A General Theory of Syntax with Bindings

Lorenzo Gheri and Andrei Popescu

May 26, 2024

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*} ∪
{(*absIn A*, *termIn*(*qOp delta inp binp*))| *A delta inp binp i. binp i* = *Some A*} ∪
{(*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* ∧ *phiAbs A*
⟨*proof*⟩

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

lemma *qTermLessPlus-wf*: *wf* (*qTermLess* ^+)
⟨*proof*⟩

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 :: $('index,'bindex,'varSort,'var,'opSym)qTerm \Rightarrow ('index,'bindex)tree$
and
qSkelAbs :: $('index,'bindex,'varSort,'var,'opSym)qAbs \Rightarrow ('index,'bindex)tree$
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* (λi. *Some*(*qSkel X*)) (λi. *None*)
⟨*proof*⟩
termination ⟨*proof*⟩

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: $\bigwedge$ *X Y*. (*X*,*Y*) ∈ *rel* ⟹ *qSkel Y* = *qSkel X* **and**
Var: $\bigwedge$ *xs x*. *phi* (*qVar xs x*) **and**
Op: $\bigwedge$ *delta inp binp*. ⟦*liftAll phi inp*; *liftAll phiAbs binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: $\bigwedge$ *xs x X*. ($\bigwedge$ *Y*. (*X*,*Y*) ∈ *rel* ⟹ *phi Y*) ⟹ *phiAbs* (*qAbs xs x X*)
shows *phi X* ∧ *phiAbs A*
⟨*proof*⟩

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* ⇒
(*'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*} ∪
{(*absIn A*, *termIn*(*qOp delta inp binp*))| *A delta inp binp j*. *binp j* = *Some A*} ∪
{(*termIn Y*, *absIn* (*qAbs xs x X*))| *X Y xs x*. (*X*,*Y*) ∈ *rel*}

lemma *qTermLess-modulo-wf*:
fixes *rel*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm rel*
assumes $\bigwedge$ *X Y*. (*X*,*Y*) ∈ *rel* ⟹ *qSkel Y* = *qSkel X*
shows *wf* (*qTermLess-modulo rel*)
⟨*proof*⟩

1.3 Swap and substitution on variables

definition *sw* :: *'varSort* ⇒ *'var* ⇒ *'var* ⇒ *'varSort* ⇒ *'var* ⇒ *'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* ⇒ *'varSort* ⇒ *'var* ⇒ *'var* ⇒ *'varSort* ⇒ *'var*

```
(- @-[- ∧ -]′-- 200)
where (x @xs[y1 ∧ y2]-ys) == sw ys y1 y2 xs x

definition sb :: 'varSort ⇒ 'var ⇒ 'var ⇒ 'varSort ⇒ 'var ⇒ '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
⟨proof⟩

theorem sw-simps2[simp]: (x @xs[y ∧ x]-xs) = y
⟨proof⟩

theorem sw-simps3[simp]:
(zs ≠ xs ∨ x ∉ {z1,z2}) ⟹ (x @xs[z1 ∧ z2]-zs) = x
⟨proof⟩

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

theorem sw-ident[simp]: (x @xs[y ∧ y]-ys) = x
⟨proof⟩

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)
⟨proof⟩

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)
⟨proof⟩

theorem sw-involutive[simp]:
((z @zs[x ∧ y]-xs) @zs[x ∧ y]-xs) = z
⟨proof⟩

theorem sw-inj[simp]:
((z @zs[x ∧ y]-xs) = (z' @zs[x ∧ y]-xs)) = (z = z')
⟨proof⟩

lemma sw-preserves-mship[simp]:
assumes {y1,y2} ⊆ Var ys
```

shows $((x\ @xs[y1 \wedge y2]\text{-}ys) \in Var\ xs) = (x \in Var\ xs)$
⟨*proof*⟩

theorem *sw-sym*:
$(z\ @zs[x \wedge y]\text{-}xs) = (z\ @zs[y \wedge x]\text{-}xs)$
⟨*proof*⟩

theorem *sw-involutive2*[*simp*]:
$((z\ @zs[x \wedge y]\text{-}xs)\ @zs[y \wedge x]\text{-}xs) = z$
⟨*proof*⟩

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

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

theorem *sb-simps1*[*simp*]: $(x\ @xs[y\ /\ x]\text{-}xs) = y$
⟨*proof*⟩

theorem *sb-simps2*[*simp*]:
$(zs \neq xs \vee z2 \neq x) \Longrightarrow (x\ @xs[z1\ /\ z2]\text{-}zs) = x$
⟨*proof*⟩

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

theorem *sb-ident*[*simp*]: $(x\ @xs[y\ /\ y]\text{-}ys) = x$
⟨*proof*⟩

theorem *sb-compose1*:
$((z\ @zs[y1\ /\ x]\text{-}xs)\ @zs[y2\ /\ x]\text{-}xs) = (z\ @zs[(y1\ @xs[y2\ /\ x]\text{-}xs)\ /\ x]\text{-}xs)$
⟨*proof*⟩

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

theorem *sb-commute*:
assumes $zs \neq zs' \vee \{x,y\}\ Int\ \{x',y'\} = \{\}$
shows $((u\ @us[x\ /\ y]\text{-}zs)\ @us[x'\ /\ y']\text{-}zs') = ((u\ @us[x'\ /\ y']\text{-}zs')\ @us[x\ /\ y]\text{-}zs)$
⟨*proof*⟩

theorem *sb-idem*[*simp*]:
$((z\ @zs[x\ /\ y]\text{-}xs)\ @zs[x\ /\ y]\text{-}xs) = (z\ @zs[x\ /\ y]\text{-}xs)$
⟨*proof*⟩

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

theorem *sb-trans*:
$us \neq zs \vee u \neq y \Longrightarrow$
$((u\ @us[y\ /\ x]\text{-}zs)\ @us[z\ /\ y]\text{-}zs) = (u\ @us[z\ /\ x]\text{-}zs)$
$\langle proof \rangle$

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 =
$(case\ K\ of\ Inl(zs,x,y,X) \Rightarrow termIn\ X$
$|Inr(zs,x,y,A) \Rightarrow absIn\ A)$

lemma *qTermLess-ingoreFirst3-wf*:
wf(*inv-image qTermLess aux-qSwap-ignoreFirst3*)
$\langle proof \rangle$

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* ∧ *y*]-*zs*) (*qSwap zs x y X*)
⟨*proof*⟩
termination
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *qSwap-ident*[*simp*]: (*X* #[[*x* ∧ *x*]]-*zs*) = *X*
⟨*proof*⟩

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*⟩

corollary *qSwap-compose*:
((*X* #[[*x* ∧ *y*]]-*zs*) #[[*x'* ∧ *y'*]]-*zs'*) =
((*X* #[[*x'* ∧ *y'*]]-*zs'*) #[[(*x* @*zs*[*x'* ∧ *y'*]-*zs'*) ∧ (*y* @*zs*[*x'* ∧ *y'*]-*zs'*)]]-*zs*)
⟨*proof*⟩

lemma *qSwap-commute*:
assumes $zs \neq zs' \vee \{x,y\}$ *Int* $\{x',y'\} = \{\}$
shows $((X \ \#[[x \wedge y]]\text{-}zs) \ \#[[x' \wedge y']]\text{-}zs') = ((X \ \#[[x' \wedge y']]\text{-}zs') \ \#[[x \wedge y]]\text{-}zs)$
⟨*proof*⟩

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*⟩

corollary *qSwap-involutive*[*simp*]:
$((X \ \#[[x \wedge y]]\text{-}zs) \ \#[[x \wedge y]]\text{-}zs) = X$
⟨*proof*⟩

lemma *qSwapAll-sym*:
fixes X::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm* **and**
A::('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qAbs* **and** *zs x y*
shows $(X \ \#[[x \wedge y]]\text{-}zs) = (X \ \#[[y \wedge x]]\text{-}zs) \ \wedge$
$(A \ \$[[x \wedge y]]\text{-}zs) = (A \ \$[[y \wedge x]]\text{-}zs)$
⟨*proof*⟩

corollary *qSwap-sym*:
$(X \ \#[[x \wedge y]]\text{-}zs) = (X \ \#[[y \wedge x]]\text{-}zs)$
⟨*proof*⟩

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* $\longrightarrow (A \ \$[[z1 \wedge z2]]\text{-}zs) = A)$
⟨*proof*⟩

corollary *qAFresh-qSwap-id*[*simp*]:
⟦*qAFresh zs z1 X*; *qAFresh zs z2 X*⟧ $\Longrightarrow (X \ \#[[z1 \wedge z2]]\text{-}zs) = X$
⟨*proof*⟩

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* $\wedge$ *qAFresh zs z X* $\longrightarrow$
$((X \ \#[[y \wedge x]]\text{-}zs) \ \#[[z \wedge y]]\text{-}zs) = (X \ \#[[z \wedge x]]\text{-}zs)) \ \wedge$
(*qAFreshAbs zs y A* $\wedge$ *qAFreshAbs zs z A* $\longrightarrow$
$((A \ \$[[y \wedge x]]\text{-}zs) \ \$[[z \wedge y]]\text{-}zs) = (A \ \$[[z \wedge x]]\text{-}zs))$
⟨*proof*⟩

corollary *qAFresh-qSwap-compose*:

$[\![qAFresh\ zs\ y\ X;\ qAFresh\ zs\ z\ X]\!] \Longrightarrow$
$((X\ \#[[y \wedge x]]\text{-}zs)\ \#[[z \wedge y]]\text{-}zs) = (X\ \#[[z \wedge x]]\text{-}zs)$
$\langle proof \rangle$

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 $qSkel(X\ \#[[x \wedge y]]\text{-}zs) = qSkel\ X\ \wedge$
$qSkelAbs(A\ \$[[x \wedge y]]\text{-}zs) = qSkelAbs\ A$
$\langle proof \rangle$

corollary *qSkel-qSwap*: $qSkel(X\ \#[[x \wedge y]]\text{-}zs) = qSkel\ X$
$\langle proof \rangle$

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: $(X,X) \in qSwapped$
|
Trans: $[\![(X,Y) \in qSwapped;\ (Y,Z) \in qSwapped]\!] \Longrightarrow (X,Z) \in qSwapped$
|
Swap: $(X,Y) \in qSwapped \Longrightarrow (X,\ Y\ \#[[x \wedge y]]\text{-}zs) \in qSwapped$

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

lemma *qSwap-qSwapped*: $(X,\ X\ \#[[x \wedge y]]\text{-}zs)$: *qSwapped*
$\langle proof \rangle$

lemma *qSwapped-qSkel*:
$(X,Y) \in qSwapped \Longrightarrow qSkel\ Y = qSkel\ X$
$\langle proof \rangle$

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*⟧
    $\Longrightarrow$ *phi* (*qOp delta inp binp*) **and**
  Abs: $\bigwedge$ *xs x X.* ⟦$\bigwedge$ *Y.* $(X,Y) \in$ *qSwapped* $\Longrightarrow$ *phi Y*;
    $\bigwedge$ *Y. qSkel Y* = *qSkel X* $\Longrightarrow$ *phi Y*⟧
    $\Longrightarrow$ *phiAbs* (*qAbs xs x X*)
shows *phi X* $\wedge$ *phiAbs A*
  ⟨*proof*⟩

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*
⟨*proof*⟩

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* $\wedge$ *z2*]]-*zs*) #[[*y0* $\wedge$ *x* @*ys*[*z1* $\wedge$ *z2*]-*zs*]]-*ys*) =
    (((*X* #[[*y* $\wedge$ *x*]]-*ys*) #[[*y0* $\wedge$ *y*]]-*ys*) #[[*z1* $\wedge$ *z2*]]-*zs*)
⟨*proof*⟩

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* $\longrightarrow$ *qFresh xs x X*) $\wedge$
    (*qAFreshAbs xs x A* $\longrightarrow$ *qFreshAbs xs x A*)
⟨*proof*⟩

corollary *qAFresh-imp-qFresh*:
qAFresh xs x X $\Longrightarrow$ *qFresh xs x X*
⟨*proof*⟩

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* $\wedge$ *z2*]-*zs*) (*X* #[[*z1* $\wedge$ *z2*]]-*zs*) = *qAFresh ys y X*) $\wedge$
 (*qAFreshAbs ys* (*y* @*ys*[*z1* $\wedge$ *z2*]-*zs*) (*A* $[[*z1* $\wedge$ *z2*]]-*zs*) = *qAFreshAbs ys y A*)
⟨*proof*⟩

corollary *qSwap-preserves-qAFresh*[*simp*]:
(*qAFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh ys y X*)
⟨*proof*⟩

lemma *qSwap-preserves-qAFresh-distinct*:
assumes *ys* ≠ *zs* ∨ *y* ∉ {*z1*,*z2*}
shows *qAFresh ys y* (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh ys y X*
⟨*proof*⟩

lemma *qAFresh-qSwap-exchange1*:
qAFresh zs z2 (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qAFresh zs z1 X*
⟨*proof*⟩

lemma *qAFresh-qSwap-exchange2*:
qAFresh zs z2 (*X* #[[*z2* ∧ *z1*]]-*zs*) = *qAFresh zs z1 X*
⟨*proof*⟩

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*⟩

corollary *qSwap-preserves-qFresh*:
(*qFresh ys* (*y* @*ys*[*z1* ∧ *z2*]-*zs*) (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh ys y X*)
⟨*proof*⟩

lemma *qSwap-preserves-qFresh-distinct*:
assumes *ys* ≠ *zs* ∨ *y* ∉ {*z1*,*z2*}
shows *qFresh ys y* (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh ys y X*
⟨*proof*⟩

lemma *qFresh-qSwap-exchange1*:
qFresh zs z2 (*X* #[[*z1* ∧ *z2*]]-*zs*) = *qFresh zs z1 X*
⟨*proof*⟩

lemma *qFresh-qSwap-exchange2*:
qFresh zs z1 X = *qFresh zs z2* (*X* #[[*z2* ∧ *z1*]]-*zs*)
⟨*proof*⟩

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*|
⟨*proof*⟩

lemma *varSort-le-Var*:
|*UNIV* :: *'varSort set*| ≤*o* |*UNIV* :: *'var set*|
⟨*proof*⟩

theorem *var-infinite-INNER*: $infinite\ (UNIV :: 'var\ set)$
⟨*proof*⟩

theorem *var-regular-INNER*: $regular\ |UNIV :: 'var\ set|$
⟨*proof*⟩

theorem *infinite-var-regular-INNER*:
$infinite\ (UNIV :: 'var\ set) \wedge regular\ |UNIV :: 'var\ set|$
⟨*proof*⟩

theorem *finite-ordLess-var*:
$(\ |S| <o\ |UNIV :: 'var\ set| \vee finite\ S) = (\ |S| <o\ |UNIV :: 'var\ set|\ )$
⟨*proof*⟩

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
$qGood\ (qVar\ xs\ x) = True$
|
$qGood\ (qOp\ delta\ inp\ binp) =$
$(liftAll\ qGood\ inp \wedge liftAll\ qGoodAbs\ binp \wedge$
$|\{i.\ inp\ i \neq None\}| <o\ |UNIV :: 'var\ set| \wedge$
$|\{i.\ binp\ i \neq None\}| <o\ |UNIV :: 'var\ set|\ )$
|
$qGoodAbs\ (qAbs\ xs\ x\ X) = qGood\ X$
⟨*proof*⟩
termination
⟨*proof*⟩

fun $qGoodItem :: ('index,'bindex,'varSort,'var,'opSym)qTermItem \Rightarrow bool$ **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
$(qGood\ X \longrightarrow qGood\ (X\ \#[[x \wedge y]]\text{-}zs)) \wedge$
$(qGoodAbs\ A \longrightarrow qGoodAbs\ (A\ \$[[x \wedge y]]\text{-}zs))$
⟨*proof*⟩

corollary *qSwap-preserves-qGood1*:
$qGood\ X \Longrightarrow qGood\ (X\ \#[[x \wedge y]]\text{-}zs)$
⟨*proof*⟩

corollary *qSwapAbs-preserves-qGoodAbs1*:
$qGoodAbs\ A \Longrightarrow qGoodAbs\ (A\ \$[[x \wedge y]]\text{-}zs)$
⟨*proof*⟩

lemma *qSwap-preserves-qGood2*:
assumes $qGood(X\ \#[[x \wedge y]]\text{-}zs)$
shows *qGood X*
⟨*proof*⟩

lemma *qSwapAbs-preserves-qGoodAbs2*:
assumes $qGoodAbs(A\ \$[[x \wedge y]]\text{-}zs)$
shows *qGoodAbs A*
⟨*proof*⟩

lemma *qSwap-preserves-qGood*: $(qGood\ (X\ \#[[x \wedge y]]\text{-}zs)) = (qGood\ X)$
⟨*proof*⟩

lemma *qSwapAbs-preserves-qGoodAbs*:
$(qGoodAbs\ (A\ \$[[x \wedge y]]\text{-}zs)) = (qGoodAbs\ A)$
⟨*proof*⟩

lemma *qSwap-twice-preserves-qGood*:
$(qGood\ ((X\ \#[[x \wedge y]]\text{-}zs)\ \#[[x' \wedge y']]\text{-}zs')) = (qGood\ X)$
⟨*proof*⟩

lemma *qSwapped-preserves-qGood*:
$(X,Y) \in qSwapped \Longrightarrow qGood\ Y = qGood\ X$
⟨*proof*⟩

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: $\bigwedge X\ Y.\ [\![qGood\ X;\ (X,Y) \in rel]\!] \Longrightarrow qGood\ Y \wedge qSkel\ Y = qSkel\ X$ **and**
Var: $\bigwedge xs\ x.\ phi\ (qVar\ xs\ x)$ **and**
Op: $\bigwedge delta\ inp\ binp.\ [\![|\{i.\ inp\ i \neq None\}| <o\ |UNIV :: 'var\ set|;$
$|\{i.\ binp\ i \neq None\}| <o\ |UNIV :: 'var\ set|;$

liftAll (λX. *qGood X* ∧ *phi X*) *inp*;
liftAll (λA. *qGoodAbs A* ∧ *phiAbs A*) *binp*⟧
⟹ *phi* (*qOp delta inp binp*) **and**
Abs: ⋀ *xs x X*. ⟦*qGood X*; ⋀ *Y*. (*X*,*Y*) ∈ *rel* ⟹ *phi Y*⟧
⟹ *phiAbs* (*qAbs xs x X*)
shows
(*qGood X* ⟶ *phi X*) ∧ (*qGoodAbs A* ⟶ *phiAbs A*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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: $\bigwedge$ *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: $\bigwedge$ *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: $\bigwedge$ *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*) **and**
Abs2: $\bigwedge$ *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*))
⟨*proof*⟩

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*⟩

corollary *single-non-qAFresh-ordLess-var*:
qGood X ⟹ |{*x*. ¬ *qAFresh xs x X*}| <*o* |*UNIV* :: *'var set*|
⟨*proof*⟩

corollary *single-non-qAFreshAbs-ordLess-var*:
qGoodAbs A ⟹ |{*x*. ¬ *qAFreshAbs xs x A*}| <*o* |*UNIV* :: *'var set*|
⟨*proof*⟩

lemma *single-non-qFresh-ordLess-var*:
assumes *qGood X*
shows |{*x*. ¬ *qFresh xs x X*}| <*o* |*UNIV* :: *'var set*|
⟨*proof*⟩

lemma *single-non-qFreshAbs-ordLess-var*:
assumes *qGoodAbs A*

shows $|\{x.\ \neg\ qFreshAbs\ xs\ x\ A\}|$ $<o$ $|UNIV :: 'var\ set|$
⟨*proof*⟩

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*⟩

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*⟩

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 $\exists\ W.\ infinite\ W \wedge W\ Int\ V = \{\} \wedge$
$(\forall\ x \in W.\ \forall\ X \in XS.\ qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$
⟨*proof*⟩

lemma *obtain-set-qFresh*:
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 $\exists\ W.\ infinite\ W \wedge W\ Int\ V = \{\} \wedge$
$(\forall\ x \in W.\ \forall\ X \in XS.\ qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$
⟨*proof*⟩

lemma *obtain-qFresh-card-of*:
assumes $|V|$ $<o$ $|UNIV :: 'var\ set|$ **and** $|XS|$ $<o$ $|UNIV :: 'var\ set|$ **and** $\forall\ X \in XS.\ qGood\ X$
shows $\exists\ x.\ x \notin V \wedge (\forall\ X \in XS.\ qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$
⟨*proof*⟩

lemma *obtain-qFresh*:
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 $\exists\ x.\ x \notin V \wedge (\forall\ X \in XS.\ qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$
⟨*proof*⟩

definition *pickQFresh* **where**
$pickQFresh\ xs\ V\ XS ==$
$SOME\ x.\ x \notin V \wedge (\forall\ X \in XS.\ qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⇒
(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTermItem*
where
aux-alpha-ignoreSecond K ==
case K of Inl(*X*,*Y*) ⇒ *termIn X*
|*Inr*(*A*,*B*) ⇒ *absIn A*

lemma *aux-alpha-ignoreSecond-qTermLessQSwapped-wf*:
wf(*inv-image qTermQSwappedLess aux-alpha-ignoreSecond*)
⟨*proof*⟩

function
alpha **and** *alphaAbs*
where
alpha (*qVar xs x*) (*qVar xs' x'*) ⟷ *xs* = *xs'* $\wedge$ *x* = *x'*
|
alpha (*qOp delta inp binp*) (*qOp delta' inp' binp'*) ⟷
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′*))
⟨*proof*⟩
termination
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *alpha-binp-None*:
qOp delta inp binp #= *qOp delta′ inp′ binp′* ⟹
(*binp i* = *None*) = (*binp′ i* = *None*)
⟨*proof*⟩

lemma *qVar-alpha-iff*:
qVar xs x #= *X′* ⟷ *X′* = *qVar xs x*
⟨*proof*⟩

lemma *alpha-qVar-iff*:
X #= *qVar xs′ x′* ⟷ *X* = *qVar xs′ x′*
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *alpha-qOp-iff*:
X #= *qOp delta′ inp′ binp′* ⟷
(∃ *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′*)

⟨*proof*⟩

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

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

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*⟩

corollary *alpha-qSkel*:
fixes $X\ X'$ :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
shows $X\ \#=\ X' \Longrightarrow qSkel\ X = qSkel\ X'$
⟨*proof*⟩

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
$(\forall\ X'.\ X\ \#=\ X' \longrightarrow X'\ \#=\ X) \wedge (\forall\ A'.\ A\ \$=\ A' \longrightarrow A'\ \$=\ A)$
⟨*proof*⟩

corollary *alpha-sym*:
fixes $X\ X'$ :: ('*index*,'*bindex*,'*varSort*,'*var*,'*opSym*)*qTerm*
shows $X\ \#=\ X' \Longrightarrow X'\ \#=\ X$
⟨*proof*⟩

corollary *alphaAbs-sym*:
fixes $A\ A'$ ::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*
shows A \$= $A' \Longrightarrow A'$ \$= A
⟨*proof*⟩

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) ∧ (*qGoodAbs* $A \longrightarrow A$ \$= A)
⟨*proof*⟩

corollary *alpha-refl*:
fixes X :: (*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*
shows *qGood* $X \Longrightarrow X$ #= X
⟨*proof*⟩

corollary *alphaAbs-refl*:
fixes A ::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*
shows *qGoodAbs* $A \Longrightarrow A$ \$= A
⟨*proof*⟩

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')) ∧
(*qGoodAbs* $A \longrightarrow (\forall\ A'.\ A$ \$= $A' \longrightarrow$ *qGoodAbs* A'))
⟨*proof*⟩

corollary *alpha-preserves-qGood1*:
⟦X #= X'; *qGood* X⟧ ⟹ *qGood* X'
⟨*proof*⟩

corollary *alphaAbs-preserves-qGoodAbs1*:
⟦A \$= A'; *qGoodAbs* A⟧ ⟹ *qGoodAbs* A'
⟨*proof*⟩

lemma *alpha-preserves-qGood2*:
⟦X #= X'; *qGood* X'⟧ ⟹ *qGood* X
⟨*proof*⟩

lemma *alphaAbs-preserves-qGoodAbs2*:
⟦A \$= A'; *qGoodAbs* A'⟧ ⟹ *qGoodAbs* A
⟨*proof*⟩

lemma *alpha-preserves-qGood*:
$X \mathrel{\#}= X' \Longrightarrow qGood\ X = qGood\ X'$
$\langle proof \rangle$

lemma *alphaAbs-preserves-qGoodAbs*:
$A \ \$= A' \Longrightarrow qGoodAbs\ A = qGoodAbs\ A'$
$\langle proof \rangle$

lemma *alpha-qSwap-preserves-qGood1*:
assumes $ALPHA$: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$ **and**
$GOOD$: $qGood\ X$
shows $qGood\ X'$
$\langle proof \rangle$

lemma *alpha-qSwap-preserves-qGood2*:
assumes $ALPHA$: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$ **and**
$GOOD'$: $qGood\ X'$
shows $qGood\ X$
$\langle proof \rangle$

lemma *alphaAbs-qSwapAbs-preserves-qGoodAbs2*:
assumes $ALPHA$: $(A \ \$[[y \wedge x]]\text{-}zs) \ \$= (A' \ \$[[y' \wedge x']]\text{-}zs')$ **and**
$GOOD'$: $qGoodAbs\ A'$
shows $qGoodAbs\ A$
$\langle proof \rangle$

lemma *alpha-qSwap-preserves-qGood*:
assumes $ALPHA$: $(X \ \#[[y \wedge x]]\text{-}zs) \ \#= (X' \ \#[[y' \wedge x']]\text{-}zs')$
shows $qGood\ X = qGood\ X'$
$\langle proof \rangle$

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
$(qGood\ X \longrightarrow (\forall\ X'\ zs\ z1\ z2.\ X \ \#= X' \longrightarrow$
$(X \ \#[[z1 \wedge z2]]\text{-}zs) \ \#= (X' \ \#[[z1 \wedge z2]]\text{-}zs))) \wedge$
$(qGoodAbs\ A \longrightarrow (\forall\ A'\ zs\ z1\ z2.\ A \ \$= A' \longrightarrow$
$(A \ \$[[z1 \wedge z2]]\text{-}zs) \ \$= (A' \ \$[[z1 \wedge z2]]\text{-}zs)))$
$\langle proof \rangle$

corollary *qSwap-preserves-alpha*:
assumes $qGood\ X \vee qGood\ X'$ **and** $X \ \#= X'$
shows $(X \ \#[[z1 \wedge z2]]\text{-}zs) \ \#= (X' \ \#[[z1 \wedge z2]]\text{-}zs)$
$\langle proof \rangle$

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

⟨*proof*⟩

lemma *qSwap-twice-preserves-alpha*:
assumes *qGood X* ∨ *qGood X′* **and** *X* #= *X′*
shows ((*X* #[[*z1* ∧ *z2*]]-*zs*) #[[*u1* ∧ *u2*]]-*us*) #= ((*X′* #[[*z1* ∧ *z2*]]-*zs*) #[[*u1* ∧ *u2*]]-*us*)
⟨*proof*⟩

lemma *alphaAll-trans*:
fixes *X*::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qTerm* **and**
A::(*′index*,*′bindex*,*′varSort*,*′var*,*′opSym*)*qAbs*
shows
(*qGood X* ⟶ (∀ *X′ X″*. *X* #= *X′* ∧ *X′* #= *X″* ⟶ *X* #= *X″*)) ∧
(*qGoodAbs A* ⟶ (∀ *A′ A″*. *A* $= *A′* ∧ *A′* $= *A″* ⟶ *A* $= *A″*))
⟨*proof*⟩

corollary *alpha-trans*:
assumes *qGood X* ∨ *qGood X′* ∨ *qGood X″ X* #= *X′* *X′* #= *X″*
shows *X* #= *X″*
⟨*proof*⟩

corollary *alphaAbs-trans*:
assumes *qGoodAbs A* ∨ *qGoodAbs A′* ∨ *qGoodAbs A″*
and *A* $= *A′* *A′* $= *A″*
shows *A* $= *A″*
⟨*proof*⟩

lemma *alpha-trans-twice*:
⟦*qGood X* ∨ *qGood X′* ∨ *qGood X″* ∨ *qGood X‴*;
X #= *X′*; *X′* #= *X″*; *X″* #= *X‴*⟧ ⟹ *X* #= *X‴*
⟨*proof*⟩

lemma *alphaAbs-trans-twice*:
⟦*qGoodAbs A* ∨ *qGoodAbs A′* ∨ *qGoodAbs A″* ∨ *qGoodAbs A‴*;
A $= *A′*; *A′* $= *A″*; *A″* $= *A‴*⟧ ⟹ *A* $= *A‴*
⟨*proof*⟩

lemma *qAbs-preserves-alpha*:
assumes *ALPHA*: *X* #= *X′* **and** *GOOD*: *qGood X* ∨ *qGood X′*
shows *qAbs xs x X* $= *qAbs xs x X′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

corollary *qFresh-imp-ex-qAFresh*:
assumes *finite V* **and** *qGood X* **and** ∀ *z* ∈ *V*. ∀ *zs* ∈ *fZs z*. *qFresh zs z X*
shows ∃ *X′*. *qGood X′* ∧ *X* #= *X′* ∧ (∀ *z* ∈ *V*. ∀ *zs* ∈ *fZs z*. *qAFresh zs z X′*)
⟨*proof*⟩

corollary *qFreshAbs-imp-ex-qAFreshAbs*:
assumes *finite V* **and** *qGoodAbs A* **and** ∀ *z* ∈ *V*. ∀ *zs* ∈ *fZs z*. *qFreshAbs zs z A*
shows ∃ *A′*. *qGoodAbs A′* ∧ *A* \$= *A′* ∧ (∀ *z* ∈ *V*. ∀ *zs* ∈ *fZs z*. *qAFreshAbs zs z A′*)
⟨*proof*⟩

lemma *qFresh-imp-ex-qAFresh1*:
assumes *qGood X* **and** *qFresh zs z X*
shows ∃ *X′*. *qGood X′* ∧ *X* #= *X′* ∧ *qAFresh zs z X′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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'
⟨*proof*⟩

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'
⟨*proof*⟩

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*⟩

corollary *qAFresh-qFresh-preserves-alpha*:
⟦*qGood X*; *qAFresh zs z X*; X #= X'⟧ ⟹ *qFresh zs z* X'
⟨*proof*⟩

corollary *qAFreshAbs-imp-qFreshAbs-preserves-alphaAbs*:
⟦*qGoodAbs A*; *qAFreshAbs zs z A*; A \$= A'⟧ ⟹ *qFreshAbs zs z* A'
⟨*proof*⟩

lemma *qFresh-preserves-alpha1*:
assumes *qGood X* **and** *qFresh zs z X* **and** X #= X'
shows *qFresh zs z* X'
⟨*proof*⟩

lemma *qFreshAbs-preserves-alphaAbs1*:
assumes *qGoodAbs A* **and** *qFreshAbs zs z A* **and** A \$= A'
shows *qFreshAbs zs z* A'
⟨*proof*⟩

lemma *qFresh-preserves-alpha*:
assumes *qGood X* ∨ *qGood* X' **and** X #= X'
shows *qFresh zs z X* ⟷ *qFresh zs z* X'
⟨*proof*⟩

lemma *qFreshAbs-preserves-alphaAbs*:
assumes *qGoodAbs A* ∨ *qGoodAbs* A' **and** A \$= A'
shows *qFreshAbs zs z A* = *qFreshAbs zs z* A'
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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′ ==
($xs = xs' \wedge$
($\forall$ y. ($y = x \vee qAFresh\ xs\ y\ X$) $\wedge$ ($y = x' \vee qAFresh\ xs\ y\ X'$) $\longrightarrow$
(X #[[$y \wedge x$]]-xs) #= (X' #[[$y \wedge x'$]]-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 #[[$y \wedge x$]]-xs) #= (X' #[[$y \wedge x'$]]-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 #[[$y \wedge x$]]-xs) #= (X' #[[$y \wedge x'$]]-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*⟩

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*⟩

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*⟩

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′*
⟨*proof*⟩

corollary *alphaAbs-qAbs-iff-ex-distinct-qAFresh*:
(*qAbs xs x X* $= *qAbs xs′ x′ X′*) =
($xs = xs' \wedge$
($\exists$ y. $y \notin \{x,x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \wedge$

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

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qAFresh*:
assumes *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$alphaAbs\text{-}ex\text{-}equal\text{-}or\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

corollary *alphaAbs-qAbs-iff-ex-equal-or-qAFresh*:
assumes *qGood X*
shows
$(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X') \wedge$
$(X \#[[y \wedge x]]\text{-}xs) \#= (X' \#[[y \wedge x']]\text{-}xs)))$
⟨*proof*⟩

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-qAFresh*:
assumes *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') = alphaAbs\text{-}ex\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

corollary *alphaAbs-qAbs-iff-ex-qAFresh*:
assumes *qGood X*
shows
$(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \wedge$
$(X \#[[y \wedge x]]\text{-}xs) \#= (X' \#[[y \wedge x']]\text{-}xs)))$
⟨*proof*⟩

lemma *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh*:
assumes *qGood X* **and** $qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X'$
shows $alphaAbs\text{-}all\text{-}equal\text{-}or\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

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*⟩

lemma *alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh*:
assumes *qGood X* **and** *qGood X'*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$alphaAbs\text{-}all\text{-}equal\text{-}or\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$

⟨*proof*⟩

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*⟩

lemma *alphaAbs-qAbs-imp-alphaAbs-all-qAFresh*:
assumes *qGood X* **and** $qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X'$
shows $alphaAbs\text{-}all\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *alphaAbs-qAbs-imp-alphaAbs-all-distinct-qAFresh*:
assumes *qGood X* **and** $qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X'$
shows $alphaAbs\text{-}all\text{-}distinct\text{-}qAFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

corollary *alphaAbs-qAbs-imp-all-distinct-qAFresh*:
assumes *qGood X* **and** $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X')$
shows
$(xs = xs' \wedge$
$(\forall\ y.\ y \notin \{x,x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
⟨*proof*⟩

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*⟩

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* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))
⟨*proof*⟩

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'* ∧
(∃ *y*. (*y* = *x* ∨ *qFresh xs y X*) ∧ (*y* = *x'* ∨ *qFresh xs y X'*) ∧
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))

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

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

definition *alphaAbs-all-equal-or-qFresh*
where
alphaAbs-all-equal-or-qFresh xs x X 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*)))

definition *alphaAbs-all-qFresh*
where
alphaAbs-all-qFresh xs x X xs' x' X' ==
(*xs* = *xs'* ∧
(∀ *y*. *qFresh xs y X* ∧ *qFresh xs y X'* ⟶

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

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

lemma *alphaAbs-ex-equal-or-qAFresh-imp-qFresh*:
alphaAbs-ex-equal-or-qAFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-ex-equal-or-qFresh xs x X xs' x' X'
⟨*proof*⟩

lemma *alphaAbs-ex-distinct-qAFresh-imp-qFresh*:
alphaAbs-ex-distinct-qAFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-ex-distinct-qFresh xs x X xs' x' X'
⟨*proof*⟩

lemma *alphaAbs-ex-qAFresh-imp-qFresh*:
alphaAbs-ex-qAFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-ex-qFresh xs x X xs' x' X'
⟨*proof*⟩

lemma *alphaAbs-all-equal-or-qFresh-imp-qAFresh*:
alphaAbs-all-equal-or-qFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-all-equal-or-qAFresh xs x X xs' x' X'
⟨*proof*⟩

lemma *alphaAbs-all-distinct-qFresh-imp-qAFresh*:
alphaAbs-all-distinct-qFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-all-distinct-qAFresh xs x X xs' x' X'
⟨*proof*⟩

lemma *alphaAbs-all-qFresh-imp-qAFresh*:
alphaAbs-all-qFresh xs x X xs' x' X' $\Longrightarrow$
alphaAbs-all-qAFresh xs x X xs' x' X'
⟨*proof*⟩

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*⟩

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*⟩

corollary *alphaAbs-qAbs-iff-ex-equal-or-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$(xs = xs' \wedge$
$(\exists\ y.\ (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \wedge$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
⟨*proof*⟩

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$alphaAbs\text{-}ex\text{-}qFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

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*⟩

lemma *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qFresh*:
assumes *GOOD*: *qGood X*
shows $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X') =$
$alphaAbs\text{-}ex\text{-}distinct\text{-}qFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

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*⟩

lemma *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh*:
assumes *qGood X* **and** $qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X'$
shows $alphaAbs\text{-}all\text{-}equal\text{-}or\text{-}qFresh\ xs\ x\ X\ xs'\ x'\ X'$
⟨*proof*⟩

corollary *alphaAbs-qAbs-imp-all-equal-or-qFresh*:
assumes *qGood X* **and** $(qAbs\ xs\ x\ X\ \$=\ qAbs\ xs'\ x'\ X')$
shows
$(xs = xs' \wedge$
$(\forall\ y.\ (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \longrightarrow$
$(X\ \#[[y \wedge x]]\text{-}xs)\ \#=\ (X'\ \#[[y \wedge x']]\text{-}xs)))$
⟨*proof*⟩

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*⟩

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'* ∧
(∀ *y*. (*y* = *x* ∨ *qFresh xs y X*) ∧ (*y* = *x'* ∨ *qFresh xs y X'*) ⟶
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))
⟨*proof*⟩

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*⟩

corollary *alphaAbs-qAbs-imp-all-qFresh*:
assumes *qGood X* **and** (*qAbs xs x X* $= *qAbs xs' x' X'*)
shows
(*xs* = *xs'* ∧
(∀ *y*. *qFresh xs y X* ∧ *qFresh xs y X'* ⟶
(*X* #[[*y* ∧ *x*]]-*xs*) #= (*X'* #[[*y* ∧ *x'*]]-*xs*)))
⟨*proof*⟩

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*⟩

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*⟩

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 (λ*X 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 ==
 $pickQFresh\ xs\ (V \cup (\bigcup\ rho \in Rho.\ \{x.\ rho\ xs\ x \neq None\}))$
 $(XS \cup (\bigcup\ rho \in Rho.\ \{X.\ \exists\ ys\ y.\ rho\ ys\ y = Some\ X\}))$

lemma *qGoodEnv-imp-card-of-qTerm*:
assumes *qGoodEnv rho*
shows $|\{X.\ \exists\ y.\ rho\ ys\ y = Some\ X\}| <o |UNIV :: 'var\ set|$
⟨*proof*⟩

lemma *qGoodEnv-imp-card-of-qTerm2*:
assumes *qGoodEnv rho*
shows $|\{X.\ \exists\ ys\ y.\ rho\ ys\ y = Some\ X\}| <o |UNIV :: 'var\ set|$
⟨*proof*⟩

lemma *qGoodEnv-iff*:
qGoodEnv rho =
 $((\forall\ xs.\ liftAll\ qGood\ (rho\ xs)) \wedge$
 $(\forall\ ys.\ |\{y.\ rho\ ys\ y \neq None\}| <o |UNIV :: 'var\ set|\ ) \wedge$
 $|\{X.\ \exists\ ys\ y.\ rho\ ys\ y = Some\ X\}| <o |UNIV :: 'var\ set|\ )$
⟨*proof*⟩

lemma *alphaEnv-refl*:
$qGoodEnv\ rho \Longrightarrow rho\ \&= rho$
⟨*proof*⟩

lemma *alphaEnv-sym*:
$rho\ \&= rho' \Longrightarrow rho'\ \&= rho$
⟨*proof*⟩

lemma *alphaEnv-trans*:
assumes *good*: *qGoodEnv rho* **and**
 alpha1: $rho\ \&= rho'$ **and** *alpha2*: $rho'\ \&= rho''$
shows $rho\ \&= rho''$
⟨*proof*⟩

lemma *pickQFreshEnv-card-of*:
assumes *Vvar*: $|V| <o |UNIV :: 'var\ set|$ **and** *XSvar*: $|XS| <o |UNIV :: 'var\ set|$
and
 good: $\forall\ X \in XS.\ qGood\ X$ **and**
 Rhovar: $|Rho| <o |UNIV :: 'var\ set|$ **and** *RhoGood*: $\forall\ rho \in Rho.\ qGoodEnv$
rho
shows
$pickQFreshEnv\ xs\ V\ XS\ Rho \notin V\ \wedge$
 $(\forall\ X \in XS.\ qFresh\ xs\ (pickQFreshEnv\ xs\ V\ XS\ Rho)\ X) \wedge$
 $(\forall\ rho \in Rho.\ qFreshEnv\ xs\ (pickQFreshEnv\ xs\ V\ XS\ Rho)\ rho)$
⟨*proof*⟩

lemma *pickQFreshEnv*:
assumes *Vvar*: $|V| <o |UNIV :: 'var\ set| \lor finite\ V$
and *XSvar*: $|XS| <o |UNIV :: 'var\ set| \lor finite\ XS$
and *good*: $\forall\ X \in XS.\ qGood\ X$
and *Rhovar*: $|Rho| <o |UNIV :: 'var\ set| \lor finite\ Rho$
and *RhoGood*: $\forall\ rho \in Rho.\ qGoodEnv\ rho$
shows
$pickQFreshEnv\ xs\ V\ XS\ Rho \notin V \land$
$(\forall\ X \in XS.\ qFresh\ xs\ (pickQFreshEnv\ xs\ V\ XS\ Rho)\ X) \land$
$(\forall\ rho \in Rho.\ qFreshEnv\ xs\ (pickQFreshEnv\ xs\ V\ XS\ Rho)\ rho)$
⟨*proof*⟩

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*: $|V| <o |UNIV :: 'var\ set| \lor finite\ V$
and *XSvar*: $|XS| <o |UNIV :: 'var\ set| \lor finite\ XS$
and *good*: $\forall\ X \in XS.\ qGood\ X$
and *Rhovar*: $|Rho| <o |UNIV :: 'var\ set| \lor finite\ Rho$
and *RhoGood*: $\forall\ rho \in Rho.\ qGoodEnv\ rho$
shows
$\exists\ z.\ z \notin V \land$
$(\forall\ X \in XS.\ qFresh\ xs\ z\ X) \land (\forall\ rho \in Rho.\ qFreshEnv\ xs\ z\ rho)$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩
termination
⟨*proof*⟩

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*⟩

corollary *qPsubst-preserves-qGood*:
⟦*qGoodEnv rho*; *qGood X*⟧ ⟹ *qGood* (*X* #[[*rho*]])
⟨*proof*⟩

corollary *qPsubstAbs-preserves-qGoodAbs*:
⟦*qGoodEnv rho*; *qGoodAbs A*⟧ ⟹ *qGoodAbs* (*A* $[[*rho*]])
⟨*proof*⟩

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*⟩

lemma *qPsubst-preserves-qFresh*:
⟦*qGood X*; *qGoodEnv rho*; *qFresh zs z X*; *qFreshEnv zs z rho*⟧
 ⟹ *qFresh zs z* (*X* #[[*rho*]])
⟨*proof*⟩

lemma *qPsubstAbs-preserves-qFreshAbs*:
⟦*qGoodAbs A*; *qGoodEnv rho*; *qFreshAbs zs z A*; *qFreshEnv zs z rho*⟧
 ⟹ *qFreshAbs zs z* (*A* $[[*rho*]])
⟨*proof*⟩

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*⟩

corollary *qPsubst-preserves-alpha1*:
assumes *qGoodEnv rho* **and** *qGood X* ∨ *qGood Y* **and** *X* #= *Y*
shows (*X* #[[*rho*]]) #= (*Y* #[[*rho*]])
⟨*proof*⟩

corollary *qPsubstAbs-preserves-alphaAbs1*:
assumes *qGoodEnv rho* **and** *qGoodAbs A* ∨ *qGoodAbs B* **and** *A* $= *B*
shows (*A* $[[*rho*]]) $= (*B* $[[*rho*]])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

corollary *qPsubst-preserves-alpha2*:
⟦*qGood X*; *qGoodEnv rho'*; *qGoodEnv rho''*; *rho'* &= *rho''*⟧
⟹ (*X* #[[*rho'*]]) #= (*X* #[[*rho''*]])
⟨*proof*⟩

corollary *qPsubstAbs-preserves-alphaAbs2*:
⟦*qGoodAbs A*; *qGoodEnv rho'*; *qGoodEnv rho''*; *rho'* &= *rho''*⟧
⟹ (*A* $[[*rho'*]]) $= (*A* $[[*rho''*]])
⟨*proof*⟩

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'*]])
⟨*proof*⟩

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'*]])
⟨*proof*⟩

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*⟩

end

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

begin

definition *pick* $X \equiv$ *SOME* $x.\ x \in X$

lemma *pick*[*simp*]: $x \in X \Longrightarrow$ *pick* $X \in X$
⟨*proof*⟩

lemma *pick-NE*[*simp*]: $X \neq \{\} \Longrightarrow$ *pick* $X \in X$ ⟨*proof*⟩

end

4 Some preliminaries on equivalence relations and quotients

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

Unary predicates vs. sets:

definition *S2P* $A \equiv \lambda\ x.\ x \in A$

lemma *S2P-app*[*simp*]: *S2P* $r\ x \longleftrightarrow x \in r$
⟨*proof*⟩

lemma *S2P-Collect*[*simp*]: *S2P* (*Collect* φ) = φ
⟨*proof*⟩

lemma *Collect-S2P*[*simp*]: *Collect* (*S2P* r) = r
⟨*proof*⟩

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)
⟨*proof*⟩

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

lemma *R2P-app*[*simp*]: *R2P* $r\ x\ y \longleftrightarrow (x,y) \in r$
⟨*proof*⟩

lemma *R2P-P2R*[*simp*]: *R2P* (*P2R* φ) = φ
⟨*proof*⟩

lemma *P2R-R2P*[*simp*]: *P2R* (*R2P* r) = r
⟨*proof*⟩

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
⟨*proof*⟩

lemma *reflP-R2P*[*simp*]: *reflP* (*S2P A*) (*R2P r*) $\longleftrightarrow$ *refl-on A r*
⟨*proof*⟩

lemma *sym-P2R*[*simp*]: *sym* (*P2R* φ) $\longleftrightarrow$ *symP* φ
⟨*proof*⟩

lemma *symP-R2P*[*simp*]: *symP* (*R2P r*) $\longleftrightarrow$ *sym r*
⟨*proof*⟩

lemma *trans-P2R*[*simp*]: *trans* (*P2R* φ) $\longleftrightarrow$ *transP* φ
⟨*proof*⟩

lemma *transP-R2P*[*simp*]: *transP* (*R2P r*) $\longleftrightarrow$ *trans r*
⟨*proof*⟩

lemma *equiv-P2R*[*simp*]: *equiv* (*Collect P*) (*P2R* φ) $\longleftrightarrow$ *equivP P* φ
⟨*proof*⟩

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

lemma *in-P2R-Im-singl*[*simp*]: $y \in$ *P2R* φ '' $\{x\} \longleftrightarrow \varphi\ x\ y$ ⟨*proof*⟩

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

lemma *proj-P2R*: *proj* $\varphi\ x =$ *P2R* φ '' $\{x\}$ ⟨*proof*⟩

lemma *proj-P2R-raw*: *proj* $\varphi = (\lambda\ x.$ *P2R* φ '' $\{x\})$
⟨*proof*⟩

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

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

lemma *proj-preserves*:
P x ⟹ (*P* /// φ) (*proj* φ *x*)

⟨*proof*⟩

lemma *proj-in-iff*:
assumes *equivP P* φ
shows $(P///\varphi)$ $(proj\ \varphi\ x)$ $\longleftrightarrow$ $P\ x$
⟨*proof*⟩

lemma *proj-iff*[*simp*]:
$[\![equivP\ P\ \varphi;\ P\ x;\ P\ y]\!] \Longrightarrow proj\ \varphi\ x = proj\ \varphi\ y \longleftrightarrow \varphi\ x\ y$
⟨*proof*⟩

lemma *in-proj*[*simp*]: $[\![equivP\ P\ \varphi;\ P\ x]\!] \Longrightarrow x \in proj\ \varphi\ x$
⟨*proof*⟩

lemma *proj-image*[*simp*]: $(proj\ \varphi)$ ‘ $(Collect\ P) = Collect\ (P///\varphi)$
⟨*proof*⟩

lemma *in-quotientP-imp-non-empty*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows $X \neq \{\}$
⟨*proof*⟩

lemma *in-quotientP-imp-in-rel*:
$[\![equivP\ P\ \varphi;\ (P///\varphi)\ X;\ x \in X;\ y \in X]\!] \Longrightarrow \varphi\ x\ y$
⟨*proof*⟩

lemma *in-quotientP-imp-closed*:
$[\![equivP\ P\ \varphi;\ (P///\varphi)\ X;\ x \in X;\ \varphi\ x\ y]\!] \Longrightarrow y \in X$
⟨*proof*⟩

lemma *in-quotientP-imp-subset*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows $X \subseteq$ *Collect P*
⟨*proof*⟩

lemma *equivP-pick-in*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows *pick* $X \in X$
⟨*proof*⟩

lemma *equivP-pick-preserves*:
assumes *equivP P* φ **and** $(P///\varphi)$ X
shows P $(pick\ X)$
⟨*proof*⟩

lemma *proj-pick*:
assumes φ: *equivP P* φ **and** X: $(P///\varphi)$ X
shows $proj\ \varphi\ (pick\ X) = X$
⟨*proof*⟩

lemma *pick-proj*:
assumes *equivP P* φ **and** *P x*
shows φ (*pick* (*proj* φ *x*)) *x*
⟨*proof*⟩

lemma *equivP-pick-iff* [*simp*]:
assumes φ: *equivP P* φ **and** *X*: (*P*///φ) *X* **and** *Y*: (*P*///φ) *Y*
shows φ (*pick X*) (*pick Y*) $\longleftrightarrow$ *X* = *Y*
⟨*proof*⟩

lemma *equivP-pick-inj-on*:
assumes *equivP P* φ
shows *inj-on pick* (*Collect* (*P*///φ))
⟨*proof*⟩

definition *congruentP* **where**
congruentP φ *f* $\equiv$ $\forall$ *x y*. φ *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*
⟨*proof*⟩

lemma *univ-commute*[*simp*]:
assumes *equivP P* φ **and** *f respectsP* φ **and** *P x*
shows (*univ f*) (*proj* φ *x*) = *f x*
⟨*proof*⟩

lemma *univ-unique*:
assumes *equivP P* φ **and** *f respectsP* φ **and** $\bigwedge$ *x*. *P x* $\Longrightarrow$ *G* (*proj* φ *x*) = *f x*
shows $\forall$ *X*. (*P*///φ) *X* $\longrightarrow$ *G X* = *univ f X*
⟨*proof*⟩

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*///φ) *X* $\longrightarrow$ *univ f X* $\in$ *B*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*]])
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

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 ≠ None}| <o |UNIV :: 'var set|

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

lemma *qGood-qGoodInp*:
qGood (qOp delta qinp qbinp) = (qGoodInp qinp ∧ qGoodBinp qbinp)
⟨*proof*⟩

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 (λqX 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'*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *alphaBinp-refl*:
qGoodBinp qbinp ⟹ *qbinp* %%= *qbinp*
⟨*proof*⟩

lemma *alphaInp-sym*:
fixes *qinp qinp'* :: (*'index*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qTerm*)*input*
shows *qinp* %= *qinp'* ⟹ *qinp'* %= *qinp*
⟨*proof*⟩

lemma *alphaBinp-sym*:
fixes *qbinp qbinp'* :: (*'bindex*,(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*qAbs*)*input*
shows *qbinp* %%= *qbinp'* ⟹ *qbinp'* %%= *qbinp*
⟨*proof*⟩

lemma *alphaInp-trans*:
assumes *good*: *qGoodInp qinp* **and**
alpha1: *qinp* %= *qinp'* **and** *alpha2*: *qinp'* %= *qinp''*
shows *qinp* %= *qinp''*

⟨*proof*⟩

lemma *alphaBinp-trans*:
assumes *good*: *qGoodBinp qbinp* **and**
alpha1: *qbinp* %%= *qbinp′* **and** *alpha2*: *qbinp′* %%= *qbinp″*
shows *qbinp* %%= *qbinp″*
⟨*proof*⟩

lemma *qSwapInp-preserves-qGoodInp*:
assumes *qGoodInp qinp*
shows *qGoodInp* (*qinp* %[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *qSwapBinp-preserves-qGoodBinp*:
assumes *qGoodBinp qbinp*
shows *qGoodBinp* (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *qSwapInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp′* **and** *qinp* %= *qinp′*
shows (*qinp* %[[*x1* ∧ *x2*]]-*xs*) %= (*qinp′* %[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *qSwapBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp′* **and** *qbinp* %%= *qbinp′*
shows (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*) %%= (*qbinp′* %%[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *qPsubstInp-preserves-qGoodInp*:
assumes *qGoodInp qinp* **and** *qGoodEnv qrho*
shows *qGoodInp* (*qinp* %[[*qrho*]])
⟨*proof*⟩

lemma *qPsubstBinp-preserves-qGoodBinp*:
assumes *qGoodBinp qbinp* **and** *qGoodEnv qrho*
shows *qGoodBinp* (*qbinp* %%[[*qrho*]])
⟨*proof*⟩

lemma *qPsubstInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp′* **and** *qGoodEnv qrho* **and** *qinp* %= *qinp′*
shows (*qinp* %[[*qrho*]]) %= (*qinp′* %[[*qrho*]])
⟨*proof*⟩

lemma *qPsubstBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp′* **and** *qGoodEnv qrho* **and** *qbinp* %%= *qbinp′*
shows (*qbinp* %%[[*qrho*]]) %%= (*qbinp′* %%[[*qrho*]])
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *qFreshInp-preserves-alphaInp*:
assumes *qGoodInp qinp* ∨ *qGoodInp qinp′* **and** *qinp* %= *qinp′*
shows *qFreshInp xs x qinp* ⟷ *qFreshInp xs x qinp′*
⟨*proof*⟩

lemma *qFreshBinp-preserves-alphaBinp*:
assumes *qGoodBinp qbinp* ∨ *qGoodBinp qbinp′* **and** *qbinp* %%= *qbinp′*
shows *qFreshBinp xs x qbinp* ⟷ *qFreshBinp xs x qbinp′*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *skelBinp-def2*:
assumes *goodBinp binp*
shows *skelBinp binp* = *lift skelAbs binp*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *substBinp-def2*:
substBinp ys Y y = *lift* (*substAbs ys Y y*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *vsubstBinp-def2*:
(*binp* %%[*y1* // *y2*]-*ys*) = *lift* (*vsubstAbs ys y1 y2*) *binp*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *termsOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *X* : *termsOf P*
shows *good X*
⟨*proof*⟩

lemma *absOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *A* : *absOf P*
shows *goodAbs A*
⟨*proof*⟩

lemma *envsOf-preserves-good*[*simp*]:
assumes *goodPar P* **and** *rho* : *envsOf P*
shows *goodEnv rho*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *univ-asTerm-alphaGood*[*simp*]:
assumes ∗: *congruentP alphaGood f* **and** ∗∗: *qGood X*

shows *univ f* (*asTerm X*) = *f X*
⟨*proof*⟩

corollary *univ-asTerm-alpha*[*simp*]:
assumes ∗: *congruentP alpha f* **and** ∗∗: *qGood X*
shows *univ f* (*asTerm X*) = *f X*
⟨*proof*⟩

lemma *pick-inj-on-good*: *inj-on pick* (*Collect good*)
⟨*proof*⟩

lemma *pick-injective-good*[*simp*]:
⟦*good X*; *good Y*⟧ ⟹ (*pick X* = *pick Y*) = (*X* = *Y*)
⟨*proof*⟩

lemma *good-imp-qGood-pick*:
good X ⟹ *qGood* (*pick X*)
⟨*proof*⟩

lemma *qGood-iff-good-asTerm*:
good (*asTerm qX*) = *qGood qX*
⟨*proof*⟩

lemma *pick-asTerm*:
assumes *qGood qX*
shows *pick* (*asTerm qX*) #= *qX*
⟨*proof*⟩

lemma *asTerm-pick*:
assumes *good X*
shows *asTerm* (*pick X*) = *X*
⟨*proof*⟩

lemma *pick-alpha*: *good X* ⟹ *pick X* #= *pick X*
⟨*proof*⟩

lemma *alpha-imp-asTerm-equal*:
assumes *qGood qX* **and** *qX* #= *qY*
shows *asTerm qX* = *asTerm qY*
⟨*proof*⟩

lemma *asTerm-equal-imp-alpha*:
assumes *qGood qX* **and** *asTerm qX* = *asTerm qY*
shows *qX* #= *qY*
⟨*proof*⟩

lemma *asTerm-equal-iff-alpha*:
assumes *qGood qX* ∨ *qGood qY*
shows (*asTerm qX* = *asTerm qY*) = (*qX* #= *qY*)

⟨proof⟩

lemma *pick-alpha-iff-equal*:
assumes *good X* **and** *good Y*
shows *pick X* #= *pick Y* ⟷ *X* = *Y*
⟨proof⟩

lemma *pick-swap-qSwap*:
assumes *good X*
shows *pick* (*X* #[*x1* ∧ *x2*]-*xs*) #= ((*pick X*) #[[*x1* ∧ *x2*]]-*xs*)
⟨proof⟩

lemma *asTerm-qSwap-swap*:
assumes *qGood qX*
shows *asTerm* (*qX* #[[*x1* ∧ *x2*]]-*xs*) = ((*asTerm qX*) #[*x1* ∧ *x2*]-*xs*)
⟨proof⟩

lemma *fresh-asTerm-qFresh*:
assumes *qGood qX*
shows *fresh xs x* (*asTerm qX*) = *qFresh xs x qX*
⟨proof⟩

lemma *skel-asTerm-qSkel*:
assumes *qGood qX*
shows *skel* (*asTerm qX*) = *qSkel qX*
⟨proof⟩

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′*)
⟨proof⟩

lemma *fresh-swap-qFresh-qSwap*:
assumes *good X*
shows *fresh xs x* (*X* #[*y1* ∧ *y2*]-*ys*) = *qFresh xs x* ((*pick X*) #[[*y1* ∧ *y2*]]-*ys*)
⟨proof⟩

5.3.2 For abstractions

lemma *alphaAbsGood-equivP*: *equivP qGoodAbs alphaAbsGood*
⟨proof⟩

lemma *univ-asAbs-alphaAbsGood*[*simp*]:
assumes *fAbs respectsP alphaAbsGood* **and** *qGoodAbs A*
shows *univ fAbs* (*asAbs A*) = *fAbs A*

⟨*proof*⟩

corollary *univ-asAbs-alphaAbs*[*simp*]:
assumes $*$: *fAbs respectsP alphaAbs* **and** $**$: *qGoodAbs A*
shows *univ fAbs* (*asAbs A*) = *fAbs A*
⟨*proof*⟩

lemma *pick-inj-on-goodAbs*: *inj-on pick* (*Collect goodAbs*)
⟨*proof*⟩

lemma *pick-injective-goodAbs*[*simp*]:
⟦*goodAbs A*; *goodAbs B*⟧ ⟹ *pick A* = *pick B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *goodAbs-imp-qGoodAbs-pick*:
goodAbs A ⟹ *qGoodAbs* (*pick A*)
⟨*proof*⟩

lemma *qGoodAbs-iff-goodAbs-asAbs*:
goodAbs(*asAbs qA*) = *qGoodAbs qA*
⟨*proof*⟩

lemma *pick-asAbs*:
assumes *qGoodAbs qA*
shows *pick* (*asAbs qA*) $= *qA*
⟨*proof*⟩

lemma *asAbs-pick*:
assumes *goodAbs A*
shows *asAbs* (*pick A*) = *A*
⟨*proof*⟩

lemma *pick-alphaAbs*: *goodAbs A* ⟹ *pick A* $= *pick A*
⟨*proof*⟩

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

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

lemma *asAbs-equal-iff-alphaAbs*:
assumes *qGoodAbs qA* ∨ *qGoodAbs qB*
shows (*asAbs qA* = *asAbs qB*) = (*qA* $= *qB*)
⟨*proof*⟩

lemma *pick-alphaAbs-iff-equal*:
assumes *goodAbs A* **and** *goodAbs B*
shows (*pick A* $= *pick B*) = (*A* = *B*)
⟨*proof*⟩

lemma *pick-swapAbs-qSwapAbs*:
assumes *goodAbs A*
shows *pick* (*A* $[*x1* ∧ *x2*]-*xs*) $= ((*pick A*) $[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *asAbs-qSwapAbs-swapAbs*:
assumes *qGoodAbs qA*
shows *asAbs* (*qA* $[[*x1* ∧ *x2*]]-*xs*) = ((*asAbs qA*) $[*x1* ∧ *x2*]-*xs*)
⟨*proof*⟩

lemma *freshAbs-asAbs-qFreshAbs*:
assumes *qGoodAbs qA*
shows *freshAbs xs x* (*asAbs qA*) = *qFreshAbs xs x qA*
⟨*proof*⟩

lemma *skelAbs-asAbs-qSkelAbs*:
assumes *qGoodAbs qA*
shows *skelAbs* (*asAbs qA*) = *qSkelAbs qA*
⟨*proof*⟩

5.3.3 For inputs

For unbound inputs:

lemma *pickInp-inj-on-goodInp*: *inj-on pickInp* (*Collect goodInp*)
⟨*proof*⟩

lemma *goodInp-imp-qGoodInp-pickInp*:
assumes *goodInp inp*
shows *qGoodInp* (*pickInp inp*)
⟨*proof*⟩

lemma *qGoodInp-iff-goodInp-asInp*:
goodInp (*asInp qinp*) = *qGoodInp qinp*
⟨*proof*⟩

lemma *pickInp-asInp*:
assumes *qGoodInp qinp*
shows *pickInp* (*asInp qinp*) %= *qinp*
⟨*proof*⟩

lemma *asInp-pickInp*:
assumes *goodInp inp*
shows *asInp* (*pickInp inp*) = *inp*

⟨*proof*⟩

lemma *pickInp-alphaInp*:
assumes *goodInp*: *goodInp inp*
shows *pickInp inp* %= *pickInp inp*
⟨*proof*⟩

lemma *alphaInp-imp-asInp-equal*:
assumes *qGoodInp qinp* **and** *qinp* %= *qinp'*
shows *asInp qinp* = *asInp qinp'*
⟨*proof*⟩

lemma *asInp-equal-imp-alphaInp*:
assumes *qGoodInp qinp* **and** *asInp qinp* = *asInp qinp'*
shows *qinp* %= *qinp'*
⟨*proof*⟩

lemma *asInp-equal-iff-alphaInp*:
qGoodInp qinp ⟹ (*asInp qinp* = *asInp qinp'*) = (*qinp* %= *qinp'*)
⟨*proof*⟩

lemma *pickInp-alphaInp-iff-equal*:
assumes *goodInp inp* **and** *goodInp inp'*
shows (*pickInp inp* %= *pickInp inp'*) = (*inp* = *inp'*)
⟨*proof*⟩

lemma *pickInp-swapInp-qSwapInp*:
assumes *goodInp inp*
shows *pickInp* (*inp* %[*x1* ∧ *x2*]-*xs*) %= ((*pickInp inp*) %[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *asInp-qSwapInp-swapInp*:
assumes *qGoodInp qinp*
shows *asInp* (*qinp* %[[*x1* ∧ *x2*]]-*xs*) = ((*asInp qinp*) %[*x1* ∧ *x2*]-*xs*)
⟨*proof*⟩

lemma *swapInp-def2*:
(*inp* %[*x1* ∧ *x2*]-*xs*) = *asInp* ((*pickInp inp*) %[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *freshInp-def2*:
freshInp xs x inp = *qFreshInp xs x* (*pickInp inp*)
⟨*proof*⟩

For bound inputs:

lemma *pickBinp-inj-on-goodBinp*: *inj-on pickBinp* (*Collect goodBinp*)
⟨*proof*⟩

lemma *goodBinp-imp-qGoodBinp-pickBinp*:

assumes *goodBinp binp*
shows *qGoodBinp* (*pickBinp binp*)
⟨*proof*⟩

lemma *qGoodBinp-iff-goodBinp-asBinp*:
goodBinp (*asBinp qbinp*) = *qGoodBinp qbinp*
⟨*proof*⟩

lemma *pickBinp-asBinp*:
assumes *qGoodBinp qbinp*
shows *pickBinp* (*asBinp qbinp*) %%= *qbinp*
⟨*proof*⟩

lemma *asBinp-pickBinp*:
assumes *goodBinp binp*
shows *asBinp* (*pickBinp binp*) = *binp*
⟨*proof*⟩

lemma *pickBinp-alphaBinp*:
assumes *goodBinp*: *goodBinp binp*
shows *pickBinp binp* %%= *pickBinp binp*
⟨*proof*⟩

lemma *alphaBinp-imp-asBinp-equal*:
assumes *qGoodBinp qbinp* **and** *qbinp* %%= *qbinp′*
shows *asBinp qbinp* = *asBinp qbinp′*
⟨*proof*⟩

lemma *asBinp-equal-imp-alphaBinp*:
assumes *qGoodBinp qbinp* **and** *asBinp qbinp* = *asBinp qbinp′*
shows *qbinp* %%= *qbinp′*
⟨*proof*⟩

lemma *asBinp-equal-iff-alphaBinp*:
qGoodBinp qbinp ⟹ (*asBinp qbinp* = *asBinp qbinp′*) = (*qbinp* %%= *qbinp′*)
⟨*proof*⟩

lemma *pickBinp-alphaBinp-iff-equal*:
assumes *goodBinp binp* **and** *goodBinp binp′*
shows (*pickBinp binp* %%= *pickBinp binp′*) = (*binp* = *binp′*)
⟨*proof*⟩

lemma *pickBinp-swapBinp-qSwapBinp*:
assumes *goodBinp binp*
shows *pickBinp* (*binp* %%[*x1* ∧ *x2*]-*xs*) %%= ((*pickBinp binp*) %%[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *asBinp-qSwapBinp-swapBinp*:
assumes *qGoodBinp qbinp*

shows *asBinp* (*qbinp* %%[[*x1* ∧ *x2*]]-*xs*) = ((*asBinp qbinp*) %%[*x1* ∧ *x2*]-*xs*)
⟨*proof*⟩

lemma *swapBinp-def2*:
(*binp* %%[*x1* ∧ *x2*]-*xs*) = *asBinp* ((*pickBinp binp*) %%[[*x1* ∧ *x2*]]-*xs*)
⟨*proof*⟩

lemma *freshBinp-def2*:
freshBinp xs x binp = *qFreshBinp xs x* (*pickBinp binp*)
⟨*proof*⟩

5.3.4 For environments

lemma *goodEnv-imp-qGoodEnv-pickE*:
assumes *goodEnv rho*
shows *qGoodEnv* (*pickE rho*)
⟨*proof*⟩

lemma *qGoodEnv-iff-goodEnv-asEnv*:
goodEnv (*asEnv qrho*) = *qGoodEnv qrho*
⟨*proof*⟩

lemma *pickE-asEnv*:
assumes *qGoodEnv qrho*
shows *pickE* (*asEnv qrho*) &= *qrho*
⟨*proof*⟩

lemma *asEnv-pickE*:
assumes *goodEnv rho* **shows** *asEnv* (*pickE rho*) *xs x* = *rho xs x*
⟨*proof*⟩

lemma *pickE-alphaEnv*:
assumes *goodEnv*: *goodEnv rho* **shows** *pickE rho* &= *pickE rho*
⟨*proof*⟩

lemma *alphaEnv-imp-asEnv-equal*:
assumes *qGoodEnv qrho* **and** *qrho* &= *qrho'*
shows *asEnv qrho* = *asEnv qrho'*
⟨*proof*⟩

lemma *asEnv-equal-imp-alphaEnv*:
assumes *qGoodEnv qrho* **and** *asEnv qrho* = *asEnv qrho'*
shows *qrho* &= *qrho'*
⟨*proof*⟩

lemma *asEnv-equal-iff-alphaEnv*:
qGoodEnv qrho ⟹ (*asEnv qrho* = *asEnv qrho'*) = (*qrho* &= *qrho'*)
⟨*proof*⟩

lemma *pickE-alphaEnv-iff-equal*:
assumes *goodEnv rho* **and** *goodEnv rho'*
shows (*pickE rho* &= *pickE rho'*) = (*rho* = *rho'*)
⟨*proof*⟩

lemma *freshEnv-def2*:
freshEnv xs x rho = *qFreshEnv xs x* (*pickE rho*)
⟨*proof*⟩

lemma *pick-psubst-qPsubst*:
assumes *good X* **and** *goodEnv rho*
shows *pick* (*X* #[*rho*]) #= ((*pick X*) #[[*pickE rho*]])
⟨*proof*⟩

lemma *pick-psubstAbs-qPsubstAbs*:
assumes *goodAbs A* **and** *goodEnv rho*
shows *pick* (*A* $[*rho*]) $= ((*pick A*) $[[*pickE rho*]])
⟨*proof*⟩

lemma *pickInp-psubstInp-qPsubstInp*:
assumes *good*: *goodInp inp* **and** *good-rho*: *goodEnv rho*
shows *pickInp* (*inp* %[*rho*]) %= ((*pickInp inp*) %[[*pickE rho*]])
⟨*proof*⟩

lemma *pickBinp-psubstBinp-qPsubstBinp*:
assumes *good*: *goodBinp binp* **and** *good-rho*: *goodEnv rho*
shows *pickBinp* (*binp* %%[*rho*]) %%= ((*pickBinp binp*) %%[[*pickE rho*]])
⟨*proof*⟩

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

lemma *pick-Var-qVar*:
pick (*Var xs x*) #= *qVar xs x*
⟨*proof*⟩

lemma *Op-asInp-asTerm-qOp*:
assumes *qGoodInp qinp* **and** *qGoodBinp qbinp*
shows *Op delta* (*asInp qinp*) (*asBinp qbinp*) = *asTerm* (*qOp delta qinp qbinp*)
⟨*proof*⟩

lemma *qOp-pickInp-pick-Op*:
assumes *goodInp inp* **and** *goodBinp binp*
shows *qOp delta* (*pickInp inp*) (*pickBinp binp*) #= *pick* (*Op delta inp binp*)
⟨*proof*⟩

lemma *Abs-asTerm-asAbs-qAbs*:
assumes *qGood qX*
shows *Abs xs x* (*asTerm qX*) = *asAbs* (*qAbs xs x qX*)

⟨*proof*⟩

lemma *qAbs-pick-Abs*:
assumes *good X*
shows *qAbs xs x* (*pick X*) $= *pick* (*Abs xs x X*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Op-preserves-good*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows *good*(*Op delta inp binp*)
⟨*proof*⟩

lemma *Abs-preserves-good*[*simp*]:
assumes *good*: *good X*
shows *goodAbs*(*Abs xs x X*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *swapAbs-preserves-good*[*simp*]:
assumes *goodAbs A*
shows *goodAbs* (*A* $[*x* ∧ *y*]-*xs*)
⟨*proof*⟩

lemma *swapInp-preserves-good*[*simp*]:
assumes *goodInp inp*
shows *goodInp* (*inp* %[*x* ∧ *y*]-*xs*)
⟨*proof*⟩

lemma *swapBinp-preserves-good*[*simp*]:
assumes *goodBinp binp*
shows *goodBinp* (*binp* %%[*x* ∧ *y*]-*xs*)

⟨*proof*⟩

lemma *swapEnvDom-preserves-good*:
assumes *goodEnv rho*
shows *goodEnv* (*swapEnvDom xs x y rho*) (**is** *goodEnv ?rho′*)
⟨*proof*⟩

lemma *swapEnvIm-preserves-good*:
assumes *goodEnv rho*
shows *goodEnv* (*swapEnvIm xs x y rho*)
⟨*proof*⟩

lemma *swapEnv-preserves-good*[*simp*]:
assumes *goodEnv rho*
shows *goodEnv* (*rho* &[*x* ∧ *y*]-*xs*)
⟨*proof*⟩

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*])
⟨*proof*⟩

lemma *psubstAbs-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *goodAbs-A*: *goodAbs A*
shows *goodAbs* (*A* $[*rho*])
⟨*proof*⟩

lemma *psubstInp-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *good*: *goodInp inp*
shows *goodInp* (*inp* %[*rho*])
⟨*proof*⟩

lemma *psubstBinp-preserves-good*[*simp*]:
assumes *good-rho*: *goodEnv rho* **and** *good*: *goodBinp binp*
shows *goodBinp* (*binp* %%[*rho*])
⟨*proof*⟩

lemma *psubstEnv-preserves-good*[*simp*]:
assumes *good*: *goodEnv rho* **and** *good′*: *goodEnv rho′*
shows *goodEnv* (*rho* &[*rho′*])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *updEnv-preserves-good*[*simp*]:
assumes *good-X*: *good X* **and** *good-rho*: *goodEnv rho*
shows *goodEnv* (*rho* [$x \leftarrow X$]-*xs*)
⟨*proof*⟩

lemma *getEnv-preserves-good*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *Some X*
shows *good X*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *substAbs-preserves-good*[*simp*]:
assumes *good X* **and** *goodAbs A*
shows *goodAbs* (*A* $[*X* / *x*]-*xs*)
⟨*proof*⟩

lemma *substInp-preserves-good*[*simp*]:
assumes *good X* **and** *goodInp inp*
shows *goodInp* (*inp* %[*X* / *x*]-*xs*)
⟨*proof*⟩

lemma *substBinp-preserves-good*[*simp*]:
assumes *good X* **and** *goodBinp binp*
shows *goodBinp* (*binp* %%[*X* / *x*]-*xs*)
⟨*proof*⟩

lemma *substEnv-preserves-good*[*simp*]:
assumes *good X* **and** *goodEnv rho*
shows *goodEnv* (*rho* &[*X* / *x*]-*xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubstAbs-preserves-good*[*simp*]:
assumes *goodAbs A*
shows *goodAbs* (*A* $[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstInp-preserves-good*[*simp*]:
assumes *goodInp inp*
shows *goodInp* (*inp* %[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstBinp-preserves-good*[*simp*]:
assumes *goodBinp binp*
shows *goodBinp* (*binp* %%[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstEnv-preserves-good*[*simp*]:
assumes *goodEnv rho*
shows *goodEnv* (*rho* &[*x1* // *x*]-*xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Op-inj*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*

and *goodInp inp'* **and** *goodBinp binp'*
shows
$(Op\ delta\ inp\ binp = Op\ delta'\ inp'\ binp') =$
$(delta = delta' \wedge inp = inp' \wedge binp = binp')$
⟨*proof*⟩

“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*⟩

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*⟩

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*: $(X\ \#[y \wedge x]\text{-}xs) = (X'\ \#[y \wedge x']\text{-}xs)$
shows $Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$
⟨*proof*⟩

lemma *Abs-swap-fresh*:
assumes *good-X*: *good X* **and** *fresh*: *fresh xs x' X*
shows $Abs\ xs\ x\ X = Abs\ xs\ x'\ (X\ \#[x' \wedge x]\text{-}xs)$
⟨*proof*⟩

lemma *Var-diff-Op*[*simp*]:
$Var\ xs\ x \neq Op\ delta\ inp\ binp$
⟨*proof*⟩

lemma *Op-diff-Var*[*simp*]:
$Op\ delta\ inp\ binp \neq Var\ xs\ x$
⟨*proof*⟩

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*⟩

theorem *abs-nchotomy*:
assumes *goodAbs A*
shows ∃ *xs x X*. *good X* ∧ *A* = *Abs xs x X*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *freshAbs-simp*[*simp*]:
assumes *good X*
shows *freshAbs ys y* (*Abs xs x X*) ⟷ (*ys* = *xs* ∧ *y* = *x* ∨ *fresh ys y X*)
⟨*proof*⟩

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

theorem *skel-Var-simp*[*simp*]:
skel (*Var xs x*) = *Branch Map.empty Map.empty*
⟨*proof*⟩

lemma *skel-Op-simp*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows *skel* (*Op delta inp binp*) = *Branch* (*skelInp inp*) (*skelBinp binp*)
⟨*proof*⟩

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

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*⟩

corollary *psubst-Var-simp1*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *None*
shows ((*Var xs x*) #[*rho*]) = *Var xs x*
⟨*proof*⟩

corollary *psubst-Var-simp2*[*simp*]:
assumes *goodEnv rho* **and** *rho xs x* = *Some X*
shows ((*Var xs x*) #[*rho*]) = *X*
⟨*proof*⟩

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*⟩

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*⟩

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

theorem *getEnv-idEnv*[*simp*]: *idEnv xs x* = *None*
⟨*proof*⟩

lemma *getEnv-updEnv*[*simp*]:
(*rho* [*x* ← *X*]-*xs*) *ys y* = (*if ys* = *xs* ∧ *y* = *x then Some X else rho ys y*)
⟨*proof*⟩

theorem *getEnv-updEnv1*:
ys ≠ *xs* ∨ *y* ≠ *x* ⟹ (*rho* [*x* ← *X*]-*xs*) *ys y* = *rho ys y*
⟨*proof*⟩

theorem *getEnv-updEnv2*:
(*rho* [*x* ← *X*]-*xs*) *xs x* = *Some X*
⟨*proof*⟩

lemma *subst-Var-simp1*[*simp*]:
assumes *good Y*
and *ys* ≠ *xs* ∨ *y* ≠ *x*
shows ((*Var xs x*) #[*Y* / *y*]-*ys*) = *Var xs x*
⟨*proof*⟩

lemma *subst-Var-simp2*[*simp*]:
assumes *good Y*
shows ((*Var xs x*) #[*Y* / *x*]-*xs*) = *Y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubstAbs-simp*[*simp*]:
assumes *good 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*)
⟨*proof*⟩

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 ⟹ |{*x*. ¬ *fresh xs x X*}| <*o* |*UNIV* :: '*var set*|
⟨*proof*⟩

lemma *single-non-freshAbs-ordLess-var*:
goodAbs A ⟹ |{*x*. ¬ *freshAbs xs x A*}| <*o* |*UNIV* :: '*var set*|
⟨*proof*⟩

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*| ∨ *finite V* **and** *XSvar*: |*XS*| <*o* |*UNIV* :: '*var set*| ∨ *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*⟩

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*: $|V| <o\ |UNIV :: 'var\ set| \vee finite\ V$ **and**
XSvar: $|XS| <o\ |UNIV :: 'var\ set| \vee finite\ XS$ **and**
ASvar: $|AS| <o\ |UNIV :: 'var\ set| \vee finite\ AS$ **and**
Rhovar: $|Rho| <o\ |UNIV :: 'var\ set| \vee finite\ Rho$ **and**
good: $\forall\ X \in XS.\ good\ X$ **and**
ASGood: $\forall\ A \in AS.\ goodAbs\ A$ **and**
RhoGood: $\forall\ rho \in Rho.\ goodEnv\ rho$
shows
$\exists\ z.\ z \notin V \wedge$
$(\forall\ X \in XS.\ fresh\ xs\ z\ X) \wedge$
$(\forall\ A \in AS.\ freshAbs\ xs\ z\ A) \wedge$
$(\forall\ rho \in Rho.\ freshEnv\ xs\ z\ rho)$
⟨*proof*⟩

5.5.3 Compositionality

lemma *swap-ident*[*simp*]:
assumes *good X*
shows $(X\ \#[x \wedge x]\text{-}xs) = X$
⟨*proof*⟩

lemma *swap-compose*:
assumes *good-X*: *good X*
shows $((X\ \#[x \wedge y]\text{-}zs)\ \#[x' \wedge y']\text{-}zs') =$
$((X\ \#[x' \wedge y']\text{-}zs')\ \#[(x\ @zs[x' \wedge y']\text{-}zs') \wedge (y\ @zs[x' \wedge y']\text{-}zs')]\text{-}zs)$
⟨*proof*⟩

lemma *swap-commute*:
$[\![good\ X;\ zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\}]\!] \Longrightarrow$
$((X\ \#[x \wedge y]\text{-}zs)\ \#[x' \wedge y']\text{-}zs') = ((X\ \#[x' \wedge y']\text{-}zs')\ \#[x \wedge y]\text{-}zs)$
⟨*proof*⟩

lemma *swap-involutive*[*simp*]:
assumes *good-X*: *good X*
shows $((X\ \#[x \wedge y]\text{-}zs)\ \#[x \wedge y]\text{-}zs) = X$

⟨*proof*⟩

theorem *swap-sym*: $(X \#[x \wedge y]\text{-}zs) = (X \#[y \wedge x]\text{-}zs)$
⟨*proof*⟩

lemma *swap-involutive2*[*simp*]:
assumes *good X*
shows $((X \#[x \wedge y]\text{-}zs) \#[y \wedge x]\text{-}zs) = X$
⟨*proof*⟩

lemma *swap-preserves-fresh*[*simp*]:
assumes *good X*
shows *fresh xs* $(x\ @xs[y1 \wedge y2]\text{-}ys)$ $(X \#[y1 \wedge y2]\text{-}ys)$ = *fresh xs x X*
⟨*proof*⟩

lemma *swap-preserves-fresh-distinct*:
assumes *good X* **and**
$xs \neq ys \vee x \notin \{y1,y2\}$
shows *fresh xs x* $(X \#[y1 \wedge y2]\text{-}ys)$ = *fresh xs x X*
⟨*proof*⟩

lemma *fresh-swap-exchange1*:
assumes *good X*
shows *fresh xs x2* $(X \#[x1 \wedge x2]\text{-}xs)$ = *fresh xs x1 X*
⟨*proof*⟩

lemma *fresh-swap-exchange2*:
assumes *good X* **and** $\{x1,x2\} \subseteq$ *var xs*
shows *fresh xs x2* $(X \#[x2 \wedge x1]\text{-}xs)$ = *fresh xs x1 X*
⟨*proof*⟩

lemma *fresh-swap-id*[*simp*]:
assumes *good X* **and** *fresh xs x1 X fresh xs x2 X*
shows $(X \#[x1 \wedge x2]\text{-}xs) = X$
⟨*proof*⟩

lemma *freshAbs-swapAbs-id*[*simp*]:
assumes *goodAbs A freshAbs xs x1 A freshAbs xs x2 A*
shows $(A\ \$[x1 \wedge x2]\text{-}xs) = A$
⟨*proof*⟩

lemma *fresh-swap-compose*:
assumes *good X fresh xs y X fresh xs z X*
shows $((X \#[y \wedge x]\text{-}xs) \#[z \wedge y]\text{-}xs) = (X \#[z \wedge x]\text{-}xs)$
⟨*proof*⟩

lemma *skel-swap*:

assumes *good X*
shows *skel* (*X* #[*x1* ∧ *x2*]-*xs*) = *skel X*
⟨*proof*⟩

5.5.4 Compositionality for environments

lemma *swapEnv-ident*[*simp*]:
assumes *goodEnv rho*
shows (*rho* &[*x* ∧ *x*]-*xs*) = *rho*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *swapEnv-involutive*[*simp*]:
assumes *goodEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*x* ∧ *y*]-*zs*) = *rho*
⟨*proof*⟩

theorem *swapEnv-sym*: (*rho* &[*x* ∧ *y*]-*zs*) = (*rho* &[*y* ∧ *x*]-*zs*)
⟨*proof*⟩

lemma *swapEnv-involutive2*[*simp*]:
assumes *good*: *goodEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*y* ∧ *x*]-*zs*) = *rho*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *freshEnv-swapEnv-exchange1*:
assumes *goodEnv rho*
shows *freshEnv xs x2* (*rho* &[*x1* ∧ *x2*]-*xs*) = *freshEnv xs x1 rho*

⟨*proof*⟩

lemma *freshEnv-swapEnv-exchange2*:
assumes *goodEnv rho*
shows *freshEnv xs x2* (*rho* &[*x2* ∧ *x1*]-*xs*) = *freshEnv xs x1 rho*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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* ∧ *y*]-*zs*) ∈ *swapped*
⟨*proof*⟩

lemma *swapped-preserves-good*:
assumes *good X* **and** (X, Y) ∈ *swapped*
shows *good Y*
⟨*proof*⟩

lemma *swapped-skel*:
assumes *good X* **and** (X, Y) ∈ *swapped*
shows *skel Y* = *skel X*
⟨*proof*⟩

lemma *obtain-rep*:
assumes *GOOD*: *good X* **and** *FRESH*: *fresh xs x′ X*
shows ∃ *X′*. (*X*,*X′*) ∈ *swapped* ∧ *good X′* ∧ *Abs xs x X* = *Abs xs x′ X′*
⟨*proof*⟩

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*⟩

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*⟧
$\Longrightarrow$ *phi* (*Op delta inp binp*) **and**
Abs: $\bigwedge$ *xs x X*. ⟦*good* X; *phi* X⟧ $\Longrightarrow$ *phiAbs* (*Abs xs x X*)
shows (*good* $X \longrightarrow$ *phi* X) $\wedge$ (*goodAbs* $A \longrightarrow$ *phiAbs* A)
⟨*proof*⟩

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*⟧
$\Longrightarrow$ *phi* (*Op delta inp binp*) **and**
Abs: $\bigwedge$ *xs x X*. ⟦*good* X;
$\bigwedge$ Y. $(X,Y) \in$ *swapped* $\Longrightarrow$ *phi* Y;
$\bigwedge$ Y. ⟦*good* Y; *skel* Y = *skel* X⟧ $\Longrightarrow$ *phi* Y⟧
$\Longrightarrow$ *phiAbs* (*Abs xs x X*)
shows (*good* $X \longrightarrow$ *phi* X) $\wedge$ (*goodAbs* $A \longrightarrow$ *phiAbs* A)
⟨*proof*⟩

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* $\Rightarrow$ '*varSort* $\Rightarrow$ '*var set*
assumes

PAR: $\bigwedge$ *p xs*. (|*varsOf xs p*| <*o* |*UNIV*::'*var set*|) **and**
var: $\bigwedge$ *xs x p*. *phi* (*Var xs x*) *p* **and**
op: $\bigwedge$ *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: $\bigwedge$ *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*⟩

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:
$\bigwedge$ *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: $\bigwedge$ *X Y*. ⟦*good X*; (*X*,*Y*) ∈ *rel*⟧ ⟹ *good Y* ∧ *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*⟧
⟹ *phi* (*Op delta inp binp*) **and**
abs:
$\bigwedge$ *xs x X*.
⟦*good X*;
x ∉ *vars xs*;
$\bigwedge$ *Y*. *Y* ∈ *terms* ⟹ *fresh xs x Y*;
$\bigwedge$ *A*. *A* ∈ *abs* ⟹ *freshAbs xs x A*;
$\bigwedge$ *rho*. *rho* ∈ *envs* ⟹ *freshEnv xs x rho*;
$\bigwedge$ *Y*. (*X*,*Y*) ∈ *rel* ⟹ *phi Y*⟧
⟹ *phiAbs* (*Abs xs x X*)
shows
(*good X* ⟶ *phi X*) ∧
(*goodAbs A* ⟶ *phiAbs A*)
⟨*proof*⟩

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: ⋀ *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*;
phi X⟧
⟹ *phiAbs* (*Abs xs x X*)
shows
(*good X* ⟶ *phi X*) ∧
(*goodAbs A* ⟶ *phiAbs A*)
⟨*proof*⟩

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: ⋀ *xs x*. *phi* (*Var xs x*) **and**
Op:
⋀ *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*⟩

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* ← *X*]-*xs*) *ys y* = (*if* (*ys* = *xs* $\wedge$ *y* = *x*) *then Some X else None*)
⟨*proof*⟩

theorem *subst-psubst-idEnv*:
(*X* #[*Y* / *y*]-*ys*) = (*X* #[*idEnv* [*y* ← *Y*]-*ys*])
⟨*proof*⟩

theorem *vsubst-psubst-idEnv*:

(*X* #[*z* // *y*]-*ys*) = (*X* #[*idEnv* [*y* ← *Var ys z*]-*ys*])
⟨*proof*⟩

theorem *substEnv-psubstEnv-idEnv*:
(*rho* &[*Y* / *y*]-*ys*) = (*rho* &[*idEnv* [*y* ← *Y*]-*ys*])
⟨*proof*⟩

theorem *vsubstEnv-psubstEnv-idEnv*:
(*rho* &[*z* // *y*]-*ys*) = (*rho* &[*idEnv* [*y* ← *Var ys z*]-*ys*])
⟨*proof*⟩

theorem *freshEnv-idEnv*: *freshEnv xs x idEnv*
⟨*proof*⟩

theorem *swapEnv-idEnv*[*simp*]: (*idEnv* &[*x* ∧ *y*]-*xs*) = *idEnv*
⟨*proof*⟩

theorem *psubstEnv-idEnv*[*simp*]: (*idEnv* &[*rho*]) = *rho*
⟨*proof*⟩

theorem *substEnv-idEnv*: (*idEnv* &[*X* / *x*]-*xs*) = (*idEnv* [*x* ← *X*]-*xs*)
⟨*proof*⟩

theorem *vsubstEnv-idEnv*: (*idEnv* &[*y* // *x*]-*xs*) = (*idEnv* [*x* ← (*Var xs y*)]-*xs*)
⟨*proof*⟩

lemma *psubstAll-idEnv*:
fixes *X*::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*term* **and**
A::(*'index*,*'bindex*,*'varSort*,*'var*,*'opSym*)*abs*
shows
(*good X* ⟶ (*X* #[*idEnv*]) = *X*) ∧
(*goodAbs A* ⟶ (*A* $[*idEnv*]) = *A*)
⟨*proof*⟩

lemma *psubst-idEnv*[*simp*]:
good X ⟹ (*X* #[*idEnv*]) = *X*
⟨*proof*⟩

lemma *psubstEnv-idEnv-id*[*simp*]:
assumes *goodEnv rho*
shows (*rho* &[*idEnv*]) = *rho*
⟨*proof*⟩

6.2 Environment update versus other operators

theorem *updEnv-overwrite*[*simp*]: ((*rho* [*x* ← *X*]-*xs*) [*x* ← *X'*]-*xs*) = (*rho* [*x* ← *X'*]-*xs*)
⟨*proof*⟩

theorem *updEnv-commute*:
assumes $xs \neq ys \vee x \neq y$
shows $((rho\ [x \leftarrow X]\text{-}xs)\ [y \leftarrow Y]\text{-}ys) = ((rho\ [y \leftarrow Y]\text{-}ys)\ [x \leftarrow X]\text{-}xs)$
⟨*proof*⟩

theorem *freshEnv-updEnv-E1*:
assumes $freshEnv\ xs\ y\ (rho\ [x \leftarrow X]\text{-}xs)$
shows $y \neq x$
⟨*proof*⟩

theorem *freshEnv-updEnv-E2*:
assumes $freshEnv\ ys\ y\ (rho\ [x \leftarrow X]\text{-}xs)$
shows $fresh\ ys\ y\ X$
⟨*proof*⟩

theorem *freshEnv-updEnv-E3*:
assumes $freshEnv\ ys\ y\ (rho\ [x \leftarrow X]\text{-}xs)$
shows $rho\ ys\ y = None$
⟨*proof*⟩

theorem *freshEnv-updEnv-E4*:
assumes $freshEnv\ ys\ y\ (rho\ [x \leftarrow X]\text{-}xs)$
and $zs \neq xs \vee z \neq x$ **and** $rho\ zs\ z = Some\ Z$
shows $fresh\ ys\ y\ Z$
⟨*proof*⟩

theorem *freshEnv-updEnv-I*:
assumes $ys \neq xs \vee y \neq x$ **and** $fresh\ ys\ y\ X$ **and** $rho\ ys\ y = None$
and $\bigwedge\ zs\ z\ Z.\ [\![zs \neq xs \vee z \neq x;\ rho\ zs\ z = Some\ Z]\!] \Longrightarrow fresh\ ys\ y\ Z$
shows $freshEnv\ ys\ y\ (rho\ [x \leftarrow X]\text{-}xs)$
⟨*proof*⟩

theorem *swapEnv-updEnv*:
$((rho\ [x \leftarrow X]\text{-}xs)\ \&[y1 \wedge y2]\text{-}ys) =$
$((rho\ \&[y1 \wedge y2]\text{-}ys)\ [(x\ @xs[y1 \wedge y2]\text{-}ys) \leftarrow (X\ \#[y1 \wedge y2]\text{-}ys)]\text{-}xs)$
⟨*proof*⟩

lemma *swapEnv-updEnv-fresh*:
assumes $ys \neq xs \vee x \notin \{y1, y2\}$ **and** $good\ X$
and $fresh\ ys\ y1\ X$ **and** $fresh\ ys\ y2\ X$
shows $((rho\ [x \leftarrow X]\text{-}xs)\ \&[y1 \wedge y2]\text{-}ys) =$
$((rho\ \&[y1 \wedge y2]\text{-}ys)\ [x \leftarrow X]\text{-}xs)$
⟨*proof*⟩

theorem *psubstEnv-updEnv*:
$((rho\ [x \leftarrow X]\text{-}xs)\ \&[rho']) = ((rho\ \&[rho'])\ [x \leftarrow (X\ \#[rho'])]\text{-}xs)$
⟨*proof*⟩

theorem *psubstEnv-updEnv-idEnv*:

((*idEnv* [*x* ← *X*]-*xs*) &[*rho*]) = (*rho* [*x* ← (*X* #[*rho*])]-*xs*)
⟨*proof*⟩

theorem *substEnv-updEnv*:
((*rho* [*x* ← *X*]-*xs*) &[*Y* / *y*]-*ys*) = ((*rho* &[*Y* / *y*]-*ys*) [*x* ← (*X* #[*Y* / *y*]-*ys*)]-*xs*)
⟨*proof*⟩

theorem *vsubstEnv-updEnv*:
((*rho* [*x* ← *X*]-*xs*) &[*y1* // *y*]-*ys*) = ((*rho* &[*y1* // *y*]-*ys*) [*x* ← (*X* #[*y1* // *y*]-*ys*)]-*xs*)
⟨*proof*⟩

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 ⋀ *xs x*. *rho xs x* = *rho′ xs x*
shows *rho* = *rho′*
⟨*proof*⟩

theorem *freshEnv-getEnv1*[*simp*]:
⟦*freshEnv ys y rho*; *rho xs x* = *Some X*⟧ ⟹ *ys* ≠ *xs* ∨ *y* ≠ *x*
⟨*proof*⟩

theorem *freshEnv-getEnv2*[*simp*]:
⟦*freshEnv ys y rho*; *rho xs x* = *Some X*⟧ ⟹ *fresh ys y X*
⟨*proof*⟩

theorem *freshEnv-getEnv*[*simp*]:
freshEnv ys y rho ⟹ *rho ys y* = *None*
⟨*proof*⟩

theorem *getEnv-swapEnv1*[*simp*]:
assumes *rho xs* (*x* @*xs* [*z1* ∧ *z2*]-*zs*) = *None*
shows (*rho* &[*z1* ∧ *z2*]-*zs*) *xs x* = *None*
⟨*proof*⟩

theorem *getEnv-swapEnv2*[*simp*]:
assumes *rho xs* (*x* @*xs* [*z1* ∧ *z2*]-*zs*) = *Some X*
shows (*rho* &[*z1* ∧ *z2*]-*zs*) *xs x* = *Some* (*X* #[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

theorem *getEnv-psubstEnv-None*[*simp*]:
assumes *rho xs x* = *None*

shows (*rho* &[*rho'*]) *xs x* = *rho' xs x*
⟨*proof*⟩

theorem *getEnv-psubstEnv-Some*[*simp*]:
assumes *rho xs x* = *Some X*
shows (*rho* &[*rho'*]) *xs x* = *Some* (*X* #[*rho'*])
⟨*proof*⟩

theorem *getEnv-substEnv1*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *None*
shows (*rho* &[*Y* / *y*]-*ys*) *xs x* = *None*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *getEnv-substEnv3*[*simp*]:
⟦*ys* ≠ *xs* ∨ *y* ≠ *x*; *freshEnv xs x rho*⟧
⟹ (*rho* &[*Y* / *y*]-*ys*) *xs x* = *None*
⟨*proof*⟩

theorem *getEnv-substEnv4*[*simp*]:
freshEnv ys y rho ⟹ (*rho* &[*Y* / *y*]-*ys*) *ys y* = *Some Y*
⟨*proof*⟩

theorem *getEnv-vsubstEnv1*[*simp*]:
assumes *ys* ≠ *xs* ∨ *y* ≠ *x* **and** *rho xs x* = *None*
shows (*rho* &[*y1* // *y*]-*ys*) *xs x* = *None*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *getEnv-vsubstEnv3*[*simp*]:
⟦*ys* ≠ *xs* ∨ *y* ≠ *x*; *freshEnv xs x rho*⟧
⟹ (*rho* &[*z* // *y*]-*ys*) *xs x* = *None*
⟨*proof*⟩

theorem *getEnv-vsubstEnv4*[*simp*]:
freshEnv ys y rho ⟹ (*rho* &[*z* // *y*]-*ys*) *ys y* = *Some* (*Var ys z*)
⟨*proof*⟩

6.4 Substitution versus other operators

definition *freshImEnvAt* ::

$'varSort \Rightarrow 'var \Rightarrow ('index,'bindex,'varSort,'var,'opSym)env \Rightarrow 'varSort \Rightarrow 'var \Rightarrow bool$
where
freshImEnvAt xs x rho ys y ==
 $rho\ ys\ y = None \wedge (ys \neq xs \vee y \neq x) \vee$
 $(\exists\ Y.\ rho\ ys\ y = Some\ Y \wedge fresh\ xs\ x\ Y)$

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
$(good\ X \longrightarrow z \in varsOf\ P \longrightarrow$
 $(\forall\ rho \in envsOf\ P.$
 $fresh\ zs\ z\ (X\ \#[rho]) =$
 $(\forall\ ys.\ \forall\ y.\ fresh\ ys\ y\ X \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)))$
 $\wedge$
 $(goodAbs\ A \longrightarrow z \in varsOf\ P \longrightarrow$
 $(\forall\ rho \in envsOf\ P.$
 $freshAbs\ zs\ z\ (A\ \$[rho]) =$
 $(\forall\ ys.\ \forall\ y.\ freshAbs\ ys\ y\ A \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)))$
⟨*proof*⟩

corollary *fresh-psubst*:
assumes *good X* **and** *goodEnv rho*
shows
$fresh\ zs\ z\ (X\ \#[rho]) =$
 $(\forall\ ys\ y.\ fresh\ ys\ y\ X \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)$
⟨*proof*⟩

corollary *fresh-psubst-E1*:
assumes *good X* **and** *goodEnv rho*
and $rho\ ys\ y = None$ **and** $fresh\ zs\ z\ (X\ \#[rho])$
shows $fresh\ ys\ y\ X \vee (ys \neq zs \vee y \neq z)$
⟨*proof*⟩

corollary *fresh-psubst-E2*:
assumes *good X* **and** *goodEnv rho*
and $rho\ ys\ y = Some\ Y$ **and** $fresh\ zs\ z\ (X\ \#[rho])$
shows $fresh\ ys\ y\ X \vee fresh\ zs\ z\ Y$
⟨*proof*⟩

corollary *fresh-psubst-I1*:
assumes *good X* **and** *goodEnv rho*
and *fresh zs z X* **and** *freshEnv zs z rho*
shows $fresh\ zs\ z\ (X\ \#[rho])$
⟨*proof*⟩

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'*])
⟨*proof*⟩

corollary *fresh-psubst-I*:
assumes *good X* **and** *goodEnv 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*])
⟨*proof*⟩

lemma *fresh-subst*:
assumes *good X* **and** *good 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*))
⟨*proof*⟩

lemma *fresh-vsubst*:
assumes *good 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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ≠ *ys* ∨ *z*
≠ *y*
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
⟨*proof*⟩

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* ≠ *ys* ∨ *z*
≠ *y*
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
⟨*proof*⟩

lemma *vsubst-preserves-fresh*:
assumes *good X*
and *fresh zs z X* **and** *zs* ≠ *ys* ∨ *z* ≠ *y1*
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*)

⟨proof⟩

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*)
⟨proof⟩

lemma *fresh-fresh-subst*[*simp*]:
assumes *good Y* **and** *good X*
and *fresh ys y Y*
shows *fresh ys y* (*X* #[*Y* / *y*]-*ys*)
⟨proof⟩

lemma *diff-fresh-vsubst*[*simp*]:
assumes *good X*
and $y \neq y1$
shows *fresh ys y* (*X* #[*y1* // *y*]-*ys*)
⟨proof⟩

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*
⟨proof⟩

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*
⟨proof⟩

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*
⟨proof⟩

lemma *fresh-vsubst-E2*:
assumes *good X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
shows *fresh ys y X* $\vee\ zs \neq ys \vee z \neq y1$
⟨proof⟩

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*⟩

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*′])
⟨*proof*⟩

lemma *fresh-psubst-updEnv*:
assumes *good X* **and** *good Y* **and** *goodEnv rho*
and *fresh xs x Y*
shows (Y #[*rho* [$x \leftarrow X$]-*xs*]) = (Y #[*rho*])
⟨*proof*⟩

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* $X \longrightarrow$
  ($\forall$ *rho* $\in$ *envsOf P*.
   ($\forall$ *zs z*. *freshEnv zs z rho* $\vee$ *fresh zs z X*)
   $\longrightarrow$ (X #[*rho*]) = X))
 $\wedge$
 (*goodAbs* $A \longrightarrow$
  ($\forall$ *rho* $\in$ *envsOf P*.
   ($\forall$ *zs z*. *freshEnv zs z rho* $\vee$ *freshAbs zs z A*)
   $\longrightarrow$ (A $[*rho*]) = A))
⟨*proof*⟩

corollary *freshEnv-psubst-ident*[*simp*]:
fixes X :: (′*index*,′*bindex*,′*varSort*,′*var*,′*opSym*)*term*
assumes *good X* **and** *goodEnv rho*
and $\bigwedge$ *zs z*. *freshEnv zs z rho* $\vee$ *fresh zs z X*
shows (X #[*rho*]) = X
⟨*proof*⟩

lemma *fresh-subst-ident*[*simp*]:
assumes *good X* **and** *good Y* **and** *fresh xs x Y*
shows (*Y* #[*X* / *x*]-*xs*) = *Y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fresh-vsubst-ident*[*simp*]:
assumes *good Y* **and** *fresh xs x Y*
shows (*Y* #[*x1* // *x*]-*xs*) = *Y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *swap-psubst*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*z1* ∧ *z2*]-*zs*) = ((*X* #[*z1* ∧ *z2*]-*zs*) #[*rho* &[*z1* ∧ *z2*]-*zs*])
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *swapEnv-psubstEnv*:
assumes *goodEnv rho* **and** *goodEnv rho′*
shows ((*rho* &[*rho′*]) &[*z1* ∧ *z2*]-*zs*) = ((*rho* &[*z1* ∧ *z2*]-*zs*) &[*rho′* &[*z1* ∧ *z2*]-*zs*])

⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

corollary *psubst-compose*:
assumes *good X* **and** *goodEnv rho* **and** *goodEnv rho′*
shows ((*X* #[*rho*]) #[*rho′*]) = (*X* #[(*rho* &[*rho′*])])
⟨*proof*⟩

lemma *psubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho′* **and** *goodEnv rho′′*
shows ((*rho* &[*rho′*]) &[*rho′′*]) = (*rho* &[(*rho′* &[*rho′′*])])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

lemma *psubst-vsubst-compose*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*y1* // *y*]-*ys*) #[*rho*]) = (*X* #[(*rho* [*y* ← ((*Var ys y1*) #[*rho*])]-*ys*)])
⟨*proof*⟩

lemma *psubstEnv-vsubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho′*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*rho′*]) = (*rho* &[(*rho′* [*y* ← ((*Var ys y1*) #[*rho′*])]-*ys*)])
⟨*proof*⟩

lemma *subst-psubst-compose*:
assumes *good X* **and** *good Y* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*Y* / *y*]-*ys*) = (*X* #[(*rho* &[*Y* / *y*]-*ys*)])
⟨*proof*⟩

lemma *substEnv-psubstEnv-compose*:
assumes *goodEnv rho* **and** *good Y* **and** *goodEnv rho′*
shows ((*rho* &[*rho′*]) &[*Y* / *y*]-*ys*) = (*rho* &[(*rho′* &[*Y* / *y*]-*ys*)])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubst-psubst-compose*:
assumes *good X* **and** *goodEnv rho*
shows ((*X* #[*rho*]) #[*y1* // *y*]-*ys*) = (*X* #[(*rho* &[*y1* // *y*]-*ys*)])
⟨*proof*⟩

lemma *vsubstEnv-psubstEnv-compose*:
assumes *goodEnv rho* **and** *goodEnv rho'*
shows ((*rho* &[*rho'*]) &[*y1* // *y*]-*ys*) = (*rho* &[(*rho'* &[*y1* // *y*]-*ys*)])
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *subst-vsubst-compose1*:
assumes *good X* **and** *good Y* **and** $y \neq y1$
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y*]-*ys*) = (*X* #[*y1* // *y*]-*ys*)
⟨*proof*⟩

lemma *substEnv-vsubstEnv-compose1*:
assumes *goodEnv rho* **and** *good Y* **and** $y \neq y1$
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Y* / *y*]-*ys*) = (*rho* &[*y1* // *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubst-compose1*:
assumes *good X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y*]-*ys*) = (*X* #[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
⟨*proof*⟩

lemma *vsubstEnv-compose1*:

assumes *goodEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*y2* // *y*]-*ys*) = (*rho* &[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubst-compose2*:
assumes *good X*
and *ys* ≠ *zs* ∨ *y* ∉ {*z*,*z1*}

shows $((X\ \#[y1\ //\ y]\text{-}ys)\ \#[z1\ //\ z]\text{-}zs) =$
$((X\ \#[z1\ //\ z]\text{-}zs)\ \#[(y1\ @ys[z1\ /\ z]\text{-}zs)\ //\ y]\text{-}ys)$
⟨*proof*⟩

lemma *vsubstEnv-compose2*:
assumes *goodEnv rho*
and $ys \neq zs \vee y \notin \{z,z1\}$
shows $((rho\ \&[y1\ //\ y]\text{-}ys)\ \&[z1\ //\ z]\text{-}zs) =$
$((rho\ \&[z1\ //\ z]\text{-}zs)\ \&[(y1\ @ys[z1\ /\ z]\text{-}zs)\ //\ y]\text{-}ys)$
⟨*proof*⟩

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
$(good\ X \longrightarrow$
$(\forall\ z.\ z \in varsOf\ P \longrightarrow (X\ \#[z\ //\ z]\text{-}zs) = X))$
$\wedge$
$(goodAbs\ A \longrightarrow$
$(\forall\ z.\ z \in varsOf\ P \longrightarrow (A\ \$[z\ //\ z]\text{-}zs) = A))$
⟨*proof*⟩

corollary *vsubst-ident*[*simp*]:
assumes *good X*
shows $(X\ \#[z\ //\ z]\text{-}zs) = X$
⟨*proof*⟩

corollary *subst-ident*[*simp*]:
assumes *good X*
shows $(X\ \#[(Var\ zs\ z)\ /\ z]\text{-}zs) = X$
⟨*proof*⟩

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
$(good\ X \longrightarrow$
$(\forall\ y1\ y2.\ \{y1,y2\} \subseteq varsOf\ P \wedge fresh\ ys\ y1\ X \longrightarrow$
$(X\ \#[y1\ //\ y2]\text{-}ys) = (X\ \#[y1 \wedge y2]\text{-}ys)))$
$\wedge$
$(goodAbs\ A \longrightarrow$
$(\forall\ y1\ y2.\ \{y1,y2\} \subseteq varsOf\ P \wedge freshAbs\ ys\ y1\ A \longrightarrow$
$(A\ \$[y1\ //\ y2]\text{-}ys) = (A\ \$[y1 \wedge y2]\text{-}ys)))$

⟨*proof*⟩

corollary *vsubst-eq-swap*:
assumes *good X* **and** *y1 = y2* ∨ *fresh ys y1 X*
shows (*X* #[*y1* // *y2*]-*ys*) = (*X* #[*y1* ∧ *y2*]-*ys*)
⟨*proof*⟩

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*⟩

corollary *skel-vsubst*:
assumes *good X*
shows *skel* (*X* #[*y1* // *y2*]-*ys*) = *skel X*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubst-trans*:
assumes *good X* **and** *fresh ys y1 X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y1*]-*ys*) = (*X* #[*y2* // *y*]-*ys*)
⟨*proof*⟩

lemma *vsubst-commute*:
assumes *X*: *good X*
and *xs* ≠ *xs'* ∨ {*x*,*y*} ∩ {*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*⟩

6.6 Abstraction versions of the properties

Environment identity and update versus other operators:

lemma *psubstAbs-idEnv*[*simp*]:
goodAbs A ⟹ (*A* $[*idEnv*]) = *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubstAbs-preserves-freshAbs*:
assumes *goodAbs A*
and *freshAbs zs z A* **and** $zs \neq ys \vee z \neq y1$
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
⟨*proof*⟩

lemma *fresh-freshAbs-substAbs*[*simp*]:
assumes *good Y* **and** *goodAbs A*
and *fresh ys y Y*
shows *freshAbs ys y* (*A* $[*Y* / *y*]-*ys*)
⟨*proof*⟩

lemma *diff-freshAbs-vsubstAbs*[*simp*]:
assumes *goodAbs A*
and $y \neq y1$
shows *freshAbs ys y* (*A* $[*y1* // *y*]-*ys*)
⟨*proof*⟩

lemma *freshAbs-substAbs-E1*:
assumes *goodAbs A* **and** *good Y*
and *freshAbs zs z* (*A* $[*Y* / *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *freshAbs zs z A*
⟨*proof*⟩

lemma *freshAbs-vsubstAbs-E1*:
assumes *goodAbs A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *freshAbs zs z A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *freshAbs-vsubstAbs-E2*:
assumes *goodAbs A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
shows *freshAbs ys y A* $\vee zs \neq ys \vee z \neq y1$
⟨*proof*⟩

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'*])

⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *freshAbs-substAbs-ident*[*simp*]:
assumes *good X* **and** *goodAbs A* **and** *freshAbs xs x A*
shows (*A* $[*X* / *x*]-*xs*) = *A*
⟨*proof*⟩

corollary *substAbs-Abs*[*simp*]:
assumes *good X* **and** *good Y*
shows ((*Abs xs x X*) $[*Y* / *x*]-*xs*) = *Abs xs x X*
⟨*proof*⟩

lemma *freshAbs-vsubstAbs-ident*[*simp*]:
assumes *goodAbs A* **and** *freshAbs xs x A*
shows (*A* $[*x1* // *x*]-*xs*) = *A*
⟨*proof*⟩

lemma *swapAbs-psubstAbs*:
assumes *goodAbs A* **and** *goodEnv rho*
shows ((*A* $[*rho*]) $[*z1* ∧ *z2*]-*zs*) = ((*A* $[*z1* ∧ *z2*]-*zs*) $[*rho* &[*z1* ∧ *z2*]-*zs*])
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *psubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho* **and** *goodEnv rho'*

shows $((A\ \$[rho])\ \$[rho']) = (A\ \$[(rho\ \&[rho'])])$
⟨*proof*⟩

lemma *psubstAbs-substAbs-compose*:
assumes *goodAbs A* **and** *good Y* **and** *goodEnv rho*
shows $((A\ \$[Y\ /\ y]\text{-}ys)\ \$[rho]) = (A\ \$[(rho\ [y \leftarrow (Y\ \#[rho])]\text{-}ys)])$
⟨*proof*⟩

lemma *psubstAbs-vsubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho*
shows $((A\ \$[y1\ //\ y]\text{-}ys)\ \$[rho]) = (A\ \$[(rho\ [y \leftarrow ((Var\ ys\ y1)\ \#[rho])]\text{-}ys)])$
⟨*proof*⟩

lemma *substAbs-psubstAbs-compose*:
assumes *goodAbs A* **and** *good Y* **and** *goodEnv rho*
shows $((A\ \$[rho])\ \$[Y\ /\ y]\text{-}ys) = (A\ \$[(rho\ \&[Y\ /\ y]\text{-}ys)])$
⟨*proof*⟩

lemma *psubstAbs-substAbs-compose-freshEnv*:
assumes *goodAbs A* **and** *goodEnv rho* **and** *good Y*
assumes *freshEnv ys y rho*
shows $((A\ \$[Y\ /\ y]\text{-}ys)\ \$[rho]) = ((A\ \$[rho])\ \$[(Y\ \#[rho])\ /\ y]\text{-}ys)$
⟨*proof*⟩

lemma *vsubstAbs-psubstAbs-compose*:
assumes *goodAbs A* **and** *goodEnv rho*
shows $((A\ \$[rho])\ \$[y1\ //\ y]\text{-}ys) = (A\ \$[(rho\ \&[y1\ //\ y]\text{-}ys)])$
⟨*proof*⟩

lemma *substAbs-compose1*:
assumes *goodAbs A* **and** *good Y1* **and** *good Y2*
shows $((A\ \$[Y1\ /\ y]\text{-}ys)\ \$[Y2\ /\ y]\text{-}ys) = (A\ \$[(Y1\ \#[Y2\ /\ y]\text{-}ys)\ /\ y]\text{-}ys)$
⟨*proof*⟩

lemma *substAbs-vsubstAbs-compose1*:
assumes *goodAbs A* **and** *good Y* **and** $y \neq y1$
shows $((A\ \$[y1\ //\ y]\text{-}ys)\ \$[Y\ /\ y]\text{-}ys) = (A\ \$[y1\ //\ y]\text{-}ys)$
⟨*proof*⟩

lemma *vsubstAbs-substAbs-compose1*:
assumes *goodAbs A* **and** *good Y*
shows $((A\ \$[Y\ /\ y]\text{-}ys)\ \$[y1\ //\ y]\text{-}ys) = (A\ \$[(Y\ \#[y1\ //\ y]\text{-}ys)\ /\ y]\text{-}ys)$
⟨*proof*⟩

lemma *vsubstAbs-compose1*:
assumes *goodAbs A*
shows $((A\ \$[y1\ //\ y]\text{-}ys)\ \$[y2\ //\ y]\text{-}ys) = (A\ \$[(y1\ @ys[y2\ /\ y]\text{-}ys)\ //\ y]\text{-}ys)$
⟨*proof*⟩

lemma *substAbs-compose2*:
assumes *goodAbs A* **and** *good Y* **and** *good 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*)
⟨*proof*⟩

lemma *substAbs-vsubstAbs-compose2*:
assumes *goodAbs A* **and** *good 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*)
⟨*proof*⟩

lemma *vsubstAbs-substAbs-compose2*:
assumes *goodAbs A* **and** *good 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*)
⟨*proof*⟩

lemma *vsubstAbs-compose2*:
assumes *goodAbs 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*)
⟨*proof*⟩

Properties specific to variable-for-variable substitution:

corollary *vsubstAbs-ident*[*simp*]:
assumes *goodAbs A*
shows (*A* $[*z* // *z*]-*zs*) = *A*
⟨*proof*⟩

corollary *substAbs-ident*[*simp*]:
assumes *goodAbs A*
shows (*A* $[(*Var zs z*) / *z*]-*zs*) = *A*
⟨*proof*⟩

corollary *vsubstAbs-eq-swapAbs*:
assumes *goodAbs A* **and** *freshAbs ys y1 A*
shows (*A* $[*y1* // *y2*]-*ys*) = (*A* $[*y1* ∧ *y2*]-*ys*)
⟨*proof*⟩

corollary *skelAbs-vsubstAbs*:
assumes *goodAbs A*
shows *skelAbs* (*A* $[*y1* // *y2*]-*ys*) = *skelAbs A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubstAbs-trans*:
assumes *goodAbs A* **and** *freshAbs ys y1 A*
shows ((*A* $[*y1* // *y*]-*ys*) $[*y2* // *y1*]-*ys*) = (*A* $[*y2* // *y*]-*ys*)
⟨*proof*⟩

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 “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**

[1] We shall use “wls" in many contexts as a prefix indicating well-sortedness or well-structuredness.

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*⟩

sublocale *FixSyn* < *FixVars*
where *dummyV* = *dummyV* **and** *dummyVS* = *undefined*::'varSort
⟨*proof*⟩

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
⟨*proof*⟩

lemma *swapInp-empInp*[*simp*]:
(*empInp* %[*x1* ∧ *x2*]-*xs*) = *empInp*
⟨*proof*⟩

lemma *psubstInp-empInp*[*simp*]:
(*empInp* %[*rho*]) = *empInp*
⟨*proof*⟩

lemma *substInp-empInp*[*simp*]:
(*empInp* %[*Y* / *y*]-*ys*) = *empInp*
⟨*proof*⟩

lemma *vsubstInp-empInp*[*simp*]:
(*empInp* %[*y1* // *y*]-*ys*) = *empInp*
⟨*proof*⟩

lemma *freshBinp-empBinp*[*simp*]:
freshBinp xs x empBinp
⟨*proof*⟩

lemma *swapBinp-empBinp*[*simp*]:
(*empBinp* %%[*x1* ∧ *x2*]-*xs*) = *empBinp*
⟨*proof*⟩

lemma *psubstBinp-empBinp*[*simp*]:
(*empBinp* %%[*rho*]) = *empBinp*
⟨*proof*⟩

lemma *substBinp-empBinp*[*simp*]:
(*empBinp* %%[*Y* / *y*]-*ys*) = *empBinp*
⟨*proof*⟩

lemma *vsubstBinp-empBinp*[*simp*]:
(*empBinp* %%[*y1* // *y*]-*ys*) = *empBinp*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *asSort-inj*[*simp*]:
(*asSort xs* = *asSort ys*) = (*xs* = *ys*)
⟨*proof*⟩

lemma *arityOf-lt-var-INNER*:
assumes *wlsOpS delta*
shows $|\{i.\ arityOf\ Delta\ delta\ i \neq None\}|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

lemma *barityOf-lt-var-INNER*:
assumes *wlsOpS delta*
shows $|\{i.\ barityOf\ Delta\ delta\ i \neq None\}|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

lemma *sort-lt-var-INNER*:
$|UNIV :: 'sort\ set|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

lemma *sort-le-var*:
$|UNIV :: 'sort\ set|\ \leq o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

lemma *varSort-sort-lt-var*:
$|UNIV :: ('varSort * 'sort)\ set|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

lemma *varSort-sort-le-var*:
$|UNIV :: ('varSort * 'sort)\ set|\ \leq o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

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* $\Rightarrow$ (*'index,'bindex,'varSort,'var,'opSym*)*term* $\Rightarrow$ *bool*
and
wlsAbs :: *'varSort* $*$ *'sort* $\Rightarrow$ (*'index,'bindex,'varSort,'var,'opSym*)*abs* $\Rightarrow$ *bool*
and
wlsInp :: *'opSym* $\Rightarrow$ (*'index,*(*'index,'bindex,'varSort,'var,'opSym*)*term*)*input* $\Rightarrow$ *bool*
and
wlsBinp :: *'opSym* $\Rightarrow$ (*'bindex,*(*'index,'bindex,'varSort,'var,'opSym*)*abs*)*input* $\Rightarrow$ *bool*
where
Var: *wls* (*asSort xs*) (*Var xs x*)

|
Op: ⟦*wlsInp delta inp*; *wlsBinp delta binp*⟧ $\Longrightarrow$ *wls* (*stOf delta*) (*Op delta inp binp*)
|
Inp:
⟦*wlsOpS delta*;
 $\bigwedge$ *i*. (*arOf delta i* = *None* ∧ *inp i* = *None*) ∨
 (∃ *s X*. *arOf delta i* = *Some s* ∧ *inp i* = *Some X* ∧ *wls s X*)⟧
 $\Longrightarrow$ *wlsInp delta inp*
|
Binp:
⟦*wlsOpS delta*;
 $\bigwedge$ *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*)⟧
 $\Longrightarrow$ *wlsBinp delta binp*
|
Abs: ⟦*isInBar* (*xs*,*s*); *wls s X*⟧ $\Longrightarrow$ *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*)
⟨*proof*⟩

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 $\bigwedge$ *s X*. *X* ∈ *set* (*XLF s*) $\Longrightarrow$ *wls s X*

and $\bigwedge$ *xs s A. A* ∈ *set* (*ALF* (*xs*,*s*)) ⟹ *wlsAbs* (*xs*,*s*) *A*
and $\bigwedge$ *rho. rho* ∈ *set rhoF* ⟹ *wlsEnv rho*
shows *wlsPar* (*ParS xLF XLF ALF rhoF*)
⟨*proof*⟩

lemma *termsOfS-preserves-wls*[*simp*]:
assumes *wlsPar P* **and** *X* : *termsOfS P s*
shows *wls s X*
⟨*proof*⟩

lemma *absOfS-preserves-wls*[*simp*]:
assumes *wlsPar P* **and** *A* : *absOfS P* (*us*,*s*)
shows *wlsAbs* (*us*,*s*) *A*
⟨*proof*⟩

lemma *envsOfS-preserves-wls*[*simp*]:
assumes *wlsPar P* **and** *rho* : *envsOfS P*
shows *wlsEnv rho*
⟨*proof*⟩

lemma *not-isInBar-absOfS-empty*[*simp*]:
assumes ∗: ¬ *isInBar* (*us*,*s*) **and** ∗∗: *wlsPar P*
shows *absOfS P* (*us*,*s*) = {}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wlsBinp-iff*:
wlsBinp delta binp =
(*wlsOpS delta* ∧ *sameDom* (*barOf delta*) *binp* ∧ *liftAll2 wlsAbs* (*barOf delta*) *binp*)
⟨*proof*⟩

lemma *exists-asSort-wls*:
∃ *X*. *wls* (*asSort xs*) *X*
⟨*proof*⟩

lemma *exists-wls-imp-exists-wlsAbs*:

assumes $*$: *isInBar* (*us*,*s*) **and** $**$: $\exists$ *X*. *wls s X*
shows $\exists$ *A*. *wlsAbs* (*us*,*s*) *A*
⟨*proof*⟩

lemma *exists-asSort-wlsAbs*:
assumes *isInBar* (*us*,*asSort xs*)
shows $\exists$ *A*. *wlsAbs* (*us*,*asSort xs*) *A*
⟨*proof*⟩

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*⟩

lemma *wf-exists-wlsAbs*:
assumes *isInBar* (*us*,*s*)
and *wf R* **and** $\bigwedge$*s*. ($\exists$ *xs*. *s* = *asSort xs*) $\vee$ *witOpS s* (*f s*) *R*
shows $\exists$ *A*. *wlsAbs* (*us*,*s*) *A*
⟨*proof*⟩

7.3.7 Well-sorted implies Good

lemma *wlsInp-empAr-empInp*[*simp*]:
assumes *wlsOpS delta* **and** *arOf delta* = *empAr*
shows *wlsInp delta empInp*
⟨*proof*⟩

lemma *wlsBinp-empBar-empBinp*[*simp*]:
assumes *wlsOpS delta* **and** *barOf delta* = *empBar*
shows *wlsBinp delta empBinp*
⟨*proof*⟩

lemmas *empInp-otherSimps* =
wlsInp-empAr-empInp wlsBinp-empBar-empBinp

lemma *wlsAll-implies-goodAll*:
(*wls s X* $\longrightarrow$ *good X*) $\wedge$
(*wlsAbs* (*xs*,s') *A* $\longrightarrow$ *goodAbs A*) $\wedge$
(*wlsInp delta inp* $\longrightarrow$ *goodInp inp*) $\wedge$

(*wlsBinp delta binp* ⟶ *goodBinp binp*)
⟨*proof*⟩

corollary *wls-imp-good*[*simp*]: *wls s X* ⟹ *good X*
⟨*proof*⟩

corollary *wlsAbs-imp-goodAbs*[*simp*]: *wlsAbs* (*xs*,*s*) *A* ⟹ *goodAbs A*
⟨*proof*⟩

corollary *wlsInp-imp-goodInp*[*simp*]: *wlsInp delta inp* ⟹ *goodInp inp*
⟨*proof*⟩

corollary *wlsBinp-imp-goodBinp*[*simp*]: *wlsBinp delta binp* ⟹ *goodBinp binp*
⟨*proof*⟩

lemma *wlsEnv-imp-goodEnv*[*simp*]: *wlsEnv rho* ⟹ *goodEnv rho*
⟨*proof*⟩

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*⟩

lemma *swap-preserves-wls*[*simp*]:
wls s X ⟹ *wls s* (*X* #[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

lemma *swap-preserves-wls2*[*simp*]:
assumes *good X*
shows *wls s* (*X* #[*z1* ∧ *z2*]-*zs*) = *wls s X*
⟨*proof*⟩

lemma *swap-preserves-wls3*:
assumes *good X* **and** *good Y*
and (*X* #[*x1* ∧ *x2*]-*xs*) = (*Y* #[*y1* ∧ *y2*]-*ys*)
shows *wls s X* = *wls s Y*
⟨*proof*⟩

lemma *swapAbs-preserves-wls*[*simp*]:
wlsAbs (*xs*,*x*) *A* ⟹ *wlsAbs* (*xs*,*x*) (*A* $[*z1* ∧ *z2*]-*zs*)

⟨*proof*⟩

lemma *swapInp-preserves-wls*[*simp*]:
wlsInp delta inp $\Longrightarrow$ *wlsInp delta* (*inp* %[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

lemma *swapBinp-preserves-wls*[*simp*]:
wlsBinp delta binp $\Longrightarrow$ *wlsBinp delta* (*binp* %%[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

lemma *swapEnvDom-preserves-wls*:
assumes *wlsEnv rho*
shows *wlsEnv* (*swapEnvDom xs x y rho*)
⟨*proof*⟩

lemma *swapEnvIm-preserves-wls*:
assumes *wlsEnv rho*
shows *wlsEnv* (*swapEnvIm xs x y rho*)
⟨*proof*⟩

lemma *swapEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho*
shows *wlsEnv* (*rho* &[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

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*⟩

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*)

$\wedge$
(*wlsBinp delta binp* $\longrightarrow$ *True*)
⟨*proof*⟩

lemma *conjLeft*: ⟦*phi1* $\wedge$ *phi2*; *phi1* $\Longrightarrow$ *chi*⟧ $\Longrightarrow$ *chi*
⟨*proof*⟩

lemma *conjRight*: ⟦*phi1* $\wedge$ *phi2*; *phi2* $\Longrightarrow$ *chi*⟧ $\Longrightarrow$ *chi*
⟨*proof*⟩

lemma *wls-inversion*[*rule-format*]:
wls s X $\longrightarrow$
(∀ *xs x*. *X* = *Var xs x* $\longrightarrow$ *s* = *asSort xs*) $\wedge$
(∀ *delta inp binp*. *goodInp inp* $\wedge$ *goodBinp binp* $\wedge$ *X* = *Op delta inp binp* $\longrightarrow$
stOf delta = *s* $\wedge$ *wlsInp delta inp* $\wedge$ *wlsBinp delta binp*)
⟨*proof*⟩

lemma *wlsAbs-inversion*[*rule-format*]:
wlsAbs (*xs*,*s*) *A* $\longrightarrow$
isInBar (*xs*,*s*) $\wedge$
(∀ *x X*. *good X* $\wedge$ *A* = *Abs xs x X* $\longrightarrow$ *wls s X*)
⟨*proof*⟩

lemma *wls-Var-simp*[*simp*]:
wls s (*Var xs x*) = (*s* = *asSort xs*)
⟨*proof*⟩

lemma *wls-Op-simp*[*simp*]:
assumes *goodInp inp* **and** *goodBinp binp*
shows
wls s (*Op delta inp binp*) =
(*stOf delta* = *s* $\wedge$ *wlsInp delta inp* $\wedge$ *wlsBinp delta binp*)
⟨*proof*⟩

lemma *wls-Abs-simp*[*simp*]:
assumes *good X*
shows *wlsAbs* (*xs*,*s*) (*Abs xs x X*) = (*isInBar* (*xs*,*s*) $\wedge$ *wls s X*)
⟨*proof*⟩

lemma *wlsAll-inversion2*:
(*wls s X* $\longrightarrow$ *True*)
$\wedge$
(*wlsAbs xs-s A* $\longrightarrow$
isInBar xs-s $\wedge$
(∃ *x X*. *wls* (*snd xs-s*) *X* $\wedge$ *A* = *Abs* (*fst xs-s*) *x X*))
$\wedge$
(*wlsInp delta inp* $\longrightarrow$ *True*)

$\wedge$
(*wlsBinp delta binp* $\longrightarrow$ *True*)
⟨*proof*⟩

lemma *wlsAbs-inversion2*[*rule-format*]:
wlsAbs (*xs*,*s*) *A* $\longrightarrow$
isInBar (*xs*,*s*) $\wedge$ ($\exists$ *x X*. *wls s X* $\wedge$ *A* = *Abs xs x X*)
⟨*proof*⟩

corollary *wlsAbs-Abs-varSort*:
assumes *X*: *good X* **and** *wlsAbs*: *wlsAbs* (*xs*,*s*) (*Abs xs' x X*)
shows *xs* = *xs'*
⟨*proof*⟩

lemma *wlsAbs*:
wlsAbs (*xs*,*s*) *A* $\longleftrightarrow$
isInBar (*xs*,*s*) $\wedge$ ($\exists$ *x X*. *wls s X* $\wedge$ *A* = *Abs xs x X*)
⟨*proof*⟩

lemma *wlsAbs-Abs*[*simp*]:
assumes *X*: *good X*
shows *wlsAbs* (*xs'*,*s*) (*Abs xs x X*) = (*isInBar* (*xs'*,*s*) $\wedge$ *xs* = *xs'* $\wedge$ *wls s X*)
⟨*proof*⟩

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: $\bigwedge$ *s X Y*. ⟦*wls s X*; (*X*,*Y*) $\in$ *rel s*⟧ $\Longrightarrow$ *wls s Y* $\wedge$ *skel Y* = *skel X* **and**
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*) $\in$ *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*⟩

theorem *wls-rawInduct*[*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*; *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*⟩

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* $\Rightarrow$ *'var set*
and *terms* :: *'sort* $\Rightarrow$ (*'index,'bindex,'varSort,'var,'opSym*)*term set*
and *abs* :: (*'varSort* $*$ *'sort*) $\Rightarrow$ (*'index,'bindex,'varSort,'var,'opSym*)*abs set*
and *envs* :: (*'index,'bindex,'varSort,'var,'opSym*)*env set*
assumes
PAR:
$\bigwedge$ *xs us s.*
  (|*vars xs*| <*o* |*UNIV* :: *'var set*| $\vee$ *finite* (*vars xs*)) $\wedge$
  (|*terms s*| <*o* |*UNIV* :: *'var set*| $\vee$ *finite* (*terms s*)) $\wedge$
  (|*abs* (*us,s*)| <*o* |*UNIV* :: *'var set*| $\vee$ *finite* (*abs* (*us,s*))) $\wedge$
  ($\forall$ *X* $\in$ *terms s. wls s X*) $\wedge$
  ($\forall$ *A* $\in$ *abs* (*us,s*). *wlsAbs* (*us,s*) *A*) $\wedge$
  (|*envs*| <*o* |*UNIV* :: *'var set*| $\vee$ *finite* (*envs*)) $\wedge$
  ($\forall$ *rho* $\in$ *envs. wlsEnv rho*) **and**
rel: $\bigwedge$ *s X Y.* ⟦*wls s X*; $(X,Y) \in$ *rel s*⟧ $\Longrightarrow$ *wls s Y* $\wedge$ *skel Y* = *skel X* **and**
Var: $\bigwedge$ *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* ⟶ *phi s X*) ∧
(*wlsAbs* (*xs*,*s'*) *A* ⟶ *phiAbs* (*xs*,*s'*) *A*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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: $\bigwedge$ *xs x*. *phi* (*asSort xs*) (*Var xs x*) **and**
Op:
$\bigwedge$ *delta inp binp*.
  $[\![$*wlsInp delta inp*; *wlsBinp delta binp*;
   liftAll2 (λs 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)
$\langle$*proof*$\rangle$

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'*)
$\langle$*proof*$\rangle$

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*)))
$\langle$*proof*$\rangle$

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*))
$\langle$*proof*$\rangle$

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

theorem *wls-Abs-inj*[*simp*]:
assumes *wls s X* **and** *wls s X'*
shows
(*Abs xs x X* = *Abs xs x X'*) =
(*X* = *X'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-Var-diff-Op*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows *Var xs x* ≠ *Op delta inp binp*
⟨*proof*⟩

theorem *wls-Op-diff-Var*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsBinp delta binp*
shows *Op delta inp binp* ≠ *Var xs x*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wls-disjoint*:
assumes *wls s X* **and** *wls s′ X*
shows *s* = *s′*
⟨*proof*⟩

lemma *wlsAbs-disjoint*:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wlsAbs* (*xs′*,*s′*) *A*
shows *xs* = *xs′* $\wedge$ *s* = *s′*
⟨*proof*⟩

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*⟩

lemma *updEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wls* (*asSort xs*) *X*
shows *wlsEnv* (*rho* [*x* ← *X*]-*xs*)
⟨*proof*⟩

lemma *getEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *rho xs x* = *Some X*
shows *wls* (*asSort xs*) *X*
⟨*proof*⟩

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*⟩

lemma *psubst-preserves-wls*[*simp*]:
⟦*wls s X*; *wlsEnv rho*⟧ ⟹ *wls s* (*X* #[*rho*])
⟨*proof*⟩

lemma *psubstAbs-preserves-wls*[*simp*]:
⟦*wlsAbs* (*xs*,*s*) *A*; *wlsEnv rho*⟧ ⟹ *wlsAbs* (*xs*,*s*) (*A* $[*rho*])
⟨*proof*⟩

lemma *psubstInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp* **and** *wlsEnv rho*
shows *wlsInp delta* (*inp* %[*rho*])
⟨*proof*⟩

lemma *psubstBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp* **and** *wlsEnv rho*
shows *wlsBinp delta* (*binp* %%[*rho*])
⟨*proof*⟩

lemma *psubstEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wlsEnv rho′*
shows *wlsEnv* (*rho* &[*rho′*])
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *substAbs-preserves-wls*[*simp*]:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wls* (*asSort ys*) *Y*
shows *wlsAbs* (*xs*,*s*) (*A* $[*Y* / *y*]-*ys*)
⟨*proof*⟩

lemma *substInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp* **and** *wls* (*asSort ys*) *Y*
shows *wlsInp delta* (*inp* %[*Y* / *y*]-*ys*)
⟨*proof*⟩

lemma *substBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp* **and** *wls* (*asSort ys*) *Y*
shows *wlsBinp delta* (*binp* %%[*Y* / *y*]-*ys*)
⟨*proof*⟩

lemma *substEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
shows *wlsEnv* (*rho* &[*Y* / *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vsubstAbs-preserves-wls*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows *wlsAbs* (*us*,*s*) (*A* $[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstInp-preserves-wls*[*simp*]:
assumes *wlsInp delta inp*
shows *wlsInp delta* (*inp* %[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstBinp-preserves-wls*[*simp*]:
assumes *wlsBinp delta binp*
shows *wlsBinp delta* (*binp* %%[*x1* // *x*]-*xs*)
⟨*proof*⟩

lemma *vsubstEnv-preserves-wls*[*simp*]:
assumes *wlsEnv rho*
shows *wlsEnv* (*rho* &[*x1* // *x*]-*xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wls-skelInp-def2*:
assumes *wlsInp delta inp*
shows *skelInp inp* = *lift skel inp*
⟨*proof*⟩

lemma *wls-skelBinp-def2*:
assumes *wlsBinp delta binp*
shows *skelBinp binp* = *lift skelAbs binp*
⟨*proof*⟩

theorem *wls-skelAbs-simp*[*simp*]:
assumes *wls s X*
shows *skelAbs* (*Abs xs x X*) = *Branch* (λ*i*. *Some* (*skel X*)) *Map.empty*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-psubst-Var-simp2*[*simp*]:
assumes *wlsEnv rho* **and** *rho xs x* = *Some X*
shows ((*Var xs x*) #[*rho*]) = *X*
⟨*proof*⟩

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*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-subst-Var-simp2*[*simp*]:
assumes *wls* (*asSort xs*) *Y*
shows ((*Var xs x*) #[*Y* / *x*]-*xs*) = *Y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-obtain-rep*:
assumes *wls s X* **and** *FRESH*: *fresh xs x′ X*

shows $\exists\ X'.\ skel\ X' = skel\ X \wedge (X,X') \in swapped \wedge wls\ s\ X' \wedge Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$
$\langle proof \rangle$

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*:
$wls\ s\ X \Longrightarrow |\{x.\ \neg\ fresh\ xs\ x\ X\}|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

theorem *wls-single-non-freshAbs-ordLess-var*:
$wlsAbs\ (us,s)\ A \Longrightarrow |\{x.\ \neg\ freshAbs\ xs\ x\ A\}|\ <o\ |UNIV :: 'var\ set|$
$\langle proof \rangle$

theorem *wls-obtain-fresh*:
fixes $V{::}'varSort \Rightarrow 'var\ set$ **and**
$XS{::}'sort \Rightarrow ('index,'bindex,'varSort,'var,'opSym)term\ set$ **and**
$AS{::}'varSort \Rightarrow 'sort \Rightarrow ('index,'bindex,'varSort,'var,'opSym)abs\ set$ **and**
$Rho{::}('index,'bindex,'varSort,'var,'opSym)env\ set$ **and** zs
assumes $VVar$: $\forall\ xs.\ |V\ xs|\ <o\ |UNIV :: 'var\ set| \vee finite\ (V\ xs)$
and $XSVar$: $\forall\ s.\ |XS\ s|\ <o\ |UNIV :: 'var\ set| \vee finite\ (XS\ s)$
and $ASVar$: $\forall\ xs\ s.\ |AS\ xs\ s|\ <o\ |UNIV :: 'var\ set| \vee finite\ (AS\ xs\ s)$
and $XSwls$: $\forall\ s.\ \forall\ X \in XS\ s.\ wls\ s\ X$
and $ASwls$: $\forall\ xs\ s.\ \forall\ A \in AS\ xs\ s.\ wlsAbs\ (xs,s)\ A$
and $RhoVar$: $|Rho|\ <o\ |UNIV :: 'var\ set| \vee finite\ Rho$
and $Rhowls$: $\forall\ rho \in Rho.\ wlsEnv\ rho$
shows
$\exists\ z.\ (\forall\ xs.\ z \notin V\ xs) \wedge$
$(\forall\ s.\ \forall\ X \in XS\ s.\ fresh\ zs\ z\ X) \wedge$
$(\forall\ xs\ s.\ \forall\ A \in AS\ xs\ s.\ freshAbs\ zs\ z\ A) \wedge$
$(\forall\ rho \in Rho.\ freshEnv\ zs\ z\ rho)$
$\langle proof \rangle$

theorem *wls-obtain-fresh-paramS*:
assumes $wlsPar\ P$
shows
$\exists\ z.$
$(\forall\ xs.\ z \notin varsOfS\ P\ xs) \wedge$
$(\forall\ s.\ \forall\ X \in termsOfS\ P\ s.\ fresh\ zs\ z\ X) \wedge$
$(\forall\ us\ s.\ \forall\ A \in absOfS\ P\ (us,s).\ freshAbs\ zs\ z\ A) \wedge$

($\forall$ *rho* $\in$ *envsOfS P*. *freshEnv zs z rho*)
⟨*proof*⟩

lemma *wlsAbs-freshAbs-nchotomy*:
assumes *A*: *wlsAbs* (*xs*,*s*) *A* **and** *fresh*: *freshAbs xs x A*
shows $\exists$ *X*. *wls s X* $\wedge$ *A* = *Abs xs x X*
⟨*proof*⟩

theorem *wlsAbs-fresh-nchotomy*:
assumes *A*: *wlsAbs* (*xs*,*s*) *A* **and** *P*: *wlsPar P*
shows $\exists$ *x X*. *A* = *Abs xs x X* $\wedge$
wls s X $\wedge$
($\forall$ *ys*. *x* $\notin$ *varsOfS P ys*) $\wedge$
($\forall$ *s'*. $\forall$ *Y* $\in$ *termsOfS P s'*. *fresh xs x Y*) $\wedge$
($\forall$ *us s'*. $\forall$ *B* $\in$ *absOfS P* (*us*,*s'*). *freshAbs xs x B*) $\wedge$
($\forall$ *rho* $\in$ *envsOfS P*. *freshEnv xs x rho*)
⟨*proof*⟩

theorem *wlsAbs-fresh-cases*:
assumes *wlsAbs* (*xs*,*s*) *A* **and** *wlsPar P*
and $\bigwedge$ *x X*.
⟦*wls s X*;
$\bigwedge$ *ys*. *x* $\notin$ *varsOfS P ys*;
$\bigwedge$ *s' Y*. *Y* $\in$ *termsOfS P s'* $\Longrightarrow$ *fresh xs x Y*;
$\bigwedge$ *us s' B*. *B* $\in$ *absOfS P* (*us*,*s'*) $\Longrightarrow$ *freshAbs xs x B*;
$\bigwedge$ *rho*. *rho* $\in$ *envsOfS P* $\Longrightarrow$ *freshEnv xs x rho*⟧
$\Longrightarrow$ *phi* (*xs*,*s*) (*Abs xs x X*) *P*
shows *phi* (*xs*,*s*) *A P*
⟨*proof*⟩

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* $\wedge$ *x*]-*xs*) = *X*
⟨*proof*⟩

theorem *wls-swap-compose*:
assumes *wls s X*
shows ((*X* #[*x* $\wedge$ *y*]-*zs*) #[*x'* $\wedge$ *y'*]-*zs'*) =
((*X* #[*x'* $\wedge$ *y'*]-*zs'*) #[(*x* @*zs*[*x'* $\wedge$ *y'*]-*zs'*) $\wedge$ (*y* @*zs*[*x'* $\wedge$ *y'*]-*zs'*)]-*zs*)
⟨*proof*⟩

theorem *wls-swap-commute*:
⟦*wls s X*; *zs* $\neq$ *zs'* $\vee$ {*x*,*y*} $\cap$ {*x'*,*y'*} = {}⟧ $\Longrightarrow$
((*X* #[*x* $\wedge$ *y*]-*zs*) #[*x'* $\wedge$ *y'*]-*zs'*) = ((*X* #[*x'* $\wedge$ *y'*]-*zs'*) #[*x* $\wedge$ *y*]-*zs*)
⟨*proof*⟩

theorem *wls-swap-involutive*[*simp*]:
assumes *wls s X*
shows $((X \#[x \wedge y]\text{-}zs) \#[x \wedge y]\text{-}zs) = X$
⟨*proof*⟩

theorem *wls-swap-inj*[*simp*]:
assumes *wls s X* **and** *wls s X′*
shows
$((X \#[x \wedge y]\text{-}zs) = (X' \#[x \wedge y]\text{-}zs)) =$
$(X = X')$
⟨*proof*⟩

theorem *wls-swap-involutive2*[*simp*]:
assumes *wls s X*
shows $((X \#[x \wedge y]\text{-}zs) \#[y \wedge x]\text{-}zs) = X$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-fresh-swap-exchange1*:
assumes *wls s X*
shows *fresh xs x2* $(X \#[x1 \wedge x2]\text{-}xs)$ = *fresh xs x1 X*
⟨*proof*⟩

theorem *wls-fresh-swap-exchange2*:
assumes *wls s X*
shows *fresh xs x2* $(X \#[x2 \wedge x1]\text{-}xs)$ = *fresh xs x1 X*
⟨*proof*⟩

theorem *wls-fresh-swap-id*[*simp*]:
assumes *wls s X* **and** *fresh xs x1 X* **and** *fresh xs x2 X*
shows $(X \#[x1 \wedge x2]\text{-}xs) = X$
⟨*proof*⟩

theorem *wls-fresh-swap-compose*:
assumes *wls s X* **and** *fresh xs y X* **and** *fresh xs z X*
shows $((X \#[y \wedge x]\text{-}xs) \#[z \wedge y]\text{-}xs) = (X \#[z \wedge x]\text{-}xs)$
⟨*proof*⟩

theorem *wls-skel-swap*:
assumes *wls s X*
shows *skel* (*X* #[*x1* ∧ *x2*]-*xs*) = *skel X*
⟨*proof*⟩

7.8.2 W.r.t. environments

theorem *wls-swapEnv-ident*[*simp*]:
assumes *wlsEnv rho*
shows (*rho* &[*x* ∧ *x*]-*xs*) = *rho*
⟨*proof*⟩

theorem *wls-swapEnv-compose*:
assumes *wlsEnv 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*⟩

theorem *wls-swapEnv-commute*:
⟦*wlsEnv rho*; *zs* ≠ *zs'* ∨ {*x*,*y*} ∩ {*x'*,*y'*} = {}⟧ ⟹
((*rho* &[*x* ∧ *y*]-*zs*) &[*x'* ∧ *y'*]-*zs'*) = ((*rho* &[*x'* ∧ *y'*]-*zs'*) &[*x* ∧ *y*]-*zs*)
⟨*proof*⟩

theorem *wls-swapEnv-involutive*[*simp*]:
assumes *wlsEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*x* ∧ *y*]-*zs*) = *rho*
⟨*proof*⟩

theorem *wls-swapEnv-inj*[*simp*]:
assumes *wlsEnv rho* **and** *wlsEnv rho'*
shows
((*rho* &[*x* ∧ *y*]-*zs*) = (*rho'* &[*x* ∧ *y*]-*zs*)) =
(*rho* = *rho'*)
⟨*proof*⟩

theorem *wls-swapEnv-involutive2*[*simp*]:
assumes *wlsEnv rho*
shows ((*rho* &[*x* ∧ *y*]-*zs*) &[*y* ∧ *x*]-*zs*) = *rho*
⟨*proof*⟩

theorem *wls-swapEnv-preserves-freshEnv*[*simp*]:
assumes *wlsEnv rho*
shows *freshEnv xs* (*x* @*xs*[*y1* ∧ *y2*]-*ys*) (*rho* &[*y1* ∧ *y2*]-*ys*) = *freshEnv xs x rho*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-freshEnv-swapEnv-exchange1*:
assumes *wlsEnv rho*
shows *freshEnv xs x2* (*rho* &[*x1* ∧ *x2*]-*xs*) = *freshEnv xs x1 rho*
⟨*proof*⟩

theorem *wls-freshEnv-swapEnv-exchange2*:
assumes *wlsEnv rho*
shows *freshEnv xs x2* (*rho* &[*x2* ∧ *x1*]-*xs*) = *freshEnv xs x1 rho*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

7.8.3 W.r.t. abstractions

theorem *wls-swapAbs-ident*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ (*A* $[*x* ∧ *x*]-*xs*) = *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-swapAbs-involutive*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ ((*A* $[*x* ∧ *y*]-*zs*) $[*x* ∧ *y*]-*zs*) = *A*
⟨*proof*⟩

theorem *wls-swapAbs-sym*:

$wlsAbs\ (us,s)\ A \Longrightarrow (A\ \$[x \wedge y]\text{-}zs) = (A\ \$[y \wedge x]\text{-}zs)$
⟨*proof*⟩

theorem *wls-swapAbs-inj*[*simp*]:
assumes $wlsAbs\ (us,s)\ A$ **and** $wlsAbs\ (us,s)\ A'$
shows
$((A\ \$[x \wedge y]\text{-}zs) = (A'\ \$[x \wedge y]\text{-}zs)) =$
$(A = A')$
⟨*proof*⟩

theorem *wls-swapAbs-involutive2*[*simp*]:
$wlsAbs\ (us,s)\ A \Longrightarrow ((A\ \$[x \wedge y]\text{-}zs)\ \$[y \wedge x]\text{-}zs) = A$
⟨*proof*⟩

theorem *wls-swapAbs-preserves-freshAbs*[*simp*]:
$wlsAbs\ (us,s)\ A$
$\Longrightarrow freshAbs\ xs\ (x\ @xs[y1 \wedge y2]\text{-}ys)\ (A\ \$[y1 \wedge y2]\text{-}ys) = freshAbs\ xs\ x\ A$
⟨*proof*⟩

theorem *wls-swapAbs-preserves-freshAbs-distinct*:
$[\![wlsAbs\ (us,s)\ A;\ xs \neq ys \vee x \notin \{y1,y2\}]\!]$
$\Longrightarrow freshAbs\ xs\ x\ (A\ \$[y1 \wedge y2]\text{-}ys) = freshAbs\ xs\ x\ A$
⟨*proof*⟩

theorem *wls-freshAbs-swapAbs-exchange1*:
$wlsAbs\ (us,s)\ A$
$\Longrightarrow freshAbs\ xs\ x2\ (A\ \$[x1 \wedge x2]\text{-}xs) = freshAbs\ xs\ x1\ A$
⟨*proof*⟩

theorem *wls-freshAbs-swapAbs-exchange2*:
$wlsAbs\ (us,s)\ A$
$\Longrightarrow freshAbs\ xs\ x2\ (A\ \$[x2 \wedge x1]\text{-}xs) = freshAbs\ xs\ x1\ A$
⟨*proof*⟩

theorem *wls-freshAbs-swapAbs-id*[*simp*]:
assumes $wlsAbs\ (us,s)\ A$
and $freshAbs\ xs\ x1\ A$ **and** $freshAbs\ xs\ x2\ A$
shows $(A\ \$[x1 \wedge x2]\text{-}xs) = A$
⟨*proof*⟩

lemma *wls-freshAbs-swapAbs-compose-aux*:
$[\![wlsAbs\ (us,s)\ A;\ wlsPar\ P]\!] \Longrightarrow$
$\forall\ x\ y\ z.\ \{x,y,z\} \subseteq varsOfS\ P\ xs \wedge freshAbs\ xs\ y\ A \wedge freshAbs\ xs\ z\ A \longrightarrow$
$((A\ \$[y \wedge x]\text{-}xs)\ \$[z \wedge y]\text{-}xs) = (A\ \$[z \wedge x]\text{-}xs)$
⟨*proof*⟩

theorem *wls-freshAbs-swapAbs-compose*:
assumes $wlsAbs\ (us,s)\ A$
and $freshAbs\ xs\ y\ A$ **and** $freshAbs\ xs\ z\ A$

shows $((A\ \$[y \wedge x]\text{-}xs)\ \$[z \wedge y]\text{-}xs) = (A\ \$[z \wedge x]\text{-}xs)$
⟨*proof*⟩

theorem *wls-skelAbs-swapAbs*:
$wlsAbs\ (us,s)\ A$
$\Longrightarrow skelAbs\ (A\ \$[x1 \wedge x2]\text{-}xs) = skelAbs\ A$
⟨*proof*⟩

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*]:
$wls\ s\ X \Longrightarrow (X\ \#[idEnv]) = X$
⟨*proof*⟩

theorem *wls-psubstEnv-idEnv-id*[*simp*]:
$wlsEnv\ rho \Longrightarrow (rho\ \&[idEnv]) = rho$
⟨*proof*⟩

theorem *wls-swapEnv-updEnv-fresh*:
assumes $zs \neq ys \vee y \notin \{z1,z2\}$ **and** $wls\ (asSort\ ys)\ Y$
and $fresh\ zs\ z1\ Y$ **and** $fresh\ zs\ z2\ Y$
shows $((rho\ [y \leftarrow Y]\text{-}ys)\ \&[z1 \wedge z2]\text{-}zs) = ((rho\ \&[z1 \wedge z2]\text{-}zs)\ [y \leftarrow Y]\text{-}ys)$
⟨*proof*⟩

7.9.2 Substitution versus other operators

theorem *wls-fresh-psubst*:
assumes $wls\ s\ X$ **and** $wlsEnv\ rho$
shows
$fresh\ zs\ z\ (X\ \#[rho]) =$
$(\forall\ ys\ y.\ fresh\ ys\ y\ X \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)$
⟨*proof*⟩

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* $\vee$ (*ys* $\neq$ *zs* $\vee$ *y* $\neq$ *z*)
⟨*proof*⟩

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* $\vee$ *fresh zs z Y*
⟨*proof*⟩

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*])
⟨*proof*⟩

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'*])
⟨*proof*⟩

theorem *wls-fresh-psubst-I*:
assumes *wls s X* **and** *wlsEnv 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*])
⟨*proof*⟩

theorem *wls-fresh-subst*:
assumes *wls s X* **and** *wls* (*asSort ys*) *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*))
⟨*proof*⟩

theorem *wls-fresh-vsubst*:
assumes *wls s 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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-substEnv-preserves-freshEnv*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
and *freshEnv zs z rho* **and** *fresh zs z Y* **and** $zs \neq ys \vee z \neq y$
shows *freshEnv zs z* (*rho* &[*Y* / *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubst-preserves-fresh*:
assumes *wls s X*
and *fresh zs z X* **and** $zs \neq ys \vee z \neq y1$
shows *fresh zs z* (*X* #[*y1* // *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubstEnv-preserves-freshEnv*:
assumes *wlsEnv rho*
and *freshEnv zs z rho* **and** $zs \neq ys \vee z \notin \{y,y1\}$
shows *freshEnv zs z* (*rho* &[*y1* // *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-diff-fresh-vsubst*[*simp*]:
assumes *wls s X*
and $y \neq y1$
shows *fresh ys y* (*X* #[*y1* // *y*]-*ys*)
⟨*proof*⟩

theorem *wls-fresh-subst-E1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and *fresh zs z* (*X* #[*Y* / *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *fresh zs z X*
⟨*proof*⟩

theorem *wls-fresh-vsubst-E1*:
assumes *wls s X*
and *fresh zs z* (*X* #[*y1* // *y*]-*ys*) **and** $zs \neq ys \vee z \neq y$
shows *fresh zs z X*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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′*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-fresh-vsubst-ident*[*simp*]:
assumes *wls s X* **and** *fresh ys y X*
shows (*X* #[*y1* // *y*]-*ys*) = *X*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-swapEnv-psubstEnv*:
assumes *wlsEnv rho* **and** *wlsEnv rho′*
shows ((*rho* &[*rho′*]) &[*z1* ∧ *z2*]-*zs*) = ((*rho* &[*z1* ∧ *z2*]-*zs*) &[*rho′* &[*z1* ∧ *z2*]-*zs*])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-psubst-compose*:
assumes *wls s X* **and** *wlsEnv rho* **and** *wlsEnv rho′*
shows ((*X* #[*rho*]) #[*rho′*]) = (*X* #[(*rho* &[*rho′*])])
⟨*proof*⟩

theorem *wls-psubstEnv-compose*:
assumes *wlsEnv rho* **and** *wlsEnv rho′* **and** *wlsEnv rho′′*
shows ((*rho* &[*rho′*]) &[*rho′′*]) = (*rho* &[(*rho′* &[*rho′′*])])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

theorem *wls-vsubst-psubst-compose*:
assumes *wls s X* **and** *wlsEnv rho*
shows ((*X* #[*rho*]) #[*y1* // *y*]-*ys*) = (*X* #[(*rho* &[*y1* // *y*]-*ys*)])
⟨*proof*⟩

theorem *wls-vsubstEnv-psubstEnv-compose*:

assumes *wlsEnv rho* **and** *wlsEnv rho′*
shows ((*rho* &[*rho′*]) &[*y1* // *y*]-*ys*) = (*rho* &[(*rho′* &[*y1* // *y*]-*ys*)])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-subst-vsubst-compose1*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y* **and** *y* ≠ *y1*
shows ((*X* #[*y1* // *y*]-*ys*) #[*Y* / *y*]-*ys*) = (*X* #[*y1* // *y*]-*ys*)
⟨*proof*⟩

theorem *wls-substEnv-vsubstEnv-compose1*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** *y* ≠ *y1*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*Y* / *y*]-*ys*) = (*rho* &[*y1* // *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-vsubst-compose1*:
assumes *wls s X*
shows ((*X* #[*y1* // *y*]-*ys*) #[*y2* // *y*]-*ys*) = (*X* #[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubstEnv-compose1*:
assumes *wlsEnv rho*
shows ((*rho* &[*y1* // *y*]-*ys*) &[*y2* // *y*]-*ys*) = (*rho* &[(*y1* @*ys*[*y2* / *y*]-*ys*) // *y*]-*ys*)
⟨*proof*⟩

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*)

⟨*proof*⟩

theorem *wls-substEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *Z*
and $ys \neq zs \vee y \neq z$ **and** *fresh*: *fresh ys y Z*
shows ((*rho* &[*Y* / *y*]-*ys*) &[*Z* / *z*]-*zs*) = ((*rho* &[*Z* / *z*]-*zs*) &[(*Y* #[*Z* / *z*]-*zs*) / *y*]-*ys*)
⟨*proof*⟩

theorem *wls-subst-vsubst-compose2*:
assumes *wls s X* **and** *wls* (*asSort zs*) *Z*
and $ys \neq zs \vee y \neq z$ **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*)
⟨*proof*⟩

theorem *wls-substEnv-vsubstEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort zs*) *Z*
and $ys \neq zs \vee y \neq z$ **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*)
⟨*proof*⟩

theorem *wls-vsubst-subst-compose2*:
assumes *wls s X* **and** *wls* (*asSort ys*) *Y*
and $ys \neq zs \vee y \notin \{z,z1\}$
shows ((*X* #[*Y* / *y*]-*ys*) #[*z1* // *z*]-*zs*) = ((*X* #[*z1* // *z*]-*zs*) #[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubstEnv-substEnv-compose2*:
assumes *wlsEnv rho* **and** *wls* (*asSort ys*) *Y*
and $ys \neq zs \vee y \notin \{z,z1\}$
shows ((*rho* &[*Y* / *y*]-*ys*) &[*z1* // *z*]-*zs*) = ((*rho* &[*z1* // *z*]-*zs*) &[(*Y* #[*z1* // *z*]-*zs*) / *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubst-compose2*:
assumes *wls s X*
and $ys \neq zs \vee y \notin \{z,z1\}$
shows ((*X* #[*y1* // *y*]-*ys*) #[*z1* // *z*]-*zs*) = ((*X* #[*z1* // *z*]-*zs*) #[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)
⟨*proof*⟩

theorem *wls-vsubstEnv-compose2*:
assumes *wlsEnv rho*
and $ys \neq zs \vee y \notin \{z,z1\}$
shows ((*rho* &[*y1* // *y*]-*ys*) &[*z1* // *z*]-*zs*) =
((*rho* &[*z1* // *z*]-*zs*) &[(*y1* @*ys*[*z1* / *z*]-*zs*) // *y*]-*ys*)

⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-subst-ident*[*simp*]:
assumes *wls s X*
shows (*X* #[(*Var zs z*) / *z*]-*zs*) = *X*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-skel-vsubst*:
assumes *wls s X*
shows *skel* (*X* #[*y1* // *y2*]-*ys*) = *skel X*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-vsubst-commute*:
assumes *wls s X*
and *xs* ≠ *xs'* ∨ {*x*,*y*} ∩ {*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*⟩

theorem *wls-induct*[*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*⟧

$\Longrightarrow$ *phi* (*stOf delta*) (*Op delta inp binp*) **and**
Abs:
$\bigwedge$ *s xs x X*.
⟦*isInBar* (*xs*,*s*); *wls s X*;
$\bigwedge$ *Y*. (*X*,*Y*) ∈ *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* ⟶ *phi s X*) ∧
(*wlsAbs* (*xs*,*s'*) *A* ⟶ *phiAbs* (*xs*,*s'*) *A*)
⟨*proof*⟩

theorem *wls-Abs-vsubst-all-aux*:
assumes *wls s X* **and** *wls s X'*
shows
(*Abs xs x X* = *Abs xs x' X'*) =
(∀ *y*. (*y* = *x* ∨ *fresh xs y X*) ∧ (*y* = *x'* ∨ *fresh xs y X'*) ⟶
(*X* #[*y* // *x*]-*xs*) = (*X'* #[*y* // *x'*]-*xs*))
⟨*proof*⟩

theorem *wls-Abs-vsubst-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*))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

7.9.4 Abstraction versions of the properties

theorem *wls-psubstAbs-idEnv*[*simp*]:
wlsAbs (*us*,*s*) *A* ⟹ (*A* $[*idEnv*]) = *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*

⟨*proof*⟩

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*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-vsubstAbs-preserves-freshAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs zs z A* **and** *zs* ≠ *ys* ∨ *z* ≠ *y1*
shows *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-diff-freshAbs-vsubstAbs*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
and *y* ≠ *y1*

shows *freshAbs ys y* (*A* $[*y1* // *y*]-*ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-freshAbs-vsubstAbs-E2*:
assumes *wlsAbs* (*us*,*s*) *A*
and *freshAbs zs z* (*A* $[*y1* // *y*]-*ys*)
shows *freshAbs ys y A* ∨ $zs \neq ys \vee z \neq y1$
⟨*proof*⟩

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'*])
⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wls-freshAbs-vsubstAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A* **and** *freshAbs xs x A*
shows (*A* $[*x1* // *x*]-*xs*) = *A*
⟨*proof*⟩

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*])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-psubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho* **and** *wlsEnv rho'*
shows ((*A* $[*rho*]) $[*rho'*]) = (*A* $[(*rho* &[*rho'*])])
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)])
⟨*proof*⟩

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*)])
⟨*proof*⟩

theorem *wls-vsubstAbs-psubstAbs-compose*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wlsEnv rho*
shows ((*A* $[*rho*]) $[*y1* // *y*]-*ys*) = (*A* $[(*rho* &[*y1* // *y*]-*ys*)])
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-substAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *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*)
⟨*proof*⟩

theorem *wls-substAbs-vsubstAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort zs*) *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*)
⟨*proof*⟩

theorem *wls-vsubstAbs-substAbs-compose2*:
assumes *wlsAbs* (*us*,*s*) *A* **and** *wls* (*asSort ys*) *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-vsubstAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows (*A* $[*z* // *z*]-*zs*) = *A*
⟨*proof*⟩

theorem *wls-substAbs-ident*[*simp*]:
assumes *wlsAbs* (*us*,*s*) *A*
shows (*A* $[(*Var zs z*) / *z*]-*zs*) = *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-skelAbs-vsubstAbs*:
assumes *wlsAbs* (*us*,*s*) *A*
shows *skelAbs* (*A* $[*y1* // *y2*]-*ys*) = *skelAbs A*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *wls-vsubstAbs-commute*:
assumes *wlsAbs* (*us*,*s*) *A*
and *xs* ≠ *xs′* ∨ {*x*,*y*} ∩ {*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*⟩

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*)
⟨*proof*⟩

lemma *not-isVar-Op*[*simp*]:
¬ *isVar s* (*Op delta inp binp*)
⟨*proof*⟩

lemma *isVar-imp-wls*:
isVar s X ⟹ *wls s X*
⟨*proof*⟩

lemmas *isVar-simps* =
isVar-asSort-Var not-isVar-Op

lemma *castVar-asSort-Var*[*simp*]:
castVar (*asSort xs*) (*Var xs x*) = (*xs*,*x*)
⟨*proof*⟩

lemma *isVar-castVar*:
assumes *isVar s X*
shows *asSort* (*fst* (*castVar s X*)) = *s* ∧
Var (*fst* (*castVar s X*)) (*snd* (*castVar s X*)) = *X*
⟨*proof*⟩

lemma *asSort-castVar*[*simp*]:
isVar s X ⟹ *asSort* (*fst* (*castVar s X*)) = *s*
⟨*proof*⟩

lemma *Var-castVar*[*simp*]:
isVar s X ⟹ *Var* (*fst* (*castVar s X*)) (*snd* (*castVar s X*)) = *X*
⟨*proof*⟩

lemma *castVar-inj*[*simp*]:
assumes ∗: *isVar s X* **and** ∗∗: *isVar s′ X′*
shows (*castVar s X* = *castVar s′ X′*) = (*s* = *s′* ∧ *X* = *X′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *not-isOp-Var*[*simp*]:
¬ *isOp s* (*Var xs X*)
⟨*proof*⟩

lemma *isOp-imp-wls*:
isOp s X ⟹ *wls s X*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *wlsInp-castOp*[*simp*]:
isOp s X ⟹ *wlsInp* (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*))
⟨*proof*⟩

lemma *wlsBinp-castOp*[*simp*]:
isOp s X ⟹ *wlsBinp* (*fst3* (*castOp s X*)) (*trd3* (*castOp s X*))
⟨*proof*⟩

lemma *stOf-castOp*[*simp*]:
isOp s X ⟹ *stOf* (*fst3* (*castOp s X*)) = *s*
⟨*proof*⟩

lemma *Op-castOp*[*simp*]:
isOp s X ⟹
Op (*fst3* (*castOp s X*)) (*snd3* (*castOp s X*)) (*trd3* (*castOp s X*)) = *X*

⟨*proof*⟩

lemma *castOp-inj*[*simp*]:
assumes *isOp s X* **and** *isOp s′ X′*
shows $(castOp\ s\ X = castOp\ s'\ X') = (s = s' \wedge X = X')$
⟨*proof*⟩

lemmas *castOp-simps* =
castOp-stOf-Op wlsInp-castOp wlsBinp-castOp
stOf-castOp Op-castOp castOp-inj

lemma *not-isVar-isOp*:
$\neg\ (isVar\ s\ X \wedge isOp\ s\ X)$
⟨*proof*⟩

lemma *isVar-or-isOp*:
$wls\ s\ X \Longrightarrow isVar\ s\ X \vee isOp\ s\ X$
⟨*proof*⟩

lemma *sortTermCase-asSort-Var-simp*[*simp*]:
$sortTermCase\ fVar\ fOp\ (asSort\ xs)\ (Var\ xs\ x) = fVar\ xs\ x$
⟨*proof*⟩

lemma *sortTermCase-stOf-Op-simp*[*simp*]:
$[\![wlsInp\ delta\ inp;\ wlsBinp\ delta\ binp]\!] \Longrightarrow$
$sortTermCase\ fVar\ fOp\ (stOf\ delta)\ (Op\ delta\ inp\ binp) = fOp\ delta\ inp\ binp$
⟨*proof*⟩

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 $wls\ s\ X \Longrightarrow$
$sortTermCase\ fVar\ fOp\ s\ X = sortTermCase\ gVar\ gOp\ s\ X$
⟨*proof*⟩

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*⟩

lemma *wls-castAbs*[*simp*]:
wlsAbs (*xs*,*s*) *A* ⟹ *wls s* (*snd* (*castAbs xs s A*))
⟨*proof*⟩

lemma *Abs-castAbs*[*simp*]:
wlsAbs (*xs*,*s*) *A* ⟹ *Abs xs* (*fst* (*castAbs xs s A*)) (*snd* (*castAbs xs s A*)) = *A*
⟨*proof*⟩

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* ∧ *x*]-*xs*) = (*X′* #[*y* ∧ *x′*]-*xs*)
⟨*proof*⟩

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*]-*xs*) = (*X′* #[*Y* / *x′*]-*xs*)
⟨*proof*⟩

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*]-*xs*) = (*X′* #[*y* // *x′*]-*xs*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *compatAbsVsubst-imp-compatAbsSubst*[*simp*]:
compatAbsVsubst xs s f ⟹ *compatAbsSubst xs s f*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⋀ *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*
⟨*proof*⟩

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 $('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs)model =$
$igWls :: 'sort \Rightarrow 'gTerm \Rightarrow bool$
$igWlsAbs :: 'varSort \times 'sort \Rightarrow 'gAbs \Rightarrow bool$

$igVar :: 'varSort \Rightarrow 'var \Rightarrow 'gTerm$
$igAbs :: 'varSort \Rightarrow 'var \Rightarrow 'gTerm \Rightarrow 'gAbs$
$igOp :: 'opSym \Rightarrow ('index,'gTerm)input \Rightarrow ('bindex,'gAbs)input \Rightarrow 'gTerm$

$igFresh :: 'varSort \Rightarrow 'var \Rightarrow 'gTerm \Rightarrow bool$
$igFreshAbs :: 'varSort \Rightarrow 'var \Rightarrow 'gAbs \Rightarrow bool$

$igSwap :: 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'gTerm \Rightarrow 'gTerm$
$igSwapAbs :: 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'gAbs \Rightarrow 'gAbs$

$igSubst :: 'varSort \Rightarrow 'gTerm \Rightarrow 'var \Rightarrow 'gTerm \Rightarrow 'gTerm$

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:

[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.

— “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, 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 ==

[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.

$\forall$ *xs s xs' s' A.*
isInBar (*xs*,*s*) $\wedge$ *isInBar* (*xs'*,*s'*) $\wedge$
igWlsAbs MOD (*xs*,*s*) *A* $\wedge$ *igWlsAbs MOD* (*xs'*,*s'*) *A*
$\longrightarrow$ *xs* = *xs'* $\wedge$ *s* = *s'*

definition *igWlsAllDisj* **where**
igWlsAllDisj MOD ==
igWlsDisj MOD $\wedge$ *igWlsAbsDisj MOD*

lemmas *igWlsAllDisj-defs* =
igWlsAllDisj-def
igWlsDisj-def igWlsAbsDisj-def

Abstration domains inhabited only within bound arities:

definition *igWlsAbsIsInBar* **where**
igWlsAbsIsInBar MOD ==
$\forall$ *us s A. igWlsAbs MOD* (*us*,*s*) *A* $\longrightarrow$ *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*
⟨*proof*⟩

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 ==

∀ *delta inp binp*.
igWls MOD (*stOf delta*) (*igOp MOD delta inp binp*) =
(*igWlsInp MOD delta inp* ∧ *igWlsBinp MOD delta binp*)

lemma *igOpIPresIGWlsSTR-imp-igOpIPresIGWls*:
igOpIPresIGWlsSTR MOD ⟹ *igOpIPresIGWls MOD*
⟨*proof*⟩

definition *igConsIPresIGWls* **where**
igConsIPresIGWls MOD ==
igVarIPresIGWls MOD ∧
igAbsIPresIGWls MOD ∧
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*
⟨*proof*⟩

“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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

"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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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 ==
$\forall$ *ys y X. igFreshAbs MOD ys y* (*igAbs MOD ys y X*)

lemma *igFreshIGAbs1STR-imp-igFreshIGAbs1*:
igFreshIGAbs1STR MOD $\Longrightarrow$ *igFreshIGAbs1 MOD*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ∧
igFreshIGAbs1STR MOD ∧ *igFreshIGAbs2STR MOD* ∧
igFreshIGOpSTR MOD

lemmas *igFreshClsSTR-defs* = *igFreshClsSTR-def*
igFreshIGVar-def
igFreshIGAbs1STR-def igFreshIGAbs2STR-def
igFreshIGOpSTR-def

lemma *igFreshClsSTR-imp-igFreshCls*:
igFreshClsSTR MOD ⟹ *igFreshCls MOD*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ==
$\forall$ *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 $\Longrightarrow$ *igSwapIGOp MOD*
⟨*proof*⟩

definition *igSwapCls* **where**
igSwapCls MOD ==
igSwapIGVar MOD $\wedge$
igSwapIGAbs MOD $\wedge$
igSwapIGOp MOD

lemmas *igSwapCls-defs* = *igSwapCls-def*
igSwapIGVar-def
igSwapIGAbs-def
igSwapIGOp-def

definition *igSwapClsSTR* **where**
igSwapClsSTR MOD ==
igSwapIGVar MOD $\wedge$
igSwapIGAbsSTR MOD $\wedge$
igSwapIGOpSTR MOD

lemmas *igSwapClsSTR-defs* = *igSwapClsSTR-def*
igSwapIGVar-def
igSwapIGAbsSTR-def
igSwapIGOpSTR-def

lemma *igSwapClsSTR-imp-igSwapCls*:
igSwapClsSTR MOD $\Longrightarrow$ *igSwapCls MOD*
⟨*proof*⟩

definition *igSubstIGVar1* **where**
igSubstIGVar1 MOD ==
$\forall$ *ys y Y xs x.*
igWls MOD (*asSort ys*) *Y* $\longrightarrow$
(*ys* $\neq$ *xs* $\vee$ *y* $\neq$ *x*) $\longrightarrow$
igSubst MOD ys Y y (*igVar MOD xs x*) = *igVar MOD xs x*

definition *igSubstIGVar1STR* **where**
igSubstIGVar1STR MOD ==
($\forall$ *ys y y1 xs x.*

$(ys \neq xs \vee x \neq y) \longrightarrow$
$igSubst\ MOD\ ys\ (igVar\ MOD\ ys\ y1)\ y\ (igVar\ MOD\ xs\ x) = igVar\ MOD\ xs\ x)$
$\wedge$
$(\forall\ ys\ y\ Y\ xs\ x.$
$igWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$
$(ys \neq xs \vee y \neq x) \longrightarrow$
$igSubst\ MOD\ ys\ Y\ y\ (igVar\ MOD\ xs\ x) = igVar\ MOD\ xs\ x)$

lemma *igSubstIGVar1STR-imp-igSubstIGVar1*:
$igSubstIGVar1STR\ MOD \Longrightarrow igSubstIGVar1\ MOD$
$\langle proof \rangle$

definition *igSubstIGVar2* **where**
$igSubstIGVar2\ MOD ==$
$\forall\ ys\ y\ Y.$
$igWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$
$igSubst\ MOD\ ys\ Y\ y\ (igVar\ MOD\ ys\ y) = Y$

definition *igSubstIGVar2STR* **where**
$igSubstIGVar2STR\ MOD ==$
$(\forall\ ys\ y\ y1.$
$igSubst\ MOD\ ys\ (igVar\ MOD\ ys\ y1)\ y\ (igVar\ MOD\ ys\ y) = igVar\ MOD\ ys\ y1)$
$\wedge$
$(\forall\ ys\ y\ Y.$
$igWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$
$igSubst\ MOD\ ys\ Y\ y\ (igVar\ MOD\ ys\ y) = Y)$

lemma *igSubstIGVar2STR-imp-igSubstIGVar2*:
$igSubstIGVar2STR\ MOD \Longrightarrow igSubstIGVar2\ MOD$
$\langle proof \rangle$

definition *igSubstIGAbs* **where**
$igSubstIGAbs\ MOD ==$
$\forall\ ys\ y\ Y\ xs\ x\ s\ X.$
$isInBar\ (xs,s) \wedge igWls\ MOD\ (asSort\ ys)\ Y \wedge igWls\ MOD\ s\ X \longrightarrow$
$(xs \neq ys \vee x \neq y) \wedge igFresh\ MOD\ xs\ x\ Y \longrightarrow$
$igSubstAbs\ MOD\ ys\ Y\ y\ (igAbs\ MOD\ xs\ x\ X) =$
$igAbs\ MOD\ xs\ x\ (igSubst\ MOD\ ys\ Y\ y\ X)$

definition *igSubstIGAbsSTR* **where**
$igSubstIGAbsSTR\ MOD ==$
$\forall\ ys\ y\ Y\ xs\ x\ X.$
$(xs \neq ys \vee x \neq y) \wedge igFresh\ MOD\ xs\ x\ Y \longrightarrow$
$igSubstAbs\ MOD\ ys\ Y\ y\ (igAbs\ MOD\ xs\ x\ X) =$
$igAbs\ MOD\ xs\ x\ (igSubst\ MOD\ ys\ Y\ y\ X)$

lemma *igSubstIGAbsSTR-imp-igSubstIGAbs*:
$igSubstIGAbsSTR\ MOD \Longrightarrow igSubstIGAbs\ MOD$
$\langle proof \rangle$

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*
⟨*proof*⟩

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 ⟹ *igSubstCls MOD*

⟨*proof*⟩

definition *igAbsCongS* **where**
igAbsCongS MOD ==
∀ *xs x x' y s X X'*.
isInBar (*xs*,*s*) ∧ *igWls MOD s X* ∧ *igWls MOD s X'* ⟶
igFresh MOD xs y X ∧ *igFresh MOD xs y X'* ∧ *igSwap MOD xs y x X* = *igSwap MOD xs y x' X'* ⟶
igAbs MOD xs x X = *igAbs MOD xs x' X'*

definition *igAbsCongSSTR* **where**
igAbsCongSSTR MOD ==
∀ *xs x x' y X X'*.
igFresh MOD xs y X ∧ *igFresh MOD xs y X'* ∧ *igSwap MOD xs y x X* = *igSwap MOD xs y x' X'* ⟶
igAbs MOD xs x X = *igAbs MOD xs x' X'*

lemma *igAbsCongSSTR-imp-igAbsCongS*:
igAbsCongSSTR MOD ⟹ *igAbsCongS MOD*
⟨*proof*⟩

definition *igAbsCongU* **where**
igAbsCongU MOD ==
∀ *xs x x' y s X X'*.
isInBar (*xs*,*s*) ∧ *igWls MOD s X* ∧ *igWls MOD s X'* ⟶
igFresh MOD xs y X ∧ *igFresh MOD xs y X'* ∧
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'*

definition *igAbsCongUSTR* **where**
igAbsCongUSTR MOD ==
∀ *xs x x' y X X'*.
igFresh MOD xs y X ∧ *igFresh MOD xs y X'* ∧
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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *igAbsRenSTR-imp-igAbsCongUSTR*:
igAbsRenSTR MOD ⟹ igAbsCongUSTR MOD
⟨*proof*⟩

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 $\Longrightarrow$ *iwlsFSw MOD*
⟨*proof*⟩

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 $\Longrightarrow$ *iwlsFSb MOD*
⟨*proof*⟩

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 $\Longrightarrow$ *igSwapInpIPresIGWlsInp MOD*
⟨*proof*⟩

lemma *imp-igSwapInpIPresIGWlsInpSTR*:
igSwapIPresIGWlsSTR MOD $\Longrightarrow$ *igSwapInpIPresIGWlsInpSTR MOD*
⟨*proof*⟩

lemma *imp-igSubstInpIPresIGWlsInp*:
igSubstIPresIGWls MOD $\Longrightarrow$ *igSubstInpIPresIGWlsInp MOD*
⟨*proof*⟩

lemma *imp-igSubstInpIPresIGWlsInpSTR*:
igSubstIPresIGWlsSTR MOD $\Longrightarrow$ *igSubstInpIPresIGWlsInpSTR MOD*
⟨*proof*⟩

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 $\Longrightarrow$ *igSwapBinpIPresIGWlsBinp MOD*
⟨*proof*⟩

lemma *imp-igSwapBinpIPresIGWlsBinpSTR*:
igSwapAbsIPresIGWlsAbsSTR MOD $\Longrightarrow$ *igSwapBinpIPresIGWlsBinpSTR MOD*

⟨*proof*⟩

lemma *imp-igSubstBinpIPresIGWlsBinp*:
$igSubstAbsIPresIGWlsAbs\ MOD \Longrightarrow igSubstBinpIPresIGWlsBinp\ MOD$
⟨*proof*⟩

lemma *imp-igSubstBinpIPresIGWlsBinpSTR*:
$igSubstAbsIPresIGWlsAbsSTR\ MOD \Longrightarrow igSubstBinpIPresIGWlsBinpSTR\ MOD$
⟨*proof*⟩

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**
$ipresIGWlsAll\ h\ hA\ MOD\ MOD' ==$
$ipresIGWls\ h\ MOD\ MOD' \land\ ipresIGWlsAbs\ hA\ MOD\ MOD'$

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**
$ipresIGAbs\ h\ hA\ MOD\ MOD' ==$
$\forall\ xs\ x\ s\ X.\ isInBar\ (xs,s) \land igWls\ MOD\ s\ X \longrightarrow$
$hA\ (igAbs\ MOD\ xs\ x\ X) = igAbs\ MOD'\ xs\ x\ (h\ X)$

definition *ipresIGOp*
where
$ipresIGOp\ h\ hA\ MOD\ MOD' ==$

∀ *delta inp binp.*
igWlsInp MOD delta inp ∧ *igWlsBinp MOD delta binp* ⟶
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′*)
⟨*proof*⟩

Some facts for free inpus:

lemma *igSwapInp-None*[*simp*]:
(*igSwapInp MOD zs z1 z2 inp i* = *None*) = (*inp i* = *None*)
⟨*proof*⟩

lemma *igSubstInp-None*[*simp*]:
(*igSubstInp MOD ys Y y inp i* = *None*) = (*inp i* = *None*)
⟨*proof*⟩

lemma *imp-igWlsInp*:
igWlsInp MOD delta inp ⟹ *ipresIGWls h MOD MOD′*
⟹ *igWlsInp MOD′ delta* (*lift h inp*)
⟨*proof*⟩

corollary *FSwImorph-igWlsInp*:
assumes *igWlsInp MOD delta inp* **and** *FSwImorph h hA MOD MOD′*
shows *igWlsInp MOD′ delta* (*lift h inp*)
⟨*proof*⟩

corollary *FSbImorph-igWlsInp*:

assumes *igWlsInp MOD delta inp* **and** *FSbImorph h hA MOD MOD′*
shows *igWlsInp MOD′ delta* (*lift h inp*)
⟨*proof*⟩

lemma *FSwSbImorph-igWlsInp*:
assumes *igWlsInp MOD delta inp* **and** *FSwSbImorph h hA MOD MOD′*
shows *igWlsInp MOD′ delta* (*lift h inp*)
⟨*proof*⟩

Similar facts for bound inpus:

lemma *igSwapBinp-None*[*simp*]:
(*igSwapBinp MOD zs z1 z2 binp i* = *None*) = (*binp i* = *None*)
⟨*proof*⟩

lemma *igSubstBinp-None*[*simp*]:
(*igSubstBinp MOD ys Y y binp i* = *None*) = (*binp i* = *None*)
⟨*proof*⟩

lemma *imp-igWlsBinp*:
assumes ∗: *igWlsBinp MOD delta binp*
and ∗∗: *ipresIGWlsAbs hA MOD MOD′*
shows *igWlsBinp MOD′ delta* (*lift hA binp*)
⟨*proof*⟩

corollary *FSwImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSwImorph h hA MOD MOD′*
shows *igWlsBinp MOD′ delta* (*lift hA binp*)
⟨*proof*⟩

corollary *FSbImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSbImorph h hA MOD MOD′*
shows *igWlsBinp MOD′ delta* (*lift hA binp*)
⟨*proof*⟩

lemma *FSwSbImorph-igWlsBinp*:
assumes *igWlsBinp MOD delta binp* **and** *FSwSbImorph h hA MOD MOD′*
shows *igWlsBinp MOD′ delta* (*lift hA binp*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *id-FSbImorph*: *FSbImorph id id MOD MOD*

⟨*proof*⟩

lemma *id-FSwSbImorph*: *FSwSbImorph id id MOD MOD*
⟨*proof*⟩

lemma *comp-ipresIGWls*:
assumes *ipresIGWls h MOD MOD'* **and** *ipresIGWls h' MOD' MOD''*
shows *ipresIGWls* (*h' o h*) *MOD MOD''*
⟨*proof*⟩

lemma *comp-ipresIGWlsAbs*:
assumes *ipresIGWlsAbs hA MOD MOD'* **and** *ipresIGWlsAbs hA' MOD' MOD''*
shows *ipresIGWlsAbs* (*hA' o hA*) *MOD MOD''*
⟨*proof*⟩

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''*
⟨*proof*⟩

lemma *comp-ipresIGVar*:
assumes *ipresIGVar h MOD MOD'* **and** *ipresIGVar h' MOD' MOD''*
shows *ipresIGVar* (*h' o h*) *MOD MOD''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *igWlsAbs-termMOD*[*simp*]: *igWlsAbs termMOD* = *wlsAbs*
⟨*proof*⟩

lemma *igWlsInp-termMOD-wlsInp*[*simp*]:
igWlsInp termMOD delta inp = *wlsInp delta inp*
⟨*proof*⟩

lemma *igWlsBinp-termMOD-wlsBinp*[*simp*]:
igWlsBinp termMOD delta binp = *wlsBinp delta binp*
⟨*proof*⟩

lemmas *igWlsAll-termMOD-simps* =
igWls-termMOD igWlsAbs-termMOD
igWlsInp-termMOD-wlsInp igWlsBinp-termMOD-wlsBinp

lemma *igVar-termMOD*[*simp*]: *igVar termMOD* = *Var*
⟨*proof*⟩

lemma *igAbs-termMOD*[*simp*]: *igAbs termMOD* = *Abs*
⟨*proof*⟩

lemma *igOp-termMOD*[*simp*]: *igOp termMOD* = *Op*
⟨*proof*⟩

lemmas *igCons-termMOD-simps* =
igVar-termMOD igAbs-termMOD igOp-termMOD

lemma *igFresh-termMOD*[*simp*]: *igFresh termMOD* = *fresh*
⟨*proof*⟩

lemma *igFreshAbs-termMOD*[*simp*]: *igFreshAbs termMOD* = *freshAbs*
⟨*proof*⟩

lemma *igFreshInp-termMOD*[*simp*]: *igFreshInp termMOD* = *freshInp*
⟨*proof*⟩

lemma *igFreshBinp-termMOD*[*simp*]: *igFreshBinp termMOD* = *freshBinp*
⟨*proof*⟩

lemmas *igFreshAll-termMOD-simps* =
igFresh-termMOD igFreshAbs-termMOD
igFreshInp-termMOD igFreshBinp-termMOD

lemma *igSwap-termMOD*[*simp*]: *igSwap termMOD* = *swap*
⟨*proof*⟩

lemma *igSwapAbs-termMOD*[*simp*]: *igSwapAbs termMOD* = *swapAbs*
⟨*proof*⟩

lemma *igSwapInp-termMOD*[*simp*]: *igSwapInp termMOD* = *swapInp*
⟨*proof*⟩

lemma *igSwapBinp-termMOD*[*simp*]: *igSwapBinp termMOD* = *swapBinp*
⟨*proof*⟩

lemmas *igSwapAll-termMOD-simps* =
igSwap-termMOD igSwapAbs-termMOD
igSwapInp-termMOD igSwapBinp-termMOD

lemma *igSubst-termMOD*[*simp*]: *igSubst termMOD* = *subst*

⟨*proof*⟩

lemma *igSubstAbs-termMOD*[*simp*]: *igSubstAbs termMOD* = *substAbs*
⟨*proof*⟩

lemma *igSubstInp-termMOD*[*simp*]: *igSubstInp termMOD* = *substInp*
⟨*proof*⟩

lemma *igSubstBinp-termMOD*[*simp*]: *igSubstBinp termMOD* = *substBinp*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *termMOD-igWlsAbsDisj*: *igWlsAbsDisj termMOD*
⟨*proof*⟩

lemma *termMOD-igWlsAllDisj*: *igWlsAllDisj termMOD*
⟨*proof*⟩

Abstraction domains inhabited only within bound arities:

lemma *termMOD-igWlsAbsIsInBar*: *igWlsAbsIsInBar termMOD*
⟨*proof*⟩

The syntactic constructs preserve the domains:

lemma *termMOD-igVarIPresIGWls*: *igVarIPresIGWls termMOD*
⟨*proof*⟩

lemma *termMOD-igAbsIPresIGWls*: *igAbsIPresIGWls termMOD*
⟨*proof*⟩

lemma *termMOD-igOpIPresIGWls*: *igOpIPresIGWls termMOD*
⟨*proof*⟩

lemma *termMOD-igConsIPresIGWls*: *igConsIPresIGWls termMOD*

⟨*proof*⟩

Swap preserves the domains:

lemma *termMOD-igSwapIPresIGWls*: *igSwapIPresIGWls termMOD*
⟨*proof*⟩

lemma *termMOD-igSwapAbsIPresIGWlsAbs*: *igSwapAbsIPresIGWlsAbs termMOD*
⟨*proof*⟩

lemma *termMOD-igSwapAllIPresIGWlsAll*: *igSwapAllIPresIGWlsAll termMOD*
⟨*proof*⟩

"Subst" preserves the domains:

lemma *termMOD-igSubstIPresIGWls*: *igSubstIPresIGWls termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstAbsIPresIGWlsAbs*: *igSubstAbsIPresIGWlsAbs termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstAllIPresIGWlsAll*: *igSubstAllIPresIGWlsAll termMOD*
⟨*proof*⟩

The "fresh" clauses hold:

lemma *termMOD-igFreshIGVar*: *igFreshIGVar termMOD*
⟨*proof*⟩

lemma *termMOD-igFreshIGAbs1*: *igFreshIGAbs1 termMOD*
⟨*proof*⟩

lemma *termMOD-igFreshIGAbs2*: *igFreshIGAbs2 termMOD*
⟨*proof*⟩

lemma *termMOD-igFreshIGOp*: *igFreshIGOp termMOD*
⟨*proof*⟩

lemma *termMOD-igFreshCls*: *igFreshCls termMOD*
⟨*proof*⟩

The "swap" clauses hold:

lemma *termMOD-igSwapIGVar*: *igSwapIGVar termMOD*
⟨*proof*⟩

lemma *termMOD-igSwapIGAbs*: *igSwapIGAbs termMOD*
⟨*proof*⟩

lemma *termMOD-igSwapIGOp*: *igSwapIGOp termMOD*
⟨*proof*⟩

lemma *termMOD-igSwapCls*: *igSwapCls termMOD*
⟨*proof*⟩

The "subst" clauses hold:

lemma *termMOD-igSubstIGVar1*: *igSubstIGVar1 termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstIGVar2*: *igSubstIGVar2 termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstIGAbs*: *igSubstIGAbs termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstIGOp*: *igSubstIGOp termMOD*
⟨*proof*⟩

lemma *termMOD-igSubstCls*: *igSubstCls termMOD*
⟨*proof*⟩

The swap-congruence clause for abstractions holds:

lemma *termMOD-igAbsCongS*: *igAbsCongS termMOD*
⟨*proof*⟩

The subst-renaming clause for abstractions holds:

lemma *termMOD-igAbsRen*: *igAbsRen termMOD*
⟨*proof*⟩

lemma *termMOD-iwlsFSw*: *iwlsFSw termMOD*
⟨*proof*⟩

lemma *termMOD-iwlsFSb*: *iwlsFSb termMOD*
⟨*proof*⟩

lemma *termMOD-iwlsFSwSb*: *iwlsFSwSb termMOD*
⟨*proof*⟩

lemma *termMOD-iwlsFSbSw*: *iwlsFSbSw termMOD*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *FSwSbImorph-termMOD*[*simp*]:
FSwSbImorph h hA termMOD MOD = termFSwSbImorph h hA MOD
⟨*proof*⟩

lemma *ipresWls-wlsInp*:
assumes *wlsInp delta inp* **and** *ipresWls h MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
⟨*proof*⟩

lemma *termFSwImorph-wlsInp*:
assumes *wlsInp delta inp* **and** *termFSwImorph h hA MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
⟨*proof*⟩

lemma *termFSwSbImorph-wlsInp*:
assumes *wlsInp delta inp* **and** *termFSwSbImorph h hA MOD*
shows *igWlsInp MOD delta* (*lift h inp*)
⟨*proof*⟩

lemma *ipresWls-wlsBinp*:
assumes *wlsBinp delta binp* **and** *ipresWlsAbs hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
⟨*proof*⟩

lemma *termFSwImorph-wlsBinp*:
assumes *wlsBinp delta binp* **and** *termFSwImorph h hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
⟨*proof*⟩

lemma *termFSwSbImorph-wlsBinp*:
assumes *wlsBinp delta binp* **and** *termFSwSbImorph h hA MOD*
shows *igWlsBinp MOD delta* (*lift hA binp*)
⟨*proof*⟩

lemma *id-termFSwImorph*: *termFSwImorph id id termMOD*
⟨*proof*⟩

lemma *id-termFSbImorph*: *termFSbImorph id id termMOD*
⟨*proof*⟩

lemma *id-termFSwSbImorph*: *termFSwSbImorph id id termMOD*
⟨*proof*⟩

lemma *comp-termFSwImorph*:
assumes ∗: *termFSwImorph h hA MOD* **and** ∗∗: *FSwImorph h′ hA′ MOD MOD′*
shows *termFSwImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
⟨*proof*⟩

lemma *comp-termFSbImorph*:
assumes ∗: *termFSbImorph h hA MOD* **and** ∗∗: *FSbImorph h′ hA′ MOD MOD′*
shows *termFSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
⟨*proof*⟩

lemma *comp-termFSwSbImorph*:
assumes ∗: *termFSwSbImorph h hA MOD* **and** ∗∗: *FSwSbImorph h′ hA′ MOD MOD′*
shows *termFSwSbImorph* (*h′ o h*) (*hA′ o hA*) *MOD′*
⟨*proof*⟩

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*⟩

lemma *ipresCons-imp-ipresFreshAll*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igFreshCls MOD*
and *igConsIPresIGWls MOD*
shows *ipresFreshAll h hA MOD*
⟨*proof*⟩

lemma *ipresCons-imp-ipresSwapAll*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSwapCls MOD*
and *igConsIPresIGWls MOD*
shows *ipresSwapAll h hA MOD*
⟨*proof*⟩

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*⟩

lemma *ipresCons-imp-ipresSubst*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubst h MOD*
⟨*proof*⟩

lemma *ipresCons-imp-ipresSubstAbs*:
assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubstAbs h hA MOD*
⟨*proof*⟩

lemma *ipresCons-imp-ipresSubstAll*:

assumes ∗: *ipresCons h hA MOD* **and** ∗∗: *igSubstCls MOD*
and *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *ipresSubstAll h hA MOD*
⟨*proof*⟩

lemma *iwlsFSw-termFSwImorph-iff*:
iwlsFSw MOD ⟹ *termFSwImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

corollary *iwlsFSwSTR-termFSwImorph-iff*:
iwlsFSwSTR MOD ⟹ *termFSwImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

lemma *iwlsFSb-termFSbImorph-iff*:
iwlsFSb MOD ⟹ *termFSbImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

corollary *iwlsFSbSwTR-termFSbImorph-iff*:
iwlsFSbSwTR MOD ⟹ *termFSbImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

lemma *iwlsFSwSb-termFSwSbImorph-iff*:
iwlsFSwSb MOD ⟹ *termFSwSbImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

lemma *iwlsFSbSw-termFSwSbImorph-iff*:
iwlsFSbSw MOD ⟹ *termFSwSbImorph h hA MOD* = *ipresCons h hA MOD*
⟨*proof*⟩

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*:
$eX \neq ERR \Longrightarrow (EX!\ X.\ eX = OK\ X)$
⟨*proof*⟩

lemma *check-OK*[*simp*]:
check (*OK X*) = *X*
⟨*proof*⟩

lemma *OK-check*[*simp*]:
$eX \neq ERR \Longrightarrow OK\ (check\ eX) = eX$
⟨*proof*⟩

lemma *checkI-OKI*[*simp*]:
checkI (*OKI inp*) = *inp*
⟨*proof*⟩

lemma *OKI-checkI*[*simp*]:
assumes *liftAll* (λ *X. X* $\neq$ *ERR*) *einp*
shows *OKI* (*checkI einp*) = *einp*
⟨*proof*⟩

lemma *OKI-inj*[*simp*]:
fixes *inp inp'* :: (*'index,'gTerm*)*input*
shows (*OKI inp* = *OKI inp'*) = (*inp* = *inp'*)
⟨*proof*⟩

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 = λxs x eX.
if (eX ≠ ERR ∧ (∃ s. isInBar (xs,s) ∧ igWls MOD s (check eX)))
then OK (igAbs MOD xs x (check eX))
else ERR,
igOp = λdelta einp ebinp.
if liftAll (λ X. X ≠ ERR) einp ∧ liftAll (λ A. A ≠ ERR) ebinp
∧ igWlsInp MOD delta (checkI einp) ∧ igWlsBinp MOD delta (checkI ebinp)
then OK (igOp MOD delta (checkI einp) (checkI ebinp))
else ERR,
igFresh = λys y eX.
if eX ≠ ERR ∧ (∃ 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*
⟨*proof*⟩

lemma *eWls-simp2*[*simp*]:
eWls MOD s ERR = *False*
⟨*proof*⟩

lemma *eWlsAbs-simp1*[*simp*]:
eWlsAbs MOD (*us,s*) (*OK A*) = *igWlsAbs MOD* (*us,s*) *A*
⟨*proof*⟩

lemma *eWlsAbs-simp2*[*simp*]:
eWlsAbs MOD (*us,s*) *ERR* = *False*
⟨*proof*⟩

lemma *eWlsInp-simp1*[*simp*]:
eWlsInp MOD delta (*OKI inp*) = *igWlsInp MOD delta inp*
⟨*proof*⟩

lemma *eWlsInp-simp2*[*simp*]:
¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) *einp* ⟹ ¬ *eWlsInp MOD delta einp*
⟨*proof*⟩

corollary *eWlsInp-simp3*[*simp*]:
¬ *eWlsInp MOD delta* (λ*i*. *Some ERR*)
⟨*proof*⟩

lemma *eWlsBinp-simp1*[*simp*]:
eWlsBinp MOD delta (*OKI binp*) = *igWlsBinp MOD delta binp*
⟨*proof*⟩

lemma *eWlsBinp-simp2*[*simp*]:
¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) *ebinp* ⟹ ¬ *eWlsBinp MOD delta ebinp*
⟨*proof*⟩

corollary *eWlsBinp-simp3*[*simp*]:
¬ *eWlsBinp MOD delta* (λ*i*. *Some ERR*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eAbs-simp1*[*simp*]:
⟦*isInBar* (*xs*,*s*); *igWls MOD s X*⟧ ⟹ *eAbs MOD xs x* (*OK X*) = *OK* (*igAbs MOD xs x X*)
⟨*proof*⟩

lemma *eAbs-simp2*[*simp*]:
∀ *s*. ¬ (*isInBar* (*xs*,*s*) ∧ *igWls MOD s X*) ⟹ *eAbs MOD xs x* (*OK X*) = *ERR*
⟨*proof*⟩

lemma *eAbs-simp3*[*simp*]:
eAbs MOD xs x ERR = *ERR*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eOp-simp2*[*simp*]:
assumes ¬ *igWlsInp MOD delta inp*
shows *eOp MOD delta* (*OKI inp*) *ebinp* = *ERR*
⟨*proof*⟩

lemma *eOp-simp3*[*simp*]:
assumes ¬ *igWlsBinp MOD delta binp*
shows *eOp MOD delta einp* (*OKI binp*) = *ERR*

⟨*proof*⟩

lemma *eOp-simp4*[*simp*]:
assumes ¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) *einp*
shows *eOp MOD delta einp ebinp* = *ERR*
⟨*proof*⟩

corollary *eOp-simp5*[*simp*]:
eOp MOD delta (λ*i*. *Some ERR*) *ebinp* = *ERR*
⟨*proof*⟩

lemma *eOp-simp6*[*simp*]:
assumes ¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) *ebinp*
shows *eOp MOD delta einp ebinp* = *ERR*
⟨*proof*⟩

corollary *eOp-simp7*[*simp*]:
eOp MOD delta einp (λ*i*. *Some ERR*) = *ERR*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eFresh-simp2*[*simp*]:
∀ *s*. ¬ *igWls MOD s X* ⟹ *eFresh MOD ys y* (*OK X*)
⟨*proof*⟩

lemma *eFresh-simp3*[*simp*]:
eFresh MOD ys y ERR
⟨*proof*⟩

lemma *eFreshAbs-simp1*[*simp*]:
igWlsAbs MOD (*us*,*s*) *A* ⟹ *eFreshAbs MOD ys y* (*OK A*) = *igFreshAbs MOD ys y A*
⟨*proof*⟩

lemma *eFreshAbs-simp2*[*simp*]:
∀ *us s*. ¬ *igWlsAbs MOD* (*us*,*s*) *A* ⟹ *eFreshAbs MOD ys y* (*OK A*)
⟨*proof*⟩

lemma *eFreshAbs-simp3*[*simp*]:
eFreshAbs MOD ys y ERR
⟨*proof*⟩

lemma *eFreshInp-simp*[*simp*]:
igWlsInp MOD delta inp
$\Longrightarrow$ *eFreshInp MOD ys y* (*OKI inp*) = *igFreshInp MOD ys y inp*
⟨*proof*⟩

lemma *eFreshBinp-simp*[*simp*]:
igWlsBinp MOD delta binp
$\Longrightarrow$ *eFreshBinp MOD ys y* (*OKI binp*) = *igFreshBinp MOD ys y binp*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eSwap-simp2*[*simp*]:
∀ *s*. ¬ *igWls MOD s X* $\Longrightarrow$ *eSwap MOD zs z1 z2* (*OK X*) = *ERR*
⟨*proof*⟩

lemma *eSwap-simp3*[*simp*]:
eSwap MOD zs z1 z2 ERR = *ERR*
⟨*proof*⟩

lemma *eSwapAbs-simp1*[*simp*]:
igWlsAbs MOD (*us*,*s*) *A*
$\Longrightarrow$ *eSwapAbs MOD zs z1 z2* (*OK A*) = *OK* (*igSwapAbs MOD zs z1 z2 A*)
⟨*proof*⟩

lemma *eSwapAbs-simp2*[*simp*]:
∀ *us s*. ¬ *igWlsAbs MOD* (*us*,*s*) *A* $\Longrightarrow$ *eSwapAbs MOD zs z1 z2* (*OK A*) = *ERR*
⟨*proof*⟩

lemma *eSwapAbs-simp3*[*simp*]:
eSwapAbs MOD zs z1 z2 ERR = *ERR*
⟨*proof*⟩

lemma *eSwapInp-simp1*[*simp*]:
igWlsInp MOD delta inp
$\Longrightarrow$ *eSwapInp MOD zs z1 z2* (*OKI inp*) = *OKI* (*igSwapInp MOD zs z1 z2 inp*)
⟨*proof*⟩

lemma *eSwapInp-simp2*[*simp*]:

assumes ¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) *einp*
shows ¬ *liftAll* (λ *eX*. *eX* ≠ *ERR*) (*eSwapInp MOD zs z1 z2 einp*)
⟨*proof*⟩

lemma *eSwapBinp-simp1*[*simp*]:
igWlsBinp MOD delta binp
⟹ *eSwapBinp MOD zs z1 z2* (*OKI binp*) = *OKI* (*igSwapBinp MOD zs z1 z2 binp*)
⟨*proof*⟩

lemma *eSwapBinp-simp2*[*simp*]:
assumes ¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) *ebinp*
shows ¬ *liftAll* (λ *eA*. *eA* ≠ *ERR*) (*eSwapBinp MOD zs z1 z2 ebinp*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eSubst-simp2*[*simp*]:
¬ *igWls MOD* (*asSort ys*) *Y* ⟹ *eSubst MOD ys* (*OK Y*) *y eX* = *ERR*
⟨*proof*⟩

lemma *eSubst-simp3*[*simp*]:
∀ *s*. ¬ *igWls MOD s X* ⟹ *eSubst MOD ys eY y* (*OK X*) = *ERR*
⟨*proof*⟩

lemma *eSubst-simp4*[*simp*]:
eSubst MOD ys eY y ERR = *ERR*
⟨*proof*⟩

lemma *eSubst-simp5*[*simp*]:
eSubst MOD ys ERR y eX = *ERR*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eSubstAbs-simp2*[*simp*]:
¬ *igWls MOD* (*asSort ys*) *Y* ⟹ *eSubstAbs MOD ys* (*OK Y*) *y eA* = *ERR*

⟨*proof*⟩

lemma *eSubstAbs-simp3*[*simp*]:
$\forall$ *us s.* $\neg$ *igWlsAbs MOD* (*us*,*s*) *A* $\Longrightarrow$ *eSubstAbs MOD ys eY y* (*OK A*) = *ERR*
⟨*proof*⟩

lemma *eSubstAbs-simp4*[*simp*]:
eSubstAbs MOD ys eY y ERR = *ERR*
⟨*proof*⟩

lemma *eSubstAbs-simp5*[*simp*]:
eSubstAbs MOD ys ERR y eA = *ERR*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eSubstInp-simp2*[*simp*]:
assumes $\neg$ *liftAll* (λ*eX. eX* $\neq$ *ERR*) *einp*
shows $\neg$ *liftAll* (λ*eX. eX* $\neq$ *ERR*) (*eSubstInp MOD ys eY y einp*)
⟨*proof*⟩

lemma *eSubstInp-simp3*[*simp*]:
assumes $*$: $\neg$ *igWls MOD* (*asSort ys*) *Y* **and** $**$: $\neg$ *einp* = (λ *i. None*)
shows $\neg$ *liftAll* (λ*eX. eX* $\neq$ *ERR*) (*eSubstInp MOD ys* (*OK Y*) *y einp*)
⟨*proof*⟩

lemma *eSubstInp-simp4*[*simp*]:
assumes $\neg$ *einp* = (λ *i. None*)
shows $\neg$ *liftAll* (λ*eX. eX* $\neq$ *ERR*) (*eSubstInp MOD ys ERR y einp*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eSubstBinp-simp2*[*simp*]:
assumes $\neg$ *liftAll* (λ*eA. eA* $\neq$ *ERR*) *ebinp*
shows $\neg$ *liftAll* (λ*eA. eA* $\neq$ *ERR*) (*eSubstBinp MOD ys eY y ebinp*)
⟨*proof*⟩

lemma *eSubstBinp-simp3*[*simp*]:
assumes $*$: $\neg$ *igWls MOD* (*asSort ys*) *Y* **and** $**$: $\neg$ *ebinp* = (λ *i. None*)
shows $\neg$ *liftAll* (λ*eA. eA* $\neq$ *ERR*) (*eSubstBinp MOD ys* (*OK Y*) *y ebinp*)

⟨*proof*⟩

lemma *eSubstBinp-simp4* [*simp*]:
assumes ¬ *ebinp* = (λ *i*. *None*)
shows ¬ *liftAll* (λ*eA*. *eA* ≠ *ERR*) (*eSubstBinp MOD ys ERR y ebinp*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eWlsAbs-nchotomy*:
(∃ *A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*) ∨ ¬ *eWlsAbs MOD* (*us*,*s*) *eA*
⟨*proof*⟩

lemma *eAbs-nchotomy*:
((∃ *s X*. *eX* = *OK X* ∧ *isInBar* (*xs*,*s*) ∧ *igWls MOD s X*)) ∨ (*eAbs MOD xs x eX* = *ERR*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eFresh-nchotomy*:
(∃ *s X*. *eX* = *OK X* ∧ *igWls MOD s X*) ∨ *eFresh MOD ys y eX*
⟨*proof*⟩

lemma *eFreshAbs-nchotomy*:
(∃ *us s A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*)

∨ *eFreshAbs MOD ys y eA*
⟨*proof*⟩

lemma *eSwap-nchotomy*:
(∃ *s X*. *eX* = *OK X* ∧ *igWls MOD s X*) ∨
(*eSwap MOD zs z1 z2 eX* = *ERR*)
⟨*proof*⟩

lemma *eSwapAbs-nchotomy*:
(∃ *us s A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*) ∨
(*eSwapAbs MOD zs z1 z2 eA* = *ERR*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

8.4.5 Inversion rules

lemma *eWls-invert*:
assumes *eWls MOD s eX*
shows ∃ *X*. *eX* = *OK X* ∧ *igWls MOD s X*
⟨*proof*⟩

lemma *eWlsAbs-invert*:
assumes *eWlsAbs MOD* (*us*,*s*) *eA*
shows ∃ *A*. *eA* = *OK A* ∧ *igWlsAbs MOD* (*us*,*s*) *A*
⟨*proof*⟩

lemma *eWlsInp-invert*:
assumes *eWlsInp MOD delta einp*
shows ∃ *inp*. *igWlsInp MOD delta inp* ∧ *einp* = *OKI inp*
⟨*proof*⟩

lemma *eWlsBinp-invert*:
assumes *eWlsBinp MOD delta ebinp*
shows ∃ *binp*. *igWlsBinp MOD delta binp* ∧ *ebinp* = *OKI binp*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *errMOD-igWlsAbsDisj*:
assumes *igWlsAbsDisj MOD*
shows *igWlsAbsDisj* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igWlsAllDisj*:
assumes *igWlsAllDisj MOD*
shows *igWlsAllDisj* (*errMOD MOD*)
⟨*proof*⟩

Only "bound arity" abstraction domains are inhabited:

lemma *errMOD-igWlsAbsIsInBar*:
assumes *igWlsAbsIsInBar MOD*
shows *igWlsAbsIsInBar* (*errMOD MOD*)
⟨*proof*⟩

The operators preserve the domains strongly:

lemma *errMOD-igVarIPresIGWlsSTR*:
assumes *igVarIPresIGWls MOD*
shows *igVarIPresIGWls* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igAbsIPresIGWlsSTR*:
assumes ∗: *igAbsIPresIGWls MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igAbsIPresIGWlsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igOpIPresIGWlsSTR*:
fixes *MOD* :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model*
assumes *igOpIPresIGWls MOD*
shows *igOpIPresIGWlsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igConsIPresIGWlsSTR*:
assumes *igConsIPresIGWls MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igConsIPresIGWlsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSwapIPresIGWlsSTR*:
assumes *igSwapIPresIGWls MOD* **and** *igWlsDisj MOD*
shows *igSwapIPresIGWlsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSwapAbsIPresIGWlsAbsSTR*:
assumes ∗: *igSwapAbsIPresIGWlsAbs MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igSwapAbsIPresIGWlsAbsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSwapAllIPresIGWlsAllSTR*:
assumes *igSwapAllIPresIGWlsAll MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igSwapAllIPresIGWlsAllSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSubstIPresIGWlsSTR*:
assumes *igSubstIPresIGWls MOD* **and** *igWlsDisj MOD*
shows *igSubstIPresIGWlsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSubstAbsIPresIGWlsAbsSTR*:
assumes ∗: *igSubstAbsIPresIGWlsAbs MOD* **and** ∗∗: *igWlsAbsDisj MOD*
and ∗∗∗: *igWlsAbsIsInBar MOD*
shows *igSubstAbsIPresIGWlsAbsSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSubstAllIPresIGWlsAllSTR*:
assumes *igSubstAllIPresIGWlsAll MOD* **and** *igWlsAllDisj MOD*
and *igWlsAbsIsInBar MOD*
shows *igSubstAllIPresIGWlsAllSTR* (*errMOD MOD*)
⟨*proof*⟩

The strong “fresh" clauses are satisfied:

lemma *errMOD-igFreshIGVarSTR*:
assumes *igVarIPresIGWls MOD* **and** *igFreshIGVar MOD*
shows *igFreshIGVar* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igFreshIGAbs1STR*:
assumes ∗: *igAbsIPresIGWls MOD* **and** ∗∗: *igFreshIGAbs1 MOD*
shows *igFreshIGAbs1STR* (*errMOD MOD*)

⟨*proof*⟩

lemma *errMOD-igFreshIGAbs2STR*:
assumes *igAbsIPresIGWls MOD* **and** *igFreshIGAbs2 MOD*
shows *igFreshIGAbs2STR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igFreshIGOpSTR*:
fixes *MOD* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'var*,*'gTerm*,*'gAbs*)*model*
assumes *igOpIPresIGWls MOD* **and** *igFreshIGOp MOD*
shows *igFreshIGOpSTR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igFreshClsSTR*:
assumes *igConsIPresIGWls MOD* **and** *igFreshCls MOD*
shows *igFreshClsSTR* (*errMOD MOD*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *errMOD-igSwapIGAbsSTR*:
assumes ∗: *igAbsIPresIGWls MOD* **and** ∗∗: *igWlsDisj MOD*
and ∗∗∗: *igSwapIPresIGWls MOD* **and** ∗∗∗∗: *igSwapIGAbs MOD*
shows *igSwapIGAbsSTR* (*errMOD MOD*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

The strong “subst" clauses are satisfied:

lemma *errMOD-igSubstIGVar1STR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIGVar1 MOD*
shows *igSubstIGVar1STR* (*errMOD MOD*)
⟨*proof*⟩

lemma *errMOD-igSubstIGVar2STR*:
assumes *igVarIPresIGWls MOD* **and** *igSubstIGVar2 MOD*
shows *igSubstIGVar2STR* (*errMOD MOD*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *errMOD-igSubstClsSTR*:
assumes *igWlsAllDisj MOD* **and** *igConsIPresIGWls MOD*
and *igWlsAbsIsInBar MOD*
and *igSubstAllIPresIGWlsAll MOD* **and** *igSubstCls MOD*
shows *igSubstClsSTR* (*errMOD MOD*)
⟨*proof*⟩

Strong swap-based congruence for abstractions holds:

lemma *errMOD-igAbsCongSSTR*:
assumes *igSwapIPresIGWls MOD* **and** *igWlsDisj MOD* **and** *igAbsCongS MOD*
shows *igAbsCongSSTR* (*errMOD MOD*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *check-ipresIGWlsAbs*:
ipresIGWlsAbs check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGWlsAll*:
ipresIGWlsAll check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

Preservation of the operations:

lemma *check-ipresIGVar*:
ipresIGVar check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGAbs*:
ipresIGAbs check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGOp*:
ipresIGOp check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGCons*:
ipresIGCons check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGFresh*:
ipresIGFresh check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGFreshAbs*:
ipresIGFreshAbs check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGFreshAll*:
ipresIGFreshAll check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSwap*:
ipresIGSwap check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSwapAbs*:
ipresIGSwapAbs check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSwapAll*:
ipresIGSwapAll check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSubst*:
ipresIGSubst check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSubstAbs*:
ipresIGSubstAbs check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

lemma *check-ipresIGSubstAll*:
ipresIGSubstAll check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

“check" is a fresh-swap morphism:

lemma *check-FSwImorph*:
FSwImorph check check (*errMOD MOD*) *MOD*
⟨*proof*⟩

“check" is a fresh-subst morphism:

lemma *check-FSbImorph*:
FSbImorph check check (*errMOD MOD*) *MOD*

⟨*proof*⟩

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*)
⟨*proof*⟩

function
qInit :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model* ⇒
(*'index,'bindex,'varSort,'var,'opSym*)*qTerm* ⇒ *'gTerm*
and
qInitAbs :: (*'index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs*)*model* ⇒
(*'index,'bindex,'varSort,'var,'opSym*)*qAbs* ⇒ *'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*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *qFreshAll-igFreshAll-qInitAll*:
assumes *igFreshClsSTR 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*))
⟨*proof*⟩

corollary *iwlsFSwSTR-qFreshAll-igFreshAll-qInitAll*:
assumes *iwlsFSwSTR 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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *qSwapAll-igSwapAll-qInitAll*:
assumes *igSwapClsSTR MOD*
shows
qInit MOD (*qX* #[[*z1* $\wedge$ *z2*]]-*zs*) = *igSwap MOD zs z1 z2* (*qInit MOD qX*) $\wedge$
qInitAbs MOD (*qA* $[[*z1* $\wedge$ *z2*]]-*zs*) = *igSwapAbs MOD zs z1 z2* (*qInitAbs MOD qA*)
⟨*proof*⟩

corollary *iwlsFSwSTR-qSwapAll-igSwapAll-qInitAll*:
assumes *wls*: *iwlsFSwSTR MOD*
shows
qInit MOD (*qX* #[[*z1* $\wedge$ *z2*]]-*zs*) = *igSwap MOD zs z1 z2* (*qInit MOD qX*) $\wedge$
qInitAbs MOD (*qA* $[[*z1* $\wedge$ *z2*]]-*zs*) = *igSwapAbs MOD zs z1 z2* (*qInitAbs MOD qA*)
⟨*proof*⟩

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* $\longrightarrow$
(∀ *ys y1 y*.
qAFresh ys y1 qX $\longrightarrow$
qInit MOD (*qX* #[[*y1* $\wedge$ *y*]]-*ys*) = *igSubst MOD ys* (*igVar MOD ys y1*) *y* (*qInit MOD qX*)))
$\wedge$

($qGoodAbs\ qA \longrightarrow$
($\forall$ *ys y1 y.*
qAFreshAbs ys y1 qA $\longrightarrow$
qInitAbs MOD (*qA* $[[*y1* $\wedge$ *y*]]-*ys*) = *igSubstAbs MOD ys* (*igVar MOD ys y1*) *y* (*qInitAbs MOD qA*)))
⟨*proof*⟩

lemma *iwlsFSbSwTR-qSwapAll-igSubstAll-qInitAll*:
assumes *wls*: *iwlsFSbSwTR MOD*
shows
(*qGood qX* $\longrightarrow$
qAFresh ys y1 qX $\longrightarrow$
qInit MOD (*qX* #[[*y1* $\wedge$ *y*]]-*ys*) = *igSubst MOD ys* (*igVar MOD ys y1*) *y* (*qInit MOD qX*))
$\wedge$
(*qGoodAbs qA* $\longrightarrow$
qAFreshAbs ys y1 qA $\longrightarrow$
qInitAbs MOD (*qA* $[[*y1* $\wedge$ *y*]]-*ys*) = *igSubstAbs MOD ys* (*igVar MOD ys y1*) *y* (*qInitAbs MOD qA*))
⟨*proof*⟩

lemma *iwlsFSwSTR-alphaAll-qInitAll*:
assumes *iwlsFSwSTR MOD*
shows
($\forall$ *qX'*. *qX* #= *qX'* $\longrightarrow$ *qInit MOD qX* = *qInit MOD qX'*) $\wedge$
($\forall$ *qA'*. *qA* $= *qA'* $\longrightarrow$ *qInitAbs MOD qA* = *qInitAbs MOD qA'*)
⟨*proof*⟩

corollary *iwlsFSwSTR-qInit-respectsP-alpha*:
assumes *iwlsFSwSTR MOD* **shows** (*qInit MOD*) *respectsP alpha*
⟨*proof*⟩

corollary *iwlsFSwSTR-qInitAbs-respectsP-alphaAbs*:
assumes *iwlsFSwSTR MOD* **shows** (*qInitAbs MOD*) *respectsP alphaAbs*
⟨*proof*⟩

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* $\longrightarrow$ ($\forall$ *qX'*. *qX* #= *qX'* $\longrightarrow$ *qInit MOD qX* = *qInit MOD qX'*)) $\wedge$
(*qGoodAbs qA* $\longrightarrow$ ($\forall$ *qA'*. *qA* $= *qA'* $\longrightarrow$ *qInitAbs MOD qA* = *qInitAbs MOD qA'*))
⟨*proof*⟩

corollary *iwlsFSbSwTR-qInit-respectsP-alphaGood*:
assumes *iwlsFSbSwTR MOD*
shows (*qInit MOD*) *respectsP alphaGood*

⟨*proof*⟩

corollary *iwlsFSbSwTR-qInitAbs-respectsP-alphaAbsGood*:
assumes *iwlsFSbSwTR MOD*
shows (*qInitAbs MOD*) *respectsP alphaAbsGood*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *iwlsFSbSwTR-iterSTR-ipresVar*:
assumes *iwlsFSbSwTR MOD*
shows *ipresVar* (*iterSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSwSTR-iterSTR-ipresAbs*:
assumes *iwlsFSwSTR MOD*
shows *ipresAbs* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSbSwTR-iterSTR-ipresAbs*:
assumes *iwlsFSbSwTR MOD*
shows *ipresAbs* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSwSTR-iterSTR-ipresOp*:
assumes *iwlsFSwSTR MOD*
shows *ipresOp* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSbSwTR-iterSTR-ipresOp*:
assumes *iwlsFSbSwTR MOD*
shows *ipresOp* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSwSTR-iterSTR-ipresCons*:

assumes *iwlsFSwSTR MOD*
shows *ipresCons* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSbSwTR-iterSTR-ipresCons*:
assumes *iwlsFSbSwTR MOD*
shows *ipresCons* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

lemma *iwlsFSwSTR-iterSTR-termFSwImorph*:
assumes *iwlsFSwSTR MOD*
shows *termFSwImorph* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

corollary *iterSTR-termFSwImorph-errMOD*:
assumes *iwlsFSw MOD*
shows
termFSwImorph (*iterSTR* (*errMOD MOD*))
(*iterAbsSTR* (*errMOD MOD*))
(*errMOD MOD*)
⟨*proof*⟩

lemma *iwlsFSbSwTR-iterSTR-termFSbImorph*:
assumes *iwlsFSbSwTR MOD*
shows *termFSbImorph* (*iterSTR MOD*) (*iterAbsSTR MOD*) *MOD*
⟨*proof*⟩

corollary *iterSTR-termFSbImorph-errMOD*:
assumes *iwlsFSb MOD*
shows
termFSbImorph (*iterSTR* (*errMOD MOD*))
(*iterAbsSTR* (*errMOD MOD*))
(*errMOD MOD*)
⟨*proof*⟩

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 ⟹ *termFSwImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
⟨*proof*⟩

theorem *iwlsFSb-iterAll-termFSbImorph*:
iwlsFSb MOD ⟹ *termFSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
⟨*proof*⟩

theorem *iwlsFSwSb-iterAll-termFSwSbImorph*:
iwlsFSwSb MOD ⟹ *termFSwSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
⟨*proof*⟩

theorem *iwlsFSbSw-iterAll-termFSwSbImorph*:
iwlsFSbSw MOD ⟹ *termFSwSbImorph* (*iter MOD*) (*iterAbs MOD*) *MOD*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

theorem *iwlsFSb-iterAll-unique-ipresCons*:
assumes *iwlsFSb MOD* **and** *ipresCons h hA MOD*
shows
(*wls s X* ⟶ *h X* = *iter MOD X*) ∧
(*wlsAbs* (*us*,*s'*) *A* ⟶ *hA A* = *iterAbs MOD A*)
⟨*proof*⟩

theorem *iwlsFSwSb-iterAll-unique-ipresCons*:
assumes *iwlsFSwSb MOD* **and** *ipresCons h hA MOD*

shows
$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$
$\langle proof \rangle$

theorem *iwlsFSbSw-iterAll-unique-ipresCons*:
assumes $*$: *iwlsFSbSw MOD* **and** $**$: *ipresCons h hA MOD*
shows
$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$
$\langle proof \rangle$

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:

[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.

- 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) =
(isInBar (xs,s) ∧ xs = xs' ∧
(∀ 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 ≡
wlsOpS delta ∧ sameDom (arOf delta) sinp ∧ liftAll2 (sWls SEM) (arOf delta) sinp

[7] In other words, syntactic bindings are captured semantically as functional bindings.

definition *sWlsBinp* **where**
sWlsBinp SEM delta sbinp ≡
wlsOpS delta ∧ *sameDom* (*barOf delta*) *sbinp* ∧ *liftAll2* (*sWlsAbs SEM*) (*barOf delta*) *sbinp*

definition *sWlsNE* **where**
sWlsNE SEM ≡
∀ *s*. ∃ *sX*. *sWls SEM s sX*

definition *sWlsDisj* **where**
sWlsDisj SEM ≡
∀ *s s′ sX*. *sWls SEM s sX* ∧ *sWls SEM s′ sX* ⟶ *s* = *s′*

definition *sOpPrSWls* **where**
sOpPrSWls SEM ≡
∀ *delta sinp sbinp*.
sWlsInp SEM delta sinp ∧ *sWlsBinp SEM delta sbinp*
⟶ *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 ≡
sWlsNE SEM ∧ *sWlsDisj SEM* ∧ *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 ≡
∀ *us s*. *isInBar* (*us*,*s*) ⟶ (∃ *sA*. *sWlsAbs SEM* (*us*,*s*) *sA*)

definition *sWlsAbsDisj* **where**
sWlsAbsDisj SEM ≡
∀ *us s us′ s′ sA*.
isInBar (*us*,*s*) ∧ *isInBar* (*us′*,*s′*) ∧ *sWlsAbs SEM* (*us*,*s*) *sA* ∧ *sWlsAbs SEM* (*us′*,*s′*) *sA*
⟶ *us* = *us′* ∧ *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* ⇒

[8]As usual in Isabelle, we first define the "raw" version, and then "fix" it with a well-structuredness predicate.

$('varSort,'var,'sTerm)val$ (- '(- := -)'-- *200*)
where
$(val\ (x := sX)\text{-}xs) \equiv$
$\lambda\ ys\ y.\ (if\ ys = xs \wedge y = x\ then\ sX\ else\ val\ ys\ y)$

definition *swapVal* ::
$'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow ('varSort,'var,'sTerm)val \Rightarrow$
$('varSort,'var,'sTerm)val$
where
$swapVal\ zs\ z1\ z2\ val \equiv \lambda xs\ x.\ val\ xs\ (x\ @xs[z1 \wedge z2]\text{-}zs)$

abbreviation *swapVal-abbrev* (- ^[- ∧ -]'-- *200*) **where**
$val\ \hat{}[z1 \wedge z2]\text{-}zs \equiv swapVal\ zs\ z1\ z2\ val$

definition *sWlsVal* **where**
$sWlsVal\ SEM\ val \equiv$
$\forall\ ys\ y.\ sWls\ SEM\ (asSort\ ys)\ (val\ ys\ y)$

definition *sWlsValNE* ::
$('index,'bindex,'varSort,'sort,'opSym,'sTerm)semDom \Rightarrow 'var \Rightarrow bool$
where
$sWlsValNE\ SEM\ x \equiv \exists\ (val :: ('varSort,'var,'sTerm)val).\ sWlsVal\ SEM\ val$

9.1.2 Basic facts

lemma *sWlsNE-imp-sWlsAbsNE*:
assumes *sWlsNE SEM*
shows *sWlsAbsNE SEM*
⟨*proof*⟩

lemma *sWlsDisj-imp-sWlsAbsDisj*:
$sWlsDisj\ SEM \Longrightarrow sWlsNE\ SEM \Longrightarrow sWlsAbsDisj\ SEM$
⟨*proof*⟩

lemma *sWlsNE-imp-sWlsValNE*:
$sWlsNE\ SEM \Longrightarrow sWlsValNE\ SEM\ x$
⟨*proof*⟩

theorem *updVal-simp*[*simp*]:
$(val\ (x := sX)\text{-}xs)\ ys\ y = (if\ ys = xs \wedge y = x\ then\ sX\ else\ val\ ys\ y)$
⟨*proof*⟩

theorem *updVal-over*[*simp*]:
$((val\ (x := sX)\text{-}xs)\ (x := sX')\text{-}xs) = (val\ (x := sX')\text{-}xs)$
⟨*proof*⟩

theorem *updVal-commute*:

assumes $xs \neq ys \vee x \neq y$
shows $((val\ (x := sX)\text{-}xs)\ (y := sY)\text{-}ys) = ((val\ (y := sY)\text{-}ys)\ (x := sX)\text{-}xs)$
$\langle proof \rangle$

theorem *updVal-preserves-sWls*[*simp*]:
assumes *sWls SEM* (*asSort xs*) *sX* **and** *sWlsVal SEM val*
shows *sWlsVal SEM* $(val\ (x := sX)\text{-}xs)$
$\langle proof \rangle$

lemmas *updVal-simps* = *updVal-simp updVal-over updVal-preserves-sWls*

theorem *swapVal-ident*[*simp*]: $(val\ \hat{}[x \wedge x]\text{-}xs) = val$
$\langle proof \rangle$

theorem *swapVal-compose*:
$((val\ \hat{}[x \wedge y]\text{-}zs)\ \hat{}[x' \wedge y']\text{-}zs') =$
$((val\ \hat{}[x'\ @zs'[x \wedge y]\text{-}zs \wedge y'\ @zs'[x \wedge y]\text{-}zs]\text{-}zs')\ \hat{}[x \wedge y]\text{-}zs)$
$\langle proof \rangle$

theorem *swapVal-commute*:
$zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\} \Longrightarrow$
$((val\ \hat{}[x \wedge y]\text{-}zs)\ \hat{}[x' \wedge y']\text{-}zs') = ((val\ \hat{}[x' \wedge y']\text{-}zs')\ \hat{}[x \wedge y]\text{-}zs)$
$\langle proof \rangle$

lemma *swapVal-involutive*[*simp*]: $((val\ \hat{}[x \wedge y]\text{-}zs)\ \hat{}[x \wedge y]\text{-}zs) = val$
$\langle proof \rangle$

lemma *swapVal-sym*: $(val\ \hat{}[x \wedge y]\text{-}zs) = (val\ \hat{}[y \wedge x]\text{-}zs)$
$\langle proof \rangle$

lemma *swapVal-preserves-sWls1*:
assumes *sWlsVal SEM val*
shows *sWlsVal SEM* $(val\ \hat{}[z1 \wedge z2]\text{-}zs)$
$\langle proof \rangle$

theorem *swapVal-preserves-sWls*[*simp*]:
sWlsVal SEM $(val\ \hat{}[z1 \wedge z2]\text{-}zs)$ = *sWlsVal SEM val*
$\langle proof \rangle$

lemmas *swapVal-simps* = *swapVal-ident swapVal-involutive swapVal-preserves-sWls*

lemma *updVal-swapVal*:
$((val\ (x := sX)\text{-}xs)\ \hat{}[y1 \wedge y2]\text{-}ys) =$
$((val\ \hat{}[y1 \wedge y2]\text{-}ys)\ ((x\ @xs[y1 \wedge y2]\text{-}ys) := sX)\text{-}xs)$
$\langle proof \rangle$

lemma *updVal-preserves-eqBut*:
assumes *eqBut val val' ys y*
shows *eqBut* $(val\ (x := sX)\text{-}xs)\ (val'\ (x := sX)\text{-}xs)$ *ys y*

⟨*proof*⟩

lemma *updVal-eqBut-eq*:
assumes *eqBut val val′ ys y*
shows $(val\ (y := sY)\text{-}ys) = (val'\ (y := sY)\text{-}ys)$
⟨*proof*⟩

lemma *swapVal-preserves-eqBut*:
assumes *eqBut val val′ xs x*
shows *eqBut* (*val* ^[*z1* ∧ *z2*]-*zs*) (*val′* ^[*z1* ∧ *z2*]-*zs*) *xs* (*x* @*xs*[*z1* ∧ *z2*]-*zs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

[9]We use the word “model" as introduced in the theory “Models-and-Recursion".

“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* ⇒ *'sTerm*

type-synonym *('varSort,'var,'sTerm) gAbs* = *('varSort,'var,'sTerm)val* ⇒ *('varSort,'sTerm)sAbs*

context *FixSyn*
begin

definition *asIMOD* ::
('index,'bindex,'varSort,'sort,'opSym,'sTerm)semDom ⇒
('index,'bindex,'varSort,'sort,'opSym,'var,
('varSort,'var,'sTerm)gTerm,
('varSort,'var,'sTerm)gAbs)model
where
asIMOD SEM ≡
⦇*igWls* = *λs 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 = *λys y. λval. if sWlsVal SEM val then val ys y else undefined,*
igAbs =
λxs x X. λval. if sWlsVal SEM val
then sAbs xs (λsX. 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 = *λdelta inp binp. λval.*
if sWlsVal SEM val then sOp SEM delta (lift (λX. X val) inp)
(lift (λA. A val) binp)
else undefined,

[10] Recall that “generalized items" inhabit models.

igFresh =
λys y X. ∀ *val val′*. *sWlsVal SEM val* ∧ *sWlsVal SEM val′* ∧ *eqBut val val′ ys y*
⟶ *X val* = *X val′*,
igFreshAbs =
λys y A. ∀ *val val′*. *sWlsVal SEM val* ∧ *sWlsVal SEM val′* ∧ *eqBut val val′ ys y*
⟶ *A val* = *A val′*,
igSwap = *λzs z1 z2 X*. *λval*. *if sWlsVal SEM val then X* (*val* ^[*z1* ∧ *z2*]-*zs*)
else undefined,
igSwapAbs = *λzs z1 z2 A*. *λval*. *if sWlsVal SEM val then A* (*val* ^[*z1* ∧ *z2*]-*zs*)
else undefined,
igSubst = *λys Y y X*. *λval*. *if sWlsVal SEM val then X* (*val* (*y* := *Y val*)-*ys*)
else undefined,
igSubstAbs = *λys Y y A*. *λval*. *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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *asIMOD-igOp*:
igOp (*asIMOD SEM*) *delta inp binp* =
(*λval*. *if sWlsVal SEM val then sOp SEM delta* (*lift* (*λX*. *X val*) *inp*)
(*lift* (*λA*. *A val*) *binp*)
else undefined)
⟨*proof*⟩

lemma *asIMOD-igVar*:
igVar (*asIMOD SEM*) *ys y* = (*λval*. *if sWlsVal SEM val then val ys y else undefined*)
⟨*proof*⟩

lemma *asIMOD-igAbs*:
igAbs (*asIMOD SEM*) *xs x X* =
(*λval*. *if sWlsVal SEM val then sAbs xs* (*λsX*. *if sWls SEM* (*asSort xs*) *sX*
then X (*val* (*x* := *sX*)-*xs*)
else sDummy SEM (*SOME s*. *sWls SEM s*
(*X val*)))
else undefined)

⟨*proof*⟩

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*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *asIMOD-igSwap*:
igSwap (*asIMOD SEM*) *zs z1 z2 X* =
(λ*val*. *if sWlsVal SEM val then X* (*val* ^[*z1* ∧ *z2*]-*zs*) *else undefined*)
⟨*proof*⟩

lemma *asIMOD-igSwapAbs*:
igSwapAbs (*asIMOD SEM*) *zs z1 z2 A* =
(λ*val*. *if sWlsVal SEM val then A* (*val* ^[*z1* ∧ *z2*]-*zs*) *else undefined*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *asIMOD-igSubstInp*:
sWlsVal SEM val ⟹
lift (λ*X. X val*) (*igSubstInp* (*asIMOD SEM*) *ys Y y inp*) =
lift (λ*X. X* (*val* (*y* := *Y val*)-*ys*)) *inp*
⟨*proof*⟩

lemma *asIMOD-igWlsBinp*:
assumes *sWlsNE SEM*
shows
igWlsBinp (*asIMOD SEM*) *delta binp* =
((∀ *val. liftAll* (λ*X. sWlsVal SEM val* ∨ *X val* = *undefined*) *binp*) ∧
(∀ *val. sWlsVal SEM val* ⟶ *sWlsBinp SEM delta* (*lift* (λ*X. X val*) *binp*)))
⟨*proof*⟩

lemma *asIMOD-igSwapBinp*:
sWlsVal SEM val ⟹
lift (λ*A. A val*) (*igSwapBinp* (*asIMOD SEM*) *zs z1 z2 binp*) =
lift (λ*A. A* (*swapVal zs z1 z2 val*)) *binp*
⟨*proof*⟩

lemma *asIMOD-igSubstBinp*:
sWlsVal SEM val ⟹
lift (λ*A. A val*) (*igSubstBinp* (*asIMOD SEM*) *ys Y y binp*) =
lift (λ*A. A* (*val* (*y* := *Y val*)-*ys*)) *binp*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *asIMOD-igWlsAbsDisj*:
sWlsNE SEM ⟹ *sWlsDisj SEM* ⟹ *igWlsAbsDisj* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igWlsAllDisj*:
sWlsNE SEM ⟹ *sWlsDisj SEM* ⟹ *igWlsAllDisj* (*asIMOD SEM*)
⟨*proof*⟩

Only "bound arit" abstraction domains are inhabited:

lemma *asIMOD-igWlsAbsIsInBar*:
sWlsNE SEM ⟹ *igWlsAbsIsInBar* (*asIMOD SEM*)
⟨*proof*⟩

Domain preservation by the operators

The constructs preserve the domains:

lemma *asIMOD-igVarIPresIGWls*: *igVarIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igAbsIPresIGWls*:
sWlsDisj SEM ⟹ *igAbsIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igOpIPresIGWls*:
sOpPrSWls SEM ⟹ *sWlsNE SEM* ⟹ *igOpIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igConsIPresIGWls*:
wlsSEM SEM ⟹ *igConsIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

Swap preserves the domains:

lemma *asIMOD-igSwapIPresIGWls*: *igSwapIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSwapAbsIPresIGWlsAbs*: *igSwapAbsIPresIGWlsAbs* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSwapAllIPresIGWlsAll*: *igSwapAllIPresIGWlsAll* (*asIMOD SEM*)
⟨*proof*⟩

Subst preserves the domains:

lemma *asIMOD-igSubstIPresIGWls*: *igSubstIPresIGWls* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstAbsIPresIGWlsAbs*: *igSubstAbsIPresIGWlsAbs* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstAllIPresIGWlsAll*: *igSubstAllIPresIGWlsAll* (*asIMOD SEM*)
⟨*proof*⟩

The clauses for fresh hold:

lemma *asIMOD-igFreshIGVar*: *igFreshIGVar* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igFreshIGAbs1*:
sWlsDisj SEM ⟹ *igFreshIGAbs1* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igFreshIGAbs2*:
sWlsDisj SEM ⟹ *igFreshIGAbs2* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igFreshIGOp*:
fixes *SEM* :: (*'index*,*'bindex*,*'varSort*,*'sort*,*'opSym*,*'sTerm*)*semDom*
shows *igFreshIGOp* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igFreshCls*:
assumes *sWlsDisj SEM*
shows *igFreshCls* (*asIMOD SEM*)
⟨*proof*⟩

The clauses for swap hold:

lemma *asIMOD-igSwapIGVar*: *igSwapIGVar* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSwapIGAbs*: *igSwapIGAbs* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSwapIGOp*: *igSwapIGOp* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSwapCls*: *igSwapCls* (*asIMOD SEM*)
⟨*proof*⟩

The clauses for subst hold:

lemma *asIMOD-igSubstIGVar1*: *igSubstIGVar1* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstIGVar2*: *igSubstIGVar2* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstIGAbs*: *igSubstIGAbs* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstIGOp*: *igSubstIGOp* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-igSubstCls*: *igSubstCls* (*asIMOD SEM*)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *asIMOD-igAbsCongS*: *sWlsDisj SEM* $\Longrightarrow$ *igAbsCongS* (*asIMOD SEM*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *asIMOD-igAbsRen*:
sWlsDisj SEM $\Longrightarrow$ *igAbsRen* (*asIMOD SEM*)
⟨*proof*⟩

The associated model forms well-structured models of all 4 kinds:

lemma *asIMOD-wlsFSw*:
assumes *wlsSEM SEM*
shows *iwlsFSw* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-wlsFSb*:
assumes *wlsSEM SEM*
shows *iwlsFSb* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-wlsFSwSb*: *wlsSEM SEM* $\Longrightarrow$ *iwlsFSwSb* (*asIMOD SEM*)
⟨*proof*⟩

lemma *asIMOD-wlsFSbSw*: *wlsSEM SEM* $\Longrightarrow$ *iwlsFSbSw* (*asIMOD SEM*)
⟨*proof*⟩

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* ≡ *iter* (*asIMOD SEM*)

definition *semIntAbs* **where** *semIntAbs SEM* ≡ *iterAbs* (*asIMOD SEM*)

lemma *semIntAll-termFSwSbImorph*:
wlsSEM SEM ⟹
termFSwSbImorph (*semInt SEM*) (*semIntAbs SEM*) (*asIMOD SEM*)
⟨*proof*⟩

lemma *semInt-prWls*:
wlsSEM SEM ⟹ *prWls* (*semInt SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAbs-prWlsAbs*:
wlsSEM SEM ⟹ *prWlsAbs* (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prWlsAll*:
wlsSEM SEM ⟹ *prWlsAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semInt-prVar*:
wlsSEM SEM ⟹ *prVar* (*semInt SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prAbs*:
fixes *SEM* :: (*'index,'bindex,'varSort,'sort,'opSym,'sTerm*)*semDom*
assumes *wlsSEM SEM*
shows *prAbs* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prOp*:
assumes *wlsSEM SEM*
shows *prOp* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prCons*:
assumes *wlsSEM SEM*

shows *prCons* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semInt-prFresh*:
assumes *wlsSEM SEM*
shows *prFresh* (*semInt SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAbs-prFreshAbs*:
assumes *wlsSEM SEM*
shows *prFreshAbs* (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prFreshAll*:
assumes *wlsSEM SEM*
shows *prFreshAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semInt-prSwap*:
assumes *wlsSEM SEM*
shows *prSwap* (*semInt SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAbs-prSwapAbs*:
assumes *wlsSEM SEM*
shows *prSwapAbs* (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prSwapAll*:
assumes *wlsSEM SEM*
shows *prSwapAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semInt-prSubst*:
assumes *wlsSEM SEM*
shows *prSubst* (*semInt SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAbs-prSubstAbs*:
assumes *wlsSEM SEM*
shows *prSubstAbs* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

lemma *semIntAll-prSubstAll*:
assumes *wlsSEM SEM*
shows *prSubstAll* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

theorem *semIntAll-compInt*:

assumes *wlsSEM SEM*
shows *compInt* (*semInt SEM*) (*semIntAbs SEM*) *SEM*
⟨*proof*⟩

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*

$gVar$ :: $'varSort \Rightarrow 'var \Rightarrow 'gTerm$
$gAbs$ ::
$'varSort \Rightarrow 'var \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow$
$'gAbs$
gOp ::
$'opSym \Rightarrow$
$('index,('index,'bindex,'varSort,'var,'opSym)term)input \Rightarrow ('index,'gTerm)input$
$\Rightarrow$
$('bindex,('index,'bindex,'varSort,'var,'opSym)abs)input \Rightarrow ('bindex,'gAbs)input$
$\Rightarrow$
$'gTerm$

$gFresh$ ::
$'varSort \Rightarrow 'var \Rightarrow ('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow bool$
$gFreshAbs$ ::
$'varSort \Rightarrow 'var \Rightarrow ('index,'bindex,'varSort,'var,'opSym)abs \Rightarrow 'gAbs \Rightarrow bool$

$gSwap$ ::
$'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow$
$'gTerm$
$gSwapAbs$ ::
$'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)abs \Rightarrow 'gAbs \Rightarrow$
$'gAbs$

$gSubst$ ::
$'varSort \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow$
$'var \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow$
$'gTerm$
$gSubstAbs$ ::
$'varSort \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)term \Rightarrow 'gTerm \Rightarrow$
$'var \Rightarrow$
$('index,'bindex,'varSort,'var,'opSym)abs \Rightarrow 'gAbs \Rightarrow$
$'gAbs$

10.2 Well-sorted models of various kinds

Lifting the model operations to inputs

definition *gFreshInp* **where**
$gFreshInp\ MOD\ ys\ y\ inp'\ inp \equiv liftAll2\ (gFresh\ MOD\ ys\ y)\ inp'\ inp$

definition *gFreshBinp* **where**
$gFreshBinp\ MOD\ ys\ y\ binp'\ binp \equiv liftAll2\ (gFreshAbs\ MOD\ ys\ y)\ binp'\ binp$

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

. 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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ∧ *gOpPres-GWls 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 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

$(inp'\ \%[z1 \wedge z2]\text{-}zs)$ $(gSwapInp\ MOD\ zs\ z1\ z2\ inp'\ inp)$
$(binp'\ \%\%[z1 \wedge z2]\text{-}zs)$ $(gSwapBinp\ MOD\ zs\ z1\ z2\ binp'\ binp)$

definition *gSwapCls* **where**
$gSwapCls\ MOD \equiv gSwapGVar\ MOD \wedge gSwapGAbs\ MOD \wedge gSwapGOp\ MOD$

lemmas *gSwapCls-defs = gSwapCls-def*
gSwapGVar-def gSwapGAbs-def gSwapGOp-def

definition *gSubstGVar1* **where**
$gSubstGVar1\ MOD \equiv \forall\ ys\ y\ Y'\ Y\ xs\ x.$
$wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$
$(ys \neq xs \vee y \neq x) \longrightarrow$
$gSubst\ MOD\ ys\ Y'\ Y\ y\ (Var\ xs\ x)\ (gVar\ MOD\ xs\ x) =$
$gVar\ MOD\ xs\ x$

definition *gSubstGVar2* **where**
$gSubstGVar2\ MOD \equiv \forall\ ys\ y\ Y'\ Y.$
$wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$
$gSubst\ MOD\ ys\ Y'\ Y\ y\ (Var\ ys\ y)\ (gVar\ MOD\ ys\ y) = Y$

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 ≡ ∀ *xs x x2 y s X′ X X2′ X2*.
isInBar (*xs*,*s*) ∧
wls s X′ ∧ *gWls MOD s X* ∧
wls s X2′ ∧ *gWls MOD s X2* ⟶
fresh xs y X′ ∧ *gFresh MOD xs y X′ X* ∧
fresh xs y X2′ ∧ *gFresh MOD xs y X2′ X2* ∧
(*X′* #[*y* ∧ *x*]-*xs*) = (*X2′* #[*y* ∧ *x2*]-*xs*) ⟶
gSwap MOD xs y x X′ X = *gSwap MOD xs y x2 X2′ X2* ⟶
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*
⟨*proof*⟩

lemma *imp-gSubstInpPresGWlsInp*:
gSubstPresGWls MOD $\Longrightarrow$ *gSubstInpPresGWlsInp MOD*
⟨*proof*⟩

Then for bound inputs:

definition *gSwapBinpPresGWlsBinp* **where**
gSwapBinpPresGWlsBinp MOD $\equiv$ $\forall$ *zs z1 z2 delta binp′ binp.*
wlsBinp delta binp′ $\wedge$ *gWlsBinp MOD delta binp* $\longrightarrow$
gWlsBinp MOD delta (*gSwapBinp MOD zs z1 z2 binp′ binp*)

definition *gSubstBinpPresGWlsBinp* **where**
gSubstBinpPresGWlsBinp MOD $\equiv$ $\forall$ *ys y Y′ Y delta binp′ binp.*
wls (*asSort ys*) *Y′* $\wedge$ *gWls MOD* (*asSort ys*) *Y* $\wedge$
wlsBinp delta binp′ $\wedge$ *gWlsBinp MOD delta binp* $\longrightarrow$
gWlsBinp MOD delta (*gSubstBinp MOD ys Y′ Y y binp′ binp*)

lemma *imp-gSwapBinpPresGWlsBinp*:
gSwapAbsPresGWlsAbs MOD $\Longrightarrow$ *gSwapBinpPresGWlsBinp MOD*
⟨*proof*⟩

lemma *imp-gSubstBinpPresGWlsBinp*:
gSubstAbsPresGWlsAbs MOD $\Longrightarrow$ *gSubstBinpPresGWlsBinp MOD*
⟨*proof*⟩

10.3 Model morphisms from the term model

definition *presWls* **where**
presWls h MOD $\equiv$ $\forall$ *s X. wls s X* $\longrightarrow$ *gWls MOD s* (*h X*)

definition *presWlsAbs* **where**
presWlsAbs hA MOD $\equiv$ $\forall$ *us s A. wlsAbs* (*us,s*) *A* $\longrightarrow$ *gWlsAbs MOD* (*us,s*) (*hA A*)

definition *presWlsAll* **where**
presWlsAll h hA MOD $\equiv$ *presWls h MOD* $\wedge$ *presWlsAbs hA MOD*

lemmas *presWlsAll-defs* = *presWlsAll-def presWls-def presWlsAbs-def*

definition *presVar* **where**
presVar h MOD $\equiv$ $\forall$ *xs x. h* (*Var xs x*) = *gVar MOD xs x*

definition *presAbs* **where**
presAbs h hA MOD $\equiv$ $\forall$ *xs x s X.*
isInBar (*xs,s*) $\wedge$ *wls s X* $\longrightarrow$
hA (*Abs xs x X*) = *gAbs MOD xs x X* (*h X*)

definition *presOp* **where**
presOp h hA MOD $\equiv$ $\forall$ *delta inp binp.*
wlsInp delta inp $\wedge$ *wlsBinp delta binp* $\longrightarrow$

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*)
⟨*proof*⟩

. for bound inputs:

lemma *presWls-wlsBinp*:
wlsBinp delta binp ⟹ *presWlsAbs hA MOD* ⟹ *gWlsBinp MOD delta* (*lift hA binp*)

⟨proof⟩

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 × 'gTerm,
('index,'bindex,'varSort,'var,'opSym)abs × 'gAbs) Iteration.model
where
fromMOD MOD ≡
(|
igWls = λs X'X. wls s (fst X'X) ∧ gWls MOD s (snd X'X),
igWlsAbs = λus-s A'A. wlsAbs us-s (fst A'A) ∧ gWlsAbs MOD us-s (snd A'A),

igVar = λxs x. (Var xs x, gVar MOD xs x),
igAbs = λxs x X'X. (Abs xs x (fst X'X), gAbs MOD xs x (fst X'X) (snd X'X)),
igOp =
λdelta 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 =
λys y X'X. fresh ys y (fst X'X) ∧ gFresh MOD ys y (fst X'X) (snd X'X),
igFreshAbs =
λys y A'A. freshAbs ys y (fst A'A) ∧ gFreshAbs MOD ys y (fst A'A) (snd A'A),

igSwap =
λzs z1 z2 X'X. ((fst X'X) #[z1 ∧ z2]-zs, gSwap MOD zs z1 z2 (fst X'X) (snd X'X)),
igSwapAbs =
λzs z1 z2 A'A. ((fst A'A) $[z1 ∧ z2]-zs, gSwapAbs MOD zs z1 z2 (fst A'A) (snd A'A)),

igSubst =
λys 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 =
λys 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))
|)

Basic simplification rules:

lemma *fromMOD-basic-simps*[*simp*]:
igWls (*fromMOD MOD*) *s X′X* =
(*wls s* (*fst X′X*) ∧ *gWls MOD s* (*snd X′X*))

igWlsAbs (*fromMOD MOD*) *us-s A′A* =
(*wlsAbs us-s* (*fst A′A*) ∧ *gWlsAbs MOD us-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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemmas *binput-fromMOD-simps* =
igWlsBinp-fromMOD igFreshBinp-fromMOD igSwapBinp-fromMOD igSubstBinp-fromMOD

Domain disjointness:

lemma *igWlsDisj-fromMOD*[*simp*]:
gWlsDisj MOD $\Longrightarrow$ *igWlsDisj* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igWlsAbsDisj-fromMOD*[*simp*]:
gWlsAbsDisj MOD $\Longrightarrow$ *igWlsAbsDisj* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igWlsAllDisj-fromMOD*[*simp*]:
gWlsAllDisj MOD $\Longrightarrow$ *igWlsAllDisj* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igWlsAllDisj-fromMOD-simps* =
igWlsDisj-fromMOD igWlsAbsDisj-fromMOD igWlsAllDisj-fromMOD

Abstractions only within IsInBar:

lemma *igWlsAbsIsInBar-fromMOD*[*simp*]:
gWlsAbsIsInBar MOD $\Longrightarrow$ *igWlsAbsIsInBar* (*fromMOD MOD*)
⟨*proof*⟩

The constructs preserve the domains:

lemma *igVarIPresIGWls-fromMOD*[*simp*]:
gVarPresGWls MOD $\Longrightarrow$ *igVarIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igAbsIPresIGWls-fromMOD*[*simp*]:
gAbsPresGWls MOD $\Longrightarrow$ *igAbsIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igOpIPresIGWls-fromMOD*[*simp*]:
gOpPresGWls MOD $\Longrightarrow$ *igOpIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igConsIPresIGWls-fromMOD*[*simp*]:
gConsPresGWls MOD $\Longrightarrow$ *igConsIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igConsIPresIGWls-fromMOD-simps* =
igVarIPresIGWls-fromMOD igAbsIPresIGWls-fromMOD
igOpIPresIGWls-fromMOD igConsIPresIGWls-fromMOD

Swap preserves the domains:

lemma *igSwapIPresIGWls-fromMOD*[*simp*]:
gSwapPresGWls MOD $\Longrightarrow$ *igSwapIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSwapAbsIPresIGWlsAbs-fromMOD*[*simp*]:
gSwapAbsPresGWlsAbs MOD $\Longrightarrow$ *igSwapAbsIPresIGWlsAbs* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSwapAllIPresIGWlsAll-fromMOD*[*simp*]:
gSwapAllPresGWlsAll MOD $\Longrightarrow$ *igSwapAllIPresIGWlsAll* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igSwapAllIPresIGWlsAll-fromMOD-simps* =
igSwapIPresIGWls-fromMOD igSwapAbsIPresIGWlsAbs-fromMOD igSwapAllIPresIG-WlsAll-fromMOD

Subst preserves the domains:

lemma *igSubstIPresIGWls-fromMOD*[*simp*]:
gSubstPresGWls MOD $\Longrightarrow$ *igSubstIPresIGWls* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstAbsIPresIGWlsAbs-fromMOD*[*simp*]:
gSubstAbsPresGWlsAbs MOD $\Longrightarrow$ *igSubstAbsIPresIGWlsAbs* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstAllIPresIGWlsAll-fromMOD*[*simp*]:
gSubstAllPresGWlsAll MOD $\Longrightarrow$ *igSubstAllIPresIGWlsAll* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igSubstAllIPresIGWlsAll-fromMOD-simps* =
igSubstIPresIGWls-fromMOD igSubstAbsIPresIGWlsAbs-fromMOD igSubstAllIPresIG-WlsAll-fromMOD

The fresh clauses:

lemma *igFreshIGVar-fromMOD*[*simp*]:
gFreshGVar MOD $\Longrightarrow$ *igFreshIGVar* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igFreshIGAbs1-fromMOD*[*simp*]:
gFreshGAbs1 MOD $\Longrightarrow$ *igFreshIGAbs1* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igFreshIGAbs2-fromMOD*[*simp*]:
gFreshGAbs2 MOD $\Longrightarrow$ *igFreshIGAbs2* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igFreshIGOp-fromMOD*[*simp*]:
gFreshGOp MOD $\Longrightarrow$ *igFreshIGOp* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igFreshCls-fromMOD*[*simp*]:
gFreshCls MOD $\Longrightarrow$ *igFreshCls* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igFreshCls-fromMOD-simps* =
igFreshIGVar-fromMOD igFreshIGAbs1-fromMOD igFreshIGAbs2-fromMOD

igFreshIGOp-fromMOD igFreshCls-fromMOD

The swap clauses

lemma *igSwapIGVar-fromMOD*[*simp*]:
gSwapGVar MOD $\Longrightarrow$ *igSwapIGVar* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSwapIGAbs-fromMOD*[*simp*]:
gSwapGAbs MOD $\Longrightarrow$ *igSwapIGAbs* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSwapIGOp-fromMOD*[*simp*]:
gSwapGOp MOD $\Longrightarrow$ *igSwapIGOp* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSwapCls-fromMOD*[*simp*]:
gSwapCls MOD $\Longrightarrow$ *igSwapCls* (*fromMOD MOD*)
⟨*proof*⟩

lemmas *igSwapCls-fromMOD-simps* =
igSwapIGVar-fromMOD igSwapIGAbs-fromMOD
igSwapIGOp-fromMOD igSwapCls-fromMOD

The subst clauses

lemma *igSubstIGVar1-fromMOD*[*simp*]:
gSubstGVar1 MOD $\Longrightarrow$ *igSubstIGVar1* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstIGVar2-fromMOD*[*simp*]:
gSubstGVar2 MOD $\Longrightarrow$ *igSubstIGVar2* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstIGAbs-fromMOD*[*simp*]:
gSubstGAbs MOD $\Longrightarrow$ *igSubstIGAbs* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstIGOp-fromMOD*[*simp*]:
gSubstGOp MOD $\Longrightarrow$ *igSubstIGOp* (*fromMOD MOD*)
⟨*proof*⟩

lemma *igSubstCls-fromMOD*[*simp*]:
gSubstCls MOD $\Longrightarrow$ *igSubstCls* (*fromMOD MOD*)
⟨*proof*⟩

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*)
⟨*proof*⟩

Abstraction renaming:

lemma *igAbsRen-fromMOD*[*simp*]:
gAbsRen MOD ⟹ *igAbsRen* (*fromMOD MOD*)
⟨*proof*⟩

Models:

lemma *iwlsFSw-fromMOD*[*simp*]:
wlsFSw MOD ⟹ *iwlsFSw* (*fromMOD MOD*)
⟨*proof*⟩

lemma *iwlsFSb-fromMOD*[*simp*]:
wlsFSb MOD ⟹ *iwlsFSb* (*fromMOD MOD*)
⟨*proof*⟩

lemma *iwlsFSwSb-fromMOD*[*simp*]:
wlsFSwSb MOD ⟹ *iwlsFSwSb* (*fromMOD MOD*)
⟨*proof*⟩

lemma *iwlsFSbSw-fromMOD*[*simp*]:
wlsFSbSw MOD ⟹ *iwlsFSbSw* (*fromMOD MOD*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *ipresCons-fromMOD-fst-inp*[*simp*]:
ipresCons h hA (*fromMOD MOD*) ⟹ *wlsInp delta inp* ⟹ *lift* (*fst o h*) *inp* = *inp*
⟨*proof*⟩

lemma *ipresCons-fromMOD-fst-binp*[*simp*]:
ipresCons h hA (*fromMOD MOD*) ⟹ *wlsBinp delta binp* ⟹ *lift* (*fst o hA*) *binp* = *binp*
⟨*proof*⟩

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′)*
⟨*proof*⟩

lemma *fromIMorAbs*[*simp*]: *fromIMorAbs hA A′ = snd (hA A′)*
⟨*proof*⟩

lemma *fromIMor-snd-inp*[*simp*]:
wlsInp delta inp ⟹ lift (fromIMor h) inp = lift (snd o h) inp
⟨*proof*⟩

lemma *fromIMorAbs-snd-binp*[*simp*]:
wlsBinp delta binp ⟹ lift (fromIMorAbs hA) binp = lift (snd o hA) binp
⟨*proof*⟩

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) ⟹ presWls (fromIMor h) MOD
⟨*proof*⟩

lemma *presWlsAbs-fromIMorAbs*[*simp*]:
ipresWlsAbs hA (fromMOD MOD) ⟹ presWlsAbs (fromIMorAbs hA) MOD
⟨*proof*⟩

lemma *presWlsAll-fromIMorAll*[*simp*]:
ipresWlsAll h hA (fromMOD MOD) ⟹ presWlsAll (fromIMor h) (fromIMorAbs hA) MOD
⟨*proof*⟩

lemmas *presWlsAll-fromIMorAll-simps =*
presWls-fromIMor presWlsAbs-fromIMorAbs presWlsAll-fromIMorAll

Preservation of the constructs

lemma *presVar-fromIMor*[*simp*]:
ipresCons h hA (fromMOD MOD) ⟹ presVar (fromIMor h) MOD
⟨*proof*⟩

lemma *presAbs-fromIMor*[*simp*]:
assumes *ipresCons h hA (fromMOD MOD)*
shows *presAbs (fromIMor h) (fromIMorAbs hA) MOD*
⟨*proof*⟩

lemma *presOp-fromIMor*[*simp*]:
assumes *ipresCons h hA (fromMOD MOD)*
shows *presOp (fromIMor h) (fromIMorAbs hA) MOD*

⟨*proof*⟩

lemma *presCons-fromIMor*[*simp*]:
assumes *ipresCons h hA* (*fromMOD MOD*)
shows *presCons* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *presFreshAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresFreshAbs hA* (*fromMOD MOD*)
$\Longrightarrow$ *presFreshAbs* (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

lemma *presFreshAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresFreshAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presFreshAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *presSwapAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSwapAbs hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSwapAbs* (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

lemma *presSwapAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSwapAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSwapAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *presSubstAbs-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSubstAbs h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSubstAbs* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

lemma *presSubstAll-fromIMor*[*simp*]:
ipresCons h hA (*fromMOD MOD*) $\Longrightarrow$ *ipresSubstAll h hA* (*fromMOD MOD*)
$\Longrightarrow$ *presSubstAll* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fromIMor-termFSbMorph*[*simp*]:
termFSbImorph h hA (*fromMOD MOD*) $\Longrightarrow$ *termFSbMorph* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

lemma *fromIMor-termFSwSbMorph*[*simp*]:
assumes *termFSwSbImorph h hA* (*fromMOD MOD*)
shows *termFSwSbMorph* (*fromIMor h*) (*fromIMorAbs hA*) *MOD*
⟨*proof*⟩

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* ≡ *fromIMor* (*iter* (*fromMOD MOD*))

definition *recAbs* **where** *recAbs MOD* ≡ *fromIMorAbs* (*iterAbs* (*fromMOD MOD*))

Existence:

theorem *wlsFSw-recAll-termFSwMorph*:
wlsFSw MOD ⟹ *termFSwMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

theorem *wlsFSb-recAll-termFSbMorph*:
wlsFSb MOD ⟹ *termFSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

theorem *wlsFSwSb-recAll-termFSwSbMorph*:
wlsFSwSb MOD ⟹ *termFSwSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

theorem *wlsFSbSw-recAll-termFSwSbMorph*:
wlsFSbSw MOD ⟹ *termFSwSbMorph* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

Uniqueness:

lemma *presCons-unique*:
assumes *presCons f fA MOD* **and** *presCons g gA MOD*
shows (*wls s X* ⟶ *f X* = *g X*) ∧ (*wlsAbs* (*us*,*s*′) *A* ⟶ *fA A* = *gA A*)
⟨*proof*⟩

theorem *wlsFSw-recAll-unique-presCons*:
assumes *wlsFSw MOD* **and** *presCons h hA MOD*
shows (*wls s X* ⟶ *h X* = *rec MOD X*) ∧
(*wlsAbs* (*us*,*s*′) *A* ⟶ *hA A* = *recAbs MOD A*)
⟨*proof*⟩

theorem *wlsFSb-recAll-unique-presCons*:
assumes *wlsFSb MOD* **and** *presCons h hA MOD*
shows (*wls s X* ⟶ *h X* = *rec MOD X*) ∧
(*wlsAbs* (*us*,*s*′) *A* ⟶ *hA A* = *recAbs MOD A*)
⟨*proof*⟩

theorem *wlsFSwSb-recAll-unique-presCons*:
assumes *wlsFSwSb MOD* **and** *presCons h hA MOD*
shows (*wls s X* ⟶ *h X* = *rec MOD X*) ∧
(*wlsAbs* (*us*,*s*′) *A* ⟶ *hA A* = *recAbs MOD A*)
⟨*proof*⟩

theorem *wlsFSbSw-recAll-unique-presCons*:

assumes *wlsFSbSw MOD* **and** *presCons h hA MOD*
shows $(wls\ s\ X \longrightarrow h\ X = rec\ MOD\ X) \wedge$
$(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = recAbs\ MOD\ A)$
$\langle proof \rangle$

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**
$gFreshGVarRev\ MOD \equiv \forall\ xs\ y\ x.$
$gFresh\ MOD\ xs\ y\ (Var\ xs\ x)\ (gVar\ MOD\ xs\ x) \longrightarrow y \neq x$

definition *gFreshGAbsRev* **where**
$gFreshGAbsRev\ MOD \equiv \forall\ ys\ y\ xs\ x\ s\ X'\ X.$
$isInBar\ (xs,s) \wedge wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$
$gFreshAbs\ MOD\ ys\ y\ (Abs\ xs\ x\ X')\ (gAbs\ MOD\ xs\ x\ X'\ X) \longrightarrow$
$(ys = xs \wedge y = x) \vee gFresh\ MOD\ ys\ y\ X'\ X$

definition *gFreshGOpRev* **where**
$gFreshGOpRev\ MOD \equiv \forall\ ys\ y\ delta\ inp'\ inp\ binp'\ binp.$
$wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \wedge wlsBinp\ delta\ binp' \wedge gWlsBinp\ MOD\ delta\ binp \longrightarrow$
$gFresh\ MOD\ ys\ y\ (Op\ delta\ inp'\ binp')\ (gOp\ MOD\ delta\ inp'\ inp\ binp'\ binp) \longrightarrow$
$gFreshInp\ MOD\ ys\ y\ inp'\ inp \wedge gFreshBinp\ MOD\ ys\ y\ binp'\ binp$

definition *gFreshClsRev* **where**
$gFreshClsRev\ MOD \equiv gFreshGVarRev\ MOD \wedge gFreshGAbsRev\ MOD \wedge gFreshGOpRev\ MOD$

lemmas *gFreshClsRev-defs = gFreshClsRev-def gFreshGVarRev-def gFreshGAbsRev-def gFreshGOpRev-def*

Injectiveness of the construct operators

definition *gVarInj* **where**
$gVarInj\ MOD \equiv \forall\ xs\ x\ y.\ gVar\ MOD\ xs\ x = gVar\ MOD\ xs\ y \longrightarrow x = y$

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*⟧

$\Longrightarrow$ *phi* (*stOf delta*) (*gOp MOD delta inp′ inp binp′ binp*) **and**
Abs:
$\bigwedge$ *xs s x X′ X*.
⟦*isInBar* (*xs*,*s*); *wls s X′*; *gWls MOD s X*; *phi s X*⟧
$\Longrightarrow$ *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*)
⟨*proof*⟩

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* ⟶
  (∃ *A'*. *wlsAbs* (*us*,*s*) *A'* ∧ *hA A'* = *A*)

definition *isSurjAll* **where**
isSurjAll h hA MOD ≡ *isSurj h MOD* ∧ *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*⟩

For fresh-swap models

theorem *wlsFSw-recAll-reflFreshAll*:
wlsFSw MOD ⟹ *gFreshClsRev MOD* ⟹ *reflFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

For fresh-subst models

theorem *wlsFSb-recAll-reflFreshAll*:
wlsFSb MOD ⟹ *gFreshClsRev MOD* ⟹ *reflFreshAll* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

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*⟩

For fresh-subst models

theorem *wlsFSb-recAll-isInjAll*:

assumes ∗: *wlsFSb MOD* **and** ∗∗: *gConsInj MOD*
shows *isInjAll* (*rec MOD*) (*recAbs MOD*)
⟨*proof*⟩

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*⟩

For fresh-swap models

theorem *wlsFSw-recAll-isSurjAll*:
wlsFSw MOD ⟹ *gConsIndif MOD* ⟹ *gInduct MOD*
⟹ *isSurjAll* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

For fresh-subst models

theorem *wlsFSb-recAll-isSurjAll*:
wlsFSb MOD ⟹ *gConsIndif MOD* ⟹ *gInduct MOD*
⟹ *isSurjAll* (*rec MOD*) (*recAbs MOD*) *MOD*
⟨*proof*⟩

lemmas *recursion-simps* =
fromMOD-simps ipresCons-fromMOD-fst-all-simps fromIMor-simps

declare *recursion-simps* [*simp del*]

end

end