp1 i* = *Some B* **and** *hAB*: *?hA A* = *?hA B*
unfolding *lift-def* **by** (*cases binp1 i*) *auto*
then obtain *us s* **where** *bar-i*: *barOf delta i* = *Some* (*us*,*s*)
using *inp1s* **unfolding** *delta1-def wlsBinp-iff sameDom-def*
by(*cases barOf delta i*) *auto*
hence *B*: *wlsAbs* (*us*,*s*) *B*
using *inp1s binp1-i* **unfolding** *delta1-def wlsBinp-iff liftAll2-def* **by** *auto*
thus *?thesis* **unfolding** *Some binp1-i*
using *bar-i Some hAB Op.IH* **unfolding** *liftAll2-def* **by** *fastforce*
qed
qed
have *?thesis* **unfolding** *Y-def delta1-def 1 2* **by** *simp*
}
ultimately show *?thesis* **using** *wls-nchotomy*[*of stOf delta Y*] *Y* **by** *blast*
qed
qed
next
case (*Abs s xs x X*)
show *?case* **proof** *clarify*
fix *B*
assume *B*: *wlsAbs* (*xs*,*s*) *B* **and** *recAbs MOD* (*Abs xs x X*) = *recAbs MOD B*
hence *eq*: *gAbs MOD xs x X* (*rec MOD X*) = *?hA B* **using** *1 Abs* **by** (*simp add*: *termFSwMorph-defs*)
hence *hX*: *gWls MOD s* (*?h X*) **using** *pWls Abs* **unfolding** *presWlsAll-defs* **by** *simp*
show *Abs xs x X* = *B*
proof−
let *?P* = *ParS*
(λ *xs'*. [])
(λ *s'*. *if s'* = *s then* [*X*] *else* [])
(λ *us-s*. [])
[]

have *P*: *wlsPar ?P* **using** *Abs* **unfolding** *wlsPar-def* **by** *simp*
{**fix** *y Y* **assume** *Y*: *wls s Y* **and** *B-def*: *B* = *Abs xs y Y*
hence *hY*: *gWls MOD s* (*?h Y*) **using** *pWls* **unfolding** *presWlsAll-defs* **by** *simp*
let *?Xsw* = *X* #[*y* ∧ *x*]-*xs* **let** *?hXsw* = *gSwap MOD xs y x X* (*?h X*)
have *hXsw*: *gWls MOD s ?hXsw*
using *Abs hX* **using** ∗ **unfolding** *wlsFSw-def gSwapAllPresGWlsAll-defs* **by** *simp*
assume ∀ *s*. ∀ *Y* ∈ *termsOfS ?P s*. *fresh xs y Y*
hence *y-fresh*: *fresh xs y X* **by** *simp*
hence *gFresh MOD xs y X* (*?h X*)
using *Abs pFresh* **unfolding** *presFreshAll-defs* **by** *simp*
hence *gAbs MOD xs y* (*?Xsw*) *?hXsw* = *gAbs MOD xs x X* (*?h X*)
using *Abs hX y-fresh* ∗ **unfolding** *gAbsRenS-def* **by** *fastforce*
also have ... = *?hA B* **using** *eq* **.**
also have *recAbs MOD B* = *gAbs MOD xs y Y* (*?h Y*)
unfolding *B-def* **using** *pCons Abs Y* **unfolding** *presCons-defs* **by** *blast*
finally have *gAbs MOD xs y ?Xsw ?hXsw* = *gAbs MOD xs y Y* (*?h Y*) **.**
hence *?hXsw* = *?h Y*
using ∗∗ *Abs hX hXsw Y hY* **unfolding** *gConsInj-def gAbsInj-def*
apply *clarify* **apply**(*erule allE*[*of* - *xs*]) **apply**(*erule allE*[*of* - *s*])
apply(*erule allE*[*of* - *y*]) **apply**(*erule allE*[*of* - *?Xsw*]) **by** *fastforce*
moreover have *?hXsw* = *?h ?Xsw*
using *Abs pSwap* **unfolding** *presSwapAll-defs* **by** *simp*
ultimately have *?h ?Xsw* = *?h Y* **by** *simp*
moreover have (*X*,*?Xsw*) ∈ *swapped* **using** *swap-swapped* **.**
ultimately have *Y-def*: *Y* = *?Xsw* **using** *Y Abs.IH* **by** *auto*
have *?thesis* **unfolding** *B-def Y-def*
using *Abs y-fresh* **by** *simp*
}
thus *?thesis* **using** *B P wlsAbs-fresh-nchotomy*[*of xs s B*] **by** *blast*
qed
qed
qed
}
thus *?thesis* **unfolding** *isInjAll-defs* **by** *blast*
qed

For fresh-subst models

theorem *wlsFSb-recAll-isInjAll*:
assumes ∗: *wlsFSb MOD* **and** ∗∗: *gConsInj MOD*
shows *isInjAll* (*rec MOD*) (*recAbs MOD*)
proof−
let *?h* = *rec MOD* **let** *?hA* = *recAbs MOD*
have *1*: *termFSbMorph ?h ?hA MOD* **using** ∗ *wlsFSb-recAll-termFSbMorph* **by** *auto*
hence *pWls*: *presWlsAll ?h ?hA MOD*
and *pCons*: *presCons ?h ?hA MOD*
and *pFresh*: *presFreshAll ?h ?hA MOD*

and *pSubst*: *presSubstAll ?h ?hA MOD* **unfolding** *termFSbMorph-def* **by** *auto*
hence *pWlsInps*[*simp*]:
$\bigwedge$ *delta inp. wlsInp delta inp* $\Longrightarrow$ *gWlsInp MOD delta* (*lift ?h inp*)
$\bigwedge$ *delta binp. wlsBinp delta binp* $\Longrightarrow$ *gWlsBinp MOD delta* (*lift ?hA binp*)
using *presWls-wlsInp presWls-wlsBinp* **unfolding** *presWlsAll-def* **by** *auto*
{**fix** *s X us s' A*
have
(*wls s X* $\longrightarrow$ ($\forall$ *Y*. *wls s Y* $\wedge$ *rec MOD X* = *rec MOD Y* $\longrightarrow$ *X* = *Y*)) $\wedge$
(*wlsAbs* (*us*,*s'*) *A* $\longrightarrow$ ($\forall$ *B*. *wlsAbs* (*us*,*s'*) *B* $\wedge$ *recAbs MOD A* = *recAbs MOD B* $\longrightarrow$ *A* = *B*))
proof(*induction rule*: *wls-induct*)
case (*Var xs x*)
show *?case* **proof** *clarify*
fix *Y*
assume *rec MOD* (*Var xs x*) = *rec MOD Y* **and** *Y*: *wls* (*asSort xs*) *Y*
hence *eq*: *gVar MOD xs x* = *rec MOD Y* **using** *1* **by** (*simp add*: *termFSbMorph-defs*)
show *Var xs x* = *Y*
proof−
{**fix** *ys y* **assume** *Y-def*: *Y* = *Var ys y* **and** *asSort ys* = *asSort xs*
hence *ys-def*: *ys* = *xs* **by** *simp*
have *rec-y-def*: *rec MOD* (*Var ys y*) = *gVar MOD ys y*
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq* ** **unfolding** *Y-def rec-y-def gConsInj-def gVarInj-def*
unfolding *ys-def* **by** *simp*
}
moreover
{**fix** *delta1 inp1 binp1* **assume** *inp1s*: *wlsInp delta1 inp1 wlsBinp delta1 binp1*
and *Y-def*: *Y* = *Op delta1 inp1 binp1* **and** *st*: *stOf delta1* = *asSort xs*
hence *rec-Op-def*:
rec MOD (*Op delta1 inp1 binp1*) =
gOp MOD delta1 inp1 (*lift ?h inp1*) *binp1* (*lift ?hA binp1*)
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq* ** **unfolding** *Y-def rec-Op-def gConsInj-def gVarGOpInj-def*
using *inp1s st* **by** *simp*
}
ultimately show *?thesis* **using** *wls-nchotomy*[*of asSort xs Y*] *Y* **by** *blast*
qed
qed
next
case (*Op delta inp binp*)
show *?case* **proof** *clarify*
fix *Y*
assume *rec MOD* (*Op delta inp binp*) = *rec MOD Y* **and** *Y*: *wls* (*stOf delta*) *Y*
hence *eq*: *gOp MOD delta inp* (*lift ?h inp*) *binp* (*lift ?hA binp*) = *?h Y*

using *Op 1* **by** (*simp add*: *termFSbMorph-defs*)
show *Op delta inp binp* = *Y*
proof−
{**fix** *ys y* **assume** *Y-def*: *Y* = *Var ys y* **and** *st*: *asSort ys* = *stOf delta*
have *rec-y-def*: *rec MOD* (*Var ys y*) = *gVar MOD ys y*
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *?thesis*
using *eq*[*THEN sym*] ∗∗ **unfolding** *Y-def rec-y-def gConsInj-def gVar-GOpInj-def*
using *Op st* **by** *simp*
}
moreover
{**fix** *delta1 inp1 binp1* **assume** *inp1s*: *wlsInp delta1 inp1 wlsBinp delta1 binp1*
and *Y-def*: *Y* = *Op delta1 inp1 binp1* **and** *st*: *stOf delta1* = *stOf delta*
hence *rec-Op-def*:
rec MOD (*Op delta1 inp1 binp1*) =
gOp MOD delta1 inp1 (*lift ?h inp1*) *binp1* (*lift ?hA binp1*)
using *pCons* **unfolding** *presCons-defs* **by** *simp*
have *0*: *delta* = *delta1* ∧ *lift ?h inp* = *lift ?h inp1* ∧ *lift ?hA binp* = *lift ?hA binp1*
using *eq* ∗∗ **unfolding** *Y-def rec-Op-def gConsInj-def gOpInj-def*
using *Op inp1s st* **apply** *clarify*
apply(*erule allE*[*of - delta*]) **apply**(*erule allE*[*of - delta1*]) **by** *force*
hence *delta1-def*: *delta1* = *delta* **by** *simp*
have *1*: *inp* = *inp1*
proof(*rule ext*)
fix *i*
show *inp i* = *inp1 i*
proof(*cases inp i*)
case *None*
hence *lift ?h inp i* = *None* **by**(*simp add*: *lift-None*)
hence *lift ?h inp1 i* = *None* **using** *0* **by** *simp*
thus *?thesis* **unfolding** *None* **by**(*simp add*: *lift-None*)
next
case (*Some X*)
hence *lift ?h inp i* = *Some* (*?h X*) **unfolding** *lift-def* **by** *simp*
hence *lift ?h inp1 i* = *Some* (*?h X*) **using** *0* **by** *simp*
then obtain *Y* **where** *inp1-i*: *inp1 i* = *Some Y* **and** *hXY*: *?h X* = *?h Y*
unfolding *lift-def* **by** (*cases inp1 i*) *auto*
then obtain *s* **where** *ar-i*: *arOf delta i* = *Some s*
using *inp1s* **unfolding** *delta1-def wlsInp-iff sameDom-def*
by (*cases arOf delta i*) *auto*
hence *Y*: *wls s Y*
using *inp1s inp1-i* **unfolding** *delta1-def wlsInp-iff liftAll2-def* **by** *auto*
thus *?thesis* **unfolding** *Some inp1-i*
using *ar-i Some hXY Op.IH* **unfolding** *liftAll2-def* **by** *auto*
qed

qed
have *2*: *binp* = *binp1*
proof(*rule ext*)
fix *i*
show *binp i* = *binp1 i*
proof(*cases binp i*)
case *None*
hence *lift ?hA binp i* = *None* **by**(*simp add*: *lift-None*)
hence *lift ?hA binp1 i* = *None* **using** *0* **by** *simp*
thus *?thesis* **unfolding** *None* **by**(*simp add*: *lift-None*)
next
case (*Some A*)
hence *lift ?hA binp i* = *Some* (*?hA A*) **unfolding** *lift-def* **by** *simp*
hence *lift ?hA binp1 i* = *Some* (*?hA A*) **using** *0* **by** *simp*
then obtain *B* **where** *binp1-i*: *binp1 i* = *Some B* **and** *hAB*: *?hA A* = *?hA B*
unfolding *lift-def* **by**(*cases binp1 i*, *auto*)
then obtain *us s* **where** *bar-i*: *barOf delta i* = *Some* (*us*,*s*)
using *inp1s* **unfolding** *delta1-def wlsBinp-iff sameDom-def*
by(*cases barOf delta i*) *auto*
hence *B*: *wlsAbs* (*us*,*s*) *B*
using *inp1s binp1-i* **unfolding** *delta1-def wlsBinp-iff liftAll2-def* **by** *auto*
thus *?thesis* **unfolding** *Some binp1-i*
using *bar-i Some hAB Op.IH*
unfolding *liftAll2-def* **by** *fastforce*
qed
qed
have *?thesis* **unfolding** *Y-def delta1-def 1 2* **by** *simp*
}
ultimately show *?thesis* **using** *wls-nchotomy*[*of stOf delta Y*] *Y* **by** *blast*
qed
qed
next
case (*Abs s xs x X*)
show *?case* **proof** *clarify*
fix *B*
assume *B*: *wlsAbs* (*xs*,*s*) *B* **and** *recAbs MOD* (*Abs xs x X*) = *recAbs MOD B*
hence *eq*: *gAbs MOD xs x X* (*rec MOD X*) = *?hA B*
using *1 Abs* **by** (*simp add*: *termFSbMorph-defs*)
hence *hX*: *gWls MOD s* (*?h X*) **using** *pWls Abs* **unfolding** *presWlsAll-defs* **by** *simp*
show *Abs xs x X* = *B*
proof−
let *?P* = *ParS*
(λ *xs*′. [])
(λ *s*′. *if s*′ = *s then* [*X*] *else* [])
(λ *us-s*. [])

[]
have *P*: *wlsPar ?P* **using** *Abs* **unfolding** *wlsPar-def* **by** *simp*
{**fix** *y Y* **assume** *Y*: *wls s Y* **and** *B-def*: *B = Abs xs y Y*
hence *hY*: *gWls MOD s* (*?h Y*) **using** *pWls* **unfolding** *presWlsAll-defs* **by** *simp*
let *?Xsb = X #[y // x]-xs*
let *?hXsb = gSubst MOD xs* (*Var xs y*) (*gVar MOD xs y*) *x X* (*?h X*)
have *1*: *wls* (*asSort xs*) (*Var xs y*) ∧ *gWls MOD* (*asSort xs*) (*gVar MOD xs y*)
using ∗ **unfolding** *wlsFSb-def gConsPresGWls-defs* **by** *simp*
hence *hXsb*: *gWls MOD s ?hXsb*
using *Abs hX* **using** ∗ **unfolding** *wlsFSb-def gSubstAllPresGWlsAll-defs* **by** *simp*
assume ∀ *s*. ∀ *Y* ∈ *termsOfS ?P s*. *fresh xs y Y*
hence *y-fresh*: *fresh xs y X* **by** *simp*
hence *gFresh MOD xs y X* (*?h X*)
using *Abs pFresh* **unfolding** *presFreshAll-defs* **by** *simp*
hence *gAbs MOD xs y* (*?Xsb*) *?hXsb = gAbs MOD xs x X* (*?h X*)
using *Abs hX y-fresh* ∗ **unfolding** *wlsFSb-def gAbsRen-def* **by** *fastforce*
also have ... = *?hA B* **using** *eq* **.**
also have ... = *gAbs MOD xs y Y* (*?h Y*)
unfolding *B-def* **using** *pCons Abs Y* **unfolding** *presCons-defs* **by** *blast*
finally have
gAbs MOD xs y ?Xsb ?hXsb = gAbs MOD xs y Y (*?h Y*) **.**
hence *?hXsb = ?h Y*
using ∗∗ *Abs hX hXsb Y hY* **unfolding** *gConsInj-def gAbsInj-def*
apply *clarify* **apply**(*erule allE*[*of - xs*]) **apply**(*erule allE*[*of - s*])
apply(*erule allE*[*of - y*]) **apply**(*erule allE*[*of - ?Xsb*]) **by** *fastforce*
moreover have *?hXsb = ?h ?Xsb*
using *Abs pSubst 1 pCons* **unfolding** *presSubstAll-defs vsubst-def presCons-defs* **by** *simp*
ultimately have *?h ?Xsb = ?h Y* **by** *simp*
hence *Y-def*: *Y = ?Xsb* **using** *Y Abs.IH* **by** (*fastforce simp add*: *termFSbMorph-defs*)
have *?thesis* **unfolding** *B-def Y-def*
using *Abs y-fresh* **by** *simp*
}
thus *?thesis* **using** *B P wlsAbs-fresh-nchotomy*[*of xs s B*] **by** *blast*
qed
qed
qed
}
thus *?thesis* **unfolding** *isInjAll-defs* **by** *blast*
qed

10.8.5 Criterion for the surjectiveness of the recursive map

First an auxiliary fact, independent of the type of model:

lemma *gInduct-gConsIndif-recAll-isSurjAll*:

assumes *pWls*: *presWlsAll* (*rec MOD*) (*recAbs MOD*) *MOD*
and *pCons*: *presCons* (*rec MOD*) (*recAbs MOD*) *MOD*
and *gConsIndif MOD* **and** ∗: *gInduct MOD*
shows *isSurjAll* (*rec MOD*) (*recAbs MOD*) *MOD*
proof−
  let *?h* = *rec MOD*   **let** *?hA* = *recAbs MOD*
  {**fix** *s X us s′ A*
   from ∗ **have**
   (*gWls MOD s X* ⟶ (∃ *X′*. *wls s X′* ∧ *rec MOD X′* = *X*)) ∧
    (*gWlsAbs MOD* (*us*,*s′*) *A* ⟶ (∃ *A′*. *wlsAbs* (*us*,*s′*) *A′* ∧ *recAbs MOD A′* = *A*))
   proof (*elim gInduct-elim*, *safe*)
     fix *xs x*
     show ∃ *X′*. *wls* (*asSort xs*) *X′* ∧ *rec MOD X′* = *gVar MOD xs x*
     using *pWls pCons*
     by (*auto simp*: *presWlsAll-defs presCons-defs intro*: *exI*[*of* - *Var xs x*])
   next
     fix *delta inp′ inp binp′ binp*
     let *?ar* = *arOf delta*   **let** *?bar* = *barOf delta* **let** *?st* = *stOf delta*
     assume *inp′*: *wlsInp delta inp′* **and** *binp′*: *wlsBinp delta binp′*
     and *inp*: *gWlsInp MOD delta inp* **and** *binp*: *gWlsBinp MOD delta binp*
     and *IH*: *liftAll2* (λ*s X*. ∃ *X′*. *wls s X′* ∧ *?h X′* = *X*) *?ar inp*
     and *BIH*: *liftAll2* (λ*us-s A*. ∃ *A′*. *wlsAbs us-s A′* ∧ *?hA A′* = *A*) *?bar binp*

     let *?phi* = λ *s X X′*. *wls s X′* ∧ *?h X′* = *X*
     obtain *inp1′* **where** *inp1′-def*:
     inp1′ =
      (λ *i*.
        case (*?ar i*, *inp i*) *of*
          (*None*, *None*) ⇒ *None*
        |(*Some s*, *Some X*) ⇒ *Some* (*SOME X′*. *?phi s X X′*)) **by** *blast*
     hence [*simp*]:
     ⋀ *i*. *?ar i* = *None* ∧ *inp i* = *None* ⟹ *inp1′ i* = *None*
     ⋀ *i s X*. *?ar i* = *Some s* ∧ *inp i* = *Some X* ⟹ *inp1′ i* = *Some* (*SOME X′*.
?phi s X X′)
     unfolding *inp1′-def* **by** *auto*
     have *inp1′*: *wlsInp delta inp1′*
     unfolding *wlsInp-iff* **proof** *safe*
       show *sameDom ?ar inp1′*
       unfolding *sameDom-def* **proof** *clarify*
         fix *i*
         have (*?ar i* = *None*) = (*inp i* = *None*)
         using *inp* **unfolding** *gWlsInp-def sameDom-def* **by** *simp*
         thus (*?ar i* = *None*) = (*inp1′ i* = *None*)
         unfolding *inp1′-def* **by** *auto*
       qed
     next
       show *liftAll2 wls ?ar inp1′*
       unfolding *liftAll2-def* **proof** *auto*
         fix *i s X1′*

assume *ari*: *?ar i = Some s* **and** *inp1'i*: *inp1' i = Some X1'*
have *sameDom inp ?ar*
using *inp* **unfolding** *gWlsInp-def* **using** *sameDom-sym* **by** *blast*
then obtain *X* **where** *inpi*: *inp i = Some X*
using *ari* **unfolding** *sameDom-def* **by**(*cases inp i*) *auto*
hence *X1'-def*: *X1' = (SOME X1'. ?phi s X X1')*
using *ari inp1'i* **unfolding** *inp1'-def* **by** *simp*
obtain *X'* **where** *X'*: *?phi s X X'*
using *inpi ari IH* **unfolding** *liftAll2-def* **by** *blast*
hence *?phi s X X1'*
unfolding *X1'-def* **by**(*rule someI*[*of ?phi s X*])
thus *wls s X1'* **by** *simp*
qed
qed(*insert binp' wlsBinp.cases*, *blast*)

have *lift-inp1'*: *lift ?h inp1' = inp*
proof(*rule ext*)
fix *i* **let** *?linp1' = lift ?h inp1'*
show *?linp1' i = inp i*
proof(*cases inp i*)
case *None*
hence *?ar i = None* **using** *inp* **unfolding** *gWlsInp-def sameDom-def* **by** *simp*
hence *inp1' i = None* **using** *None* **by** *simp*
thus *lift* (*rec MOD*) *inp1' i = inp i* **using** *None* **by** (*auto simp*: *lift-def*)
next
case (*Some X*)
then obtain *s* **where** *ari*: *?ar i = Some s*
using *inp* **unfolding** *gWlsInp-def sameDom-def* **by**(*cases ?ar i*) *auto*
let *?X1' = SOME X1'. ?phi s X X1'*
have *inp1'i*: *inp1' i = Some ?X1'* **using** *ari Some* **by** *simp*
hence *linp1'i*: *?linp1' i = Some* (*?h ?X1'*) **unfolding** *lift-def* **by** *simp*
obtain *X'* **where** *X'*: *?phi s X X'*
using *Some ari IH* **unfolding** *liftAll2-def* **by** *blast*
hence *?phi s X ?X1'* **by**(*rule someI*[*of ?phi s X*])
thus *lift* (*rec MOD*) *inp1' i = inp i* **using** *Some linp1'i* **by** (*auto simp*: *lift-def*)
qed
qed

let *?bphi = λ (us,s) A A'. wlsAbs (us,s) A' ∧ ?hA A' = A*
obtain *binp1'* **where** *binp1'-def*:
binp1' =
(*λ i.*
case (*?bar i*, *binp i*) *of*
(*None*, *None*) ⇒ *None*
|(*Some* (*us,s*), *Some A*) ⇒ *Some* (*SOME A'. ?bphi* (*us,s*) *A A'*)) **by** *blast*
hence [*simp*]:
⋀ *i. ?bar i = None ∧ binp i = None ⟹ binp1' i = None*

and ∗:
$\bigwedge$ *i us s A. ?bar i = Some* (*us*,*s*) ∧ *binp i = Some A* ⟹
binp1′ i = Some (*SOME A′. ?bphi* (*us*,*s*) *A A′*)
unfolding *binp1′-def* **by** *auto*
have *binp1′*: *wlsBinp delta binp1′*
unfolding *wlsBinp-iff* **proof** *safe*
show *sameDom ?bar binp1′*
unfolding *sameDom-def* **proof** *clarify*
fix *i*
have (*?bar i = None*) = (*binp i = None*)
using *binp* **unfolding** *gWlsBinp-def sameDom-def* **by** *simp*
thus (*?bar i = None*) = (*binp1′ i = None*)
unfolding *binp1′-def* **by** *auto*
qed
next
show *liftAll2 wlsAbs ?bar binp1′*
unfolding *liftAll2-def* **proof** *auto*
fix *i us s A1′*
assume *bari*: *?bar i = Some* (*us*,*s*) **and** *binp1′i*: *binp1′ i = Some A1′*
have *sameDom binp ?bar*
using *binp* **unfolding** *gWlsBinp-def* **using** *sameDom-sym* **by** *blast*
then obtain *A* **where** *binpi*: *binp i = Some A*
using *bari* **unfolding** *sameDom-def* **by**(*cases binp i*, *auto*)
hence *A1′-def*: *A1′* = (*SOME A1′. ?bphi* (*us*,*s*) *A A1′*)
using *bari binp1′i* **unfolding** *binp1′-def* **by** *simp*
obtain *A′* **where** *A′*: *?bphi* (*us*,*s*) *A A′*
using *binpi bari BIH* **unfolding** *liftAll2-def* **by** *fastforce*
hence *?bphi* (*us*,*s*) *A A1′*
unfolding *A1′-def* **by**(*rule someI*[*of ?bphi* (*us*,*s*) *A*])
thus *wlsAbs* (*us*,*s*) *A1′* **by** *simp*
qed
qed(*insert binp′ wlsBinp.cases*, *blast*)

have *lift-binp1′*: *lift ?hA binp1′ = binp*
proof(*rule ext*)
fix *i* **let** *?lbinp1′ = lift ?hA binp1′*
show *?lbinp1′ i = binp i*
proof(*cases binp i*)
case *None*
hence *?bar i = None* **using** *binp* **unfolding** *gWlsBinp-def sameDom-def* **by** *simp*
hence *binp1′ i = None* **using** *None* **by** *simp*
thus *lift* (*recAbs MOD*) *binp1′ i = binp i* **using** *None* **by** (*simp add*: *lift-def*)
next
case (*Some A*)
then obtain *us s* **where** *bari*: *?bar i = Some* (*us*,*s*)
using *binp* **unfolding** *gWlsBinp-def sameDom-def* **by**(*cases ?bar i*, *auto*)
let *?A1′ = SOME A1′. ?bphi* (*us*,*s*) *A A1′*

have *binp1′i*: *binp1′ i* = *Some ?A1′* **using** *bari Some* ∗[*of i us s A*] **by** *simp*
hence *lbinp1′i*: *?lbinp1′ i* = *Some* (*?hA ?A1′*) **unfolding** *lift-def* **by** *simp*
obtain *A′* **where** *A′*: *?bphi* (*us*,*s*) *A A′*
using *Some bari BIH* **unfolding** *liftAll2-def* **by** *fastforce*
hence *?bphi* (*us*,*s*) *A ?A1′* **by**(*rule someI*[*of ?bphi* (*us*,*s*) *A*])
thus *lift* (*recAbs MOD*) *binp1′ i* = *binp i* **using** *Some lbinp1′i* **by** *simp*
qed
qed

let *?X′* = *Op delta inp1′ binp1′*
have *X′*: *wls ?st ?X′* **using** *inp1′ binp1′* **by** *simp*
have *?h ?X′* = *gOp MOD delta inp1′ inp binp1′ binp*
using *inp1′ binp1′ pCons lift-inp1′ lift-binp1′*
unfolding *presCons-defs* **by** *simp*
hence *?h ?X′* = *gOp MOD delta inp′ inp binp′ binp*
using *inp′ inp1′ inp binp′ binp1′ binp assms*
unfolding *gConsIndif-defs* **by** *metis*
thus ∃ *X′*. *wls* (*stOf delta*) *X′* ∧ *?h X′* = *gOp MOD delta inp′ inp binp′ binp*
using *X′* **by** *blast*
next
fix *xs s x X′ X1′*
assume *xs-s*: *isInBar* (*xs*,*s*) **and** *X′*: *wls s X′* **and**
hX1′: *gWls MOD s* (*?h X1′*) **and** *X1′*: *wls s X1′*
thus ∃ *A′*. *wlsAbs* (*xs*,*s*) *A′* ∧ *?hA A′* = *gAbs MOD xs x X′* (*?h X1′*)
apply(*intro exI*[*of - Abs xs x X1′*])
using *pCons* **unfolding** *presCons-def presAbs-def* **apply** *safe*
apply(*elim allE*[*of - xs*]) **apply**(*elim allE*[*of - x*]) **apply**(*elim allE*[*of - s*])
apply *simp-all*
using *assms* **unfolding** *gConsIndif-defs* **by** *blast*
qed
}
thus *?thesis* **unfolding** *isSurjAll-defs* **by** *blast*
qed

For fresh-swap models

theorem *wlsFSw-recAll-isSurjAll*:
wlsFSw MOD ⟹ *gConsIndif MOD* ⟹ *gInduct MOD*
⟹ *isSurjAll* (*rec MOD*) (*recAbs MOD*) *MOD*
using *wlsFSw-recAll-termFSwMorph*
by (*auto simp*: *termFSwMorph-def intro*: *gInduct-gConsIndif-recAll-isSurjAll*)

For fresh-subst models

theorem *wlsFSb-recAll-isSurjAll*:
wlsFSb MOD ⟹ *gConsIndif MOD* ⟹ *gInduct MOD*
⟹ *isSurjAll* (*rec MOD*) (*recAbs MOD*) *MOD*
using *wlsFSb-recAll-termFSbMorph*
by (*auto simp*: *termFSbMorph-def intro*: *gInduct-gConsIndif-recAll-isSurjAll*)

lemmas *recursion-simps* =
fromMOD-simps ipresCons-fromMOD-fst-all-simps fromIMor-simps

declare *recursion-simps* [*simp del*]

end

end