

# A General Theory of Syntax with Bindings

Lorenzo Gheri and Andrei Popescu

June 17, 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

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Quasi-Terms with Swapping and Freshness</b>                 | <b>1</b>  |
| 1.1      | The datatype of quasi-terms with bindings . . . . .            | 1         |
| 1.2      | Induction principles . . . . .                                 | 2         |
| 1.3      | Swap and substitution on variables . . . . .                   | 4         |
| 1.4      | The swapping and freshness operators . . . . .                 | 6         |
| 1.5      | Compositional properties of swapping . . . . .                 | 8         |
| 1.6      | Induction and well-foundedness modulo swapping . . . . .       | 10        |
| 1.7      | More properties connecting swapping and freshness . . . . .    | 11        |
| <b>2</b> | <b>Availability of Fresh Variables and Alpha-Equivalence</b>   | <b>13</b> |
| 2.1      | The FixVars locale . . . . .                                   | 13        |
| 2.2      | Good quasi-terms . . . . .                                     | 14        |
| 2.3      | The ability to pick fresh variables . . . . .                  | 17        |
| 2.4      | Alpha-equivalence . . . . .                                    | 19        |
| 2.4.1    | Definition . . . . .   | 19        |
| 2.4.2    | Simplification and elimination rules . . . . .                 | 20        |
| 2.4.3    | Basic properties . . . . .                                     | 21        |
| 2.4.4    | Picking fresh representatives . . . . .                        | 25        |
| 2.5      | Properties of swapping and freshness modulo alpha . . . . .    | 25        |
| 2.6      | Alternative statements of the alpha-clause for bound arguments | 27        |

|          |   |            |
|----------|---|------------|
| 2.6.1    | First for “qAFresh” . . . . .   | 27         |
| 2.6.2    | Then for “qFresh” . . . . .   | 31         |
| <b>3</b> | <b>Environments and Substitution for Quasi-Terms</b>                                      | <b>35</b>  |
| 3.1      | Environments . . . . .  | 35         |
| 3.2      | Parallel substitution . . . . .   | 37         |
| <b>4</b> | <b>Some preliminaries on equivalence relations and quotients</b>                          | <b>41</b>  |
| <b>5</b> | <b>Transition from Quasi-Terms to Terms</b>   | <b>45</b>  |
| 5.1      | Preparation: Integrating quasi-inputs as first-class citizens . .                         | 45         |
| 5.2      | Definitions of terms and their operators . . . . .  | 51         |
| 5.3      | Items versus quasi-items modulo alpha . . . . .   | 58         |
| 5.3.1    | For terms . . . . .   | 58         |
| 5.3.2    | For abstractions . . . . .  | 60         |
| 5.3.3    | For inputs . . . . .  | 61         |
| 5.3.4    | For environments . . . . .  | 64         |
| 5.3.5    | The structural alpha-equivPalence maps commute with<br>the syntactic constructs . . . . . | 65         |
| 5.4      | All operators preserve the “good” predicate . . . . .                                     | 66         |
| 5.4.1    | The syntactic operators are almost constructors . . . .                                   | 70         |
| 5.5      | Properties lifted from quasi-terms to terms . . . . .                                     | 71         |
| 5.5.1    | Simplification rules . . . . .  | 71         |
| 5.5.2    | The ability to pick fresh variables . . . . .   | 75         |
| 5.5.3    | Compositionality . . . . .  | 75         |
| 5.5.4    | Compositionality for environments . . . . .   | 77         |
| 5.5.5    | Properties of the relation of being swapped . . . . .                                     | 78         |
| 5.6      | Induction . . . . .   | 79         |
| 5.6.1    | Induction lifted from quasi-terms . . . . .   | 79         |
| 5.6.2    | Fresh induction . . . . .   | 80         |
| <b>6</b> | <b>More on Terms</b>  | <b>82</b>  |
| 6.1      | Identity environment versus other operators . . . . .                                     | 83         |
| 6.2      | Environment update versus other operators . . . . .                                       | 84         |
| 6.3      | Environment “get” versus other operators . . . . .  | 85         |
| 6.4      | Substitution versus other operators . . . . .   | 87         |
| 6.5      | Properties specific to variable-for-variable substitution . . . .                         | 96         |
| 6.6      | Abstraction versions of the properties . . . . .  | 98         |
| <b>7</b> | <b>Binding Signatures and well-sorted terms</b>   | <b>104</b> |
| 7.1      | Binding signatures . . . . .  | 104        |
| 7.2      | The Binding Syntax locale . . . . .   | 104        |
| 7.3      | Definitions and basic properties of well-sortedness . . . . .                             | 106        |
| 7.3.1    | Notations and definitions . . . . .   | 106        |

|          |   |            |
|----------|---|------------|
| 7.3.2    | Sublocale of "FixVars"  | 106        |
| 7.3.3    | Abbreviations   | 106        |
| 7.3.4    | Inner versions of the locale assumptions                                | 108        |
| 7.3.5    | Definitions of well-sorted items  | 109        |
| 7.3.6    | Well-sorted exists  | 111        |
| 7.3.7    | Well-sorted implies Good  | 112        |
| 7.3.8    | Swapping preserves well-sortedness                                      | 112        |
| 7.3.9    | Inversion rules for well-sortedness                                     | 114        |
| 7.4      | Induction principles for well-sorted terms                              | 116        |
| 7.4.1    | Regular induction   | 116        |
| 7.4.2    | Fresh induction   | 116        |
| 7.4.3    | The syntactic constructs are almost free (on well-sorted terms)         | 118        |
| 7.5      | The non-construct operators preserve well-sortedness                    | 121        |
| 7.6      | Simplification rules for swapping, substitution, freshness and skeleton | 123        |
| 7.7      | The ability to pick fresh variables                                     | 126        |
| 7.8      | Compositionality properties of freshness and swapping                   | 128        |
| 7.8.1    | W.r.t. terms  | 128        |
| 7.8.2    | W.r.t. environments   | 129        |
| 7.8.3    | W.r.t. abstractions   | 130        |
| 7.9      | Compositionality properties for the other operators                     | 132        |
| 7.9.1    | Environment identity, update and "get" versus other operators           | 132        |
| 7.9.2    | Substitution versus other operators                                     | 133        |
| 7.9.3    | Properties specific to variable-for-variable substitution               | 140        |
| 7.9.4    | Abstraction versions of the properties                                  | 142        |
| 7.10     | Operators for down-casting and case-analyzing well-sorted items         | 148        |
| 7.10.1   | For terms   | 148        |
| 7.10.2   | For abstractions  | 152        |
| <b>8</b> | <b>Iteration</b>  | <b>155</b> |
| 8.1      | Models  | 155        |
| 8.1.1    | Raw models  | 156        |
| 8.1.2    | Well-sorted models of various kinds                                     | 156        |
| 8.2      | Morphisms of models   | 172        |
| 8.2.1    | Preservation of the domains   | 172        |
| 8.2.2    | Preservation of the constructs  | 173        |
| 8.2.3    | Preservation of freshness   | 173        |
| 8.2.4    | Preservation of swapping  | 174        |
| 8.2.5    | Preservation of subst   | 174        |
| 8.2.6    | Fresh-swap morphisms  | 174        |
| 8.2.7    | Fresh-subst morphisms   | 175        |
| 8.2.8    | Fresh-swap-subst morphisms  | 175        |

|           |   |            |
|-----------|---|------------|
| 8.2.9     | Basic facts . . . . .   | 175        |
| 8.2.10    | Identity and composition . . . . .  | 177        |
| 8.3       | The term model . . . . .  | 179        |
| 8.3.1     | Definitions and simplification rules . . . . .  | 179        |
| 8.3.2     | Well-sortedness of the term model . . . . .   | 181        |
| 8.3.3     | Direct description of morphisms from the term models . . . . .  | 184        |
| 8.3.4     | Sufficient criteria for being a morphism to a well-sorted model (of various kinds) . . . . .            | 190        |
| 8.4       | The "error" model of associated to a model . . . . .  | 191        |
| 8.4.1     | Preliminaries . . . . .   | 192        |
| 8.4.2     | Definitions and notations . . . . .   | 193        |
| 8.4.3     | Simplification rules . . . . .  | 194        |
| 8.4.4     | Nchotomies . . . . .  | 200        |
| 8.4.5     | Inversion rules . . . . .   | 201        |
| 8.4.6     | The error model is strongly well-sorted as a fresh-swap-subst and as a fresh-subst-swap model . . . . . | 203        |
| 8.4.7     | The natural morphism from an error model to its original model . . . . .                                | 207        |
| 8.5       | Initiality of the models of terms . . . . .   | 209        |
| 8.5.1     | The initial map from quasi-terms to a strong model . . . . .  | 209        |
| 8.5.2     | The initial morphism (iteration map) from the term model to any strong model . . . . .                  | 212        |
| 8.5.3     | The initial morphism (iteration map) from the term model to any model . . . . .                         | 214        |
| <b>9</b>  | <b>Interpretation of syntax in semantic domains</b>   | <b>215</b> |
| 9.1       | Semantic domains and valuations . . . . .   | 216        |
| 9.1.1     | Definitions: . . . . .  | 216        |
| 9.1.2     | Basic facts . . . . .   | 218        |
| 9.2       | Interpretation maps . . . . .   | 220        |
| 9.2.1     | Definitions . . . . .   | 220        |
| 9.2.2     | Extension of domain preservation to inputs . . . . .  | 222        |
| 9.3       | The iterative model associated to a semantic domain . . . . .   | 222        |
| 9.3.1     | Definition and basic facts . . . . .  | 223        |
| 9.3.2     | The associated model is well-structured . . . . .   | 226        |
| 9.4       | The semantic interpretation . . . . .   | 230        |
| <b>10</b> | <b>General Recursion</b>  | <b>232</b> |
| 10.1      | Raw models . . . . .  | 233        |
| 10.2      | Well-sorted models of various kinds . . . . .   | 234        |
| 10.3      | Model morphisms from the term model . . . . .   | 241        |
| 10.4      | From models to iterative models . . . . .   | 244        |
| 10.5      | The recursion-iteration "identity trick" . . . . .  | 251        |
| 10.6      | From iteration morphisms to morphisms . . . . .   | 251        |

|        |   |     |
|--------|---|-----|
| 10.7   | The recursion theorem . . . . .                                 | 255 |
| 10.8   | Models that are even “closer” to the term model . . . . .       | 256 |
| 10.8.1 | Relevant predicates on models . . . . .                         | 256 |
| 10.8.2 | Relevant predicates on maps from the term model . . . . .       | 259 |
| 10.8.3 | Criterion for the reflection of freshness . . . . .             | 260 |
| 10.8.4 | Criterion for the injectiveness of the recursive map . . . . .  | 261 |
| 10.8.5 | Criterion for the surjectiveness of the recursive map . . . . . | 261 |

## 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* +

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$

**abbreviation**  $\text{termIn} ::$

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

**abbreviation**  $\text{absIn} ::$

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

## 1.2 Induction principles

**definition**  $qTermLess :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTermItem \text{ rel}$

**where**

$qTermLess == \{(termIn X, termIn(qOp \text{ delta } inp \text{ binp})) \mid X \text{ delta } inp \text{ binp } i. \text{ inp } i = \text{Some } X\} \cup$   
 $\{(absIn A, termIn(qOp \text{ delta } inp \text{ binp})) \mid A \text{ delta } inp \text{ binp } i. \text{ binp } i = \text{Some } A\} \cup$   
 $\{(termIn X, absIn (qAbs \text{ xs } x \text{ X})) \mid X \text{ xs } x. \text{ True}\}$

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

**lemma**  $qTerm\text{-rawInduct}[case\text{-names } Var \text{ Op } Abs]:$

**fixes**  $X :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm$  **and**

$A :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$  **and**  $\text{phi } \text{phiAbs}$

**assumes**

$Var: \bigwedge \text{xs } x. \text{ phi } (qVar \text{ xs } x)$  **and**

$Op: \bigwedge \text{delta } inp \text{ binp}. \llbracket \text{liftAll } \text{phi } inp; \text{ liftAll } \text{phiAbs } \text{binp} \rrbracket \Longrightarrow \text{phi } (qOp \text{ delta } inp \text{ binp})$  **and**

$Abs: \bigwedge \text{xs } x \text{ X}. \text{ phi } X \Longrightarrow \text{phiAbs } (qAbs \text{ xs } x \text{ X})$

**shows**  $\text{phi } X \wedge \text{phiAbs } A$

$\langle \text{proof} \rangle$

**lemma**  $qTermLess\text{-wf}: \text{wf } qTermLess$

$\langle \text{proof} \rangle$

**lemma**  $qTermLessPlus\text{-wf}: \text{wf } (qTermLess \hat{+})$

$\langle \text{proof} \rangle$

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**

$qSkel :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm \Rightarrow (\text{'index, 'bindex})tree$

**and**

$qSkelAbs :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs \Rightarrow (\text{'index, 'bindex})tree$

**where**

$qSkel (qVar \text{ xs } x) = \text{Branch } (\lambda i. \text{None}) (\lambda i. \text{None})$

|

```

qSkel (qOp delta inp binp) = Branch (lift qSkel inp) (lift qSkelAbs binp)
|
qSkelAbs (qAbs xs x X) = Branch ( $\lambda i. \text{Some}(qSkel X)$ ) ( $\lambda i. \text{None}$ )
⟨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:  $\bigwedge X Y. (X, Y) \in \text{rel} \implies qSkel Y = qSkel X$  and
Var:  $\bigwedge xs x. \text{phi} (qVar xs x)$  and
Op:  $\bigwedge \text{delta inp binp}. \llbracket \text{liftAll phi inp}; \text{liftAll phiAbs binp} \rrbracket$ 
 $\implies \text{phi} (qOp \text{delta inp binp})$  and
Abs:  $\bigwedge xs x X. (\bigwedge Y. (X, Y) \in \text{rel} \implies \text{phi} Y) \implies \text{phiAbs} (qAbs xs x X)$ 
shows phi X  $\wedge$  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  $\Rightarrow$ 
('index,'bindex,'varSort,'var,'opSym)qTermItem rel
where
qTermLess-modulo rel ==
{(termIn X, termIn(qOp delta inp binp)) | X delta inp binp i. inp i = Some X}  $\cup$ 
{(absIn A, termIn(qOp delta inp binp)) | A delta inp binp j. binp j = Some A}  $\cup$ 
{(termIn Y, absIn (qAbs xs x X)) | X Y xs x. (X, Y)  $\in$  rel}

```

```

lemma qTermLess-modulo-wf:
fixes rel::('index,'bindex,'varSort,'var,'opSym)qTerm rel
assumes  $\bigwedge X Y. (X, Y) \in \text{rel} \implies qSkel Y = qSkel X$ 
shows wf (qTermLess-modulo rel)
⟨proof⟩

```

### 1.3 Swap and substitution on variables

```

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

```

```

abbreviation sw-abbrev :: 'var  $\Rightarrow$  'varSort  $\Rightarrow$  'var  $\Rightarrow$  'var  $\Rightarrow$  'varSort  $\Rightarrow$  'var

```

$(- @[- \wedge -]'- 200)$

**where**  $(x @xs[y1 \wedge y2]-ys) == sw\ ys\ y1\ y2\ xs\ x$

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

**where**

$sb\ ys\ y1\ y2\ xs\ x ==$

$if\ ys = xs\ then\ if\ x = y2\ then\ y1$   
 $else\ x$

$else\ x$

**abbreviation**  $sb-abbrev :: 'var \Rightarrow 'varSort \Rightarrow 'var \Rightarrow 'var \Rightarrow 'varSort \Rightarrow 'var$

$(- @[- '/' -]'- 200)$

**where**  $(x @xs[y1 / y2]-ys) == sb\ ys\ y1\ y2\ xs\ x$

**theorem**  $sw-simps1[simp]: (x @xs[x \wedge y]-xs) = y$   
 $\langle proof \rangle$

**theorem**  $sw-simps2[simp]: (x @xs[y \wedge x]-xs) = y$   
 $\langle proof \rangle$

**theorem**  $sw-simps3[simp]:$

$(zs \neq xs \vee x \notin \{z1, z2\}) \implies (x @xs[z1 \wedge z2]-zs) = x$   
 $\langle proof \rangle$

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

**theorem**  $sw-ident[simp]: (x @xs[y \wedge y]-ys) = x$   
 $\langle proof \rangle$

**theorem**  $sw-compose:$

$((z @zs[x \wedge y]-xs) @zs[x' \wedge y']-xs') =$   
 $((z @zs[x' \wedge y']-xs') @zs[(x @xs[x' \wedge y']-xs') \wedge (y @xs[x' \wedge y']-xs')]-xs)$   
 $\langle proof \rangle$

**theorem**  $sw-commute:$

**assumes**  $zs \neq zs' \vee \{x, y\} \cap \{x', y'\} = \{\}$

**shows**  $((u @us[x \wedge y]-zs) @us[x' \wedge y']-zs') = ((u @us[x' \wedge y']-zs') @us[x \wedge y]-zs)$   
 $\langle proof \rangle$

**theorem**  $sw-involutive[simp]:$

$((z @zs[x \wedge y]-xs) @zs[x \wedge y]-xs) = z$   
 $\langle proof \rangle$

**theorem**  $sw-inj[simp]:$

$((z @zs[x \wedge y]-xs) = (z' @zs[x \wedge y]-xs)) = (z = z')$   
 $\langle proof \rangle$

**lemma**  $sw-preserves-mship[simp]:$

**assumes**  $\{y1, y2\} \subseteq Var\ ys$



**shows**  $((x @xs[y1 \wedge y2]-ys) \in Var\ xs) = (x \in Var\ xs)$   
 $\langle proof \rangle$

**theorem sw-sym:**  
 $(z @zs[x \wedge y]-xs) = (z @zs[y \wedge x]-xs)$   
 $\langle proof \rangle$

**theorem sw-involutive2[simp]:**  
 $((z @zs[x \wedge y]-xs) @zs[y \wedge x]-xs) = z$   
 $\langle proof \rangle$

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

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

**theorem sb-simps1[simp]:**  $(x @xs[y / x]-xs) = y$   
 $\langle proof \rangle$

**theorem sb-simps2[simp]:**  
 $(zs \neq xs \vee z2 \neq x) \implies (x @xs[z1 / z2]-zs) = x$   
 $\langle proof \rangle$

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

**theorem sb-ident[simp]:**  $(x @xs[y / y]-ys) = x$   
 $\langle proof \rangle$

**theorem sb-compose1:**  
 $((z @zs[y1 / x]-xs) @zs[y2 / x]-xs) = (z @zs[(y1 @xs[y2 / x]-xs) / x]-xs)$   
 $\langle proof \rangle$

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

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

**theorem sb-idem[simp]:**  
 $((z @zs[x / y]-xs) @zs[x / y]-xs) = (z @zs[x / y]-xs)$   
 $\langle proof \rangle$

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

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

**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**  
 $aux-qSwap-ignoreFirst3 K =$   
 $(\text{case } K \text{ of } \text{Inl}(zs, x, y, X) \Rightarrow \text{termIn } X$   
 $\quad | \text{Inr}(zs, x, y, A) \Rightarrow \text{absIn } A)$

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

**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**  
 $qSwap \text{ } zs \text{ } x \text{ } y \text{ } (qVar \text{ } zs' \text{ } z) = qVar \text{ } zs' \text{ } (z @zs'[x \wedge y]-zs)$   
 $|$   
 $qSwap \text{ } zs \text{ } x \text{ } y \text{ } (qOp \text{ } delta \text{ } inp \text{ } binp) =$   
 $qOp \text{ } delta \text{ } (\text{lift } (qSwap \text{ } zs \text{ } x \text{ } y) \text{ } inp) \text{ } (\text{lift } (qSwapAbs \text{ } zs \text{ } x \text{ } y) \text{ } binp)$   
 $|$

$qSwapAbs\ zs\ x\ y\ (qAbs\ zs'\ z\ X) = qAbs\ zs'\ (z\ @zs'[x\ \wedge\ y]-zs)\ (qSwap\ zs\ x\ y\ X)$   
 <proof>

**termination**

<proof>

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

**abbreviation**  $qSwap-abbrev ::$

$(index, bindex, varSort, var, opSym)qTerm \Rightarrow var \Rightarrow var \Rightarrow varSort \Rightarrow$

$(index, bindex, varSort, var, opSym)qTerm\ (-\ #[[-\ \wedge\ -]]'\ --\ 200)$

**where**  $(X\ #[[z1\ \wedge\ z2]]-zs) == qSwap\ zs\ z1\ z2\ X$

**abbreviation**  $qSwapAbs-abbrev ::$

$(index, bindex, varSort, var, opSym)qAbs \Rightarrow var \Rightarrow var \Rightarrow varSort \Rightarrow$

$(index, bindex, varSort, var, opSym)qAbs\ (-\ \$[[[-\ \wedge\ -]]'\ --\ 200)$

**where**  $(A\ \$[[z1\ \wedge\ z2]]-zs) == qSwapAbs\ zs\ z1\ z2\ A$

**definition**

$aux-qFresh-ignoreFirst2 ::$

$varSort * var * (index, bindex, varSort, var, opSym)qTerm +$

$varSort * var * (index, bindex, varSort, var, opSym)qAbs \Rightarrow$

$(index, bindex, varSort, var, opSym)qTermItem$

**where**

$aux-qFresh-ignoreFirst2\ K =$

$(case\ K\ of\ Inl(zs, x, X) \Rightarrow termIn\ X$

$| Inr\ (zs, x, A) \Rightarrow absIn\ A)$

**lemma**  $qTermLess-ignoreFirst2-wf: wf(inv-image\ qTermLess\ aux-qFresh-ignoreFirst2)$

<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**

$qAFresh :: varSort \Rightarrow var \Rightarrow (index, bindex, varSort, var, opSym)qTerm \Rightarrow bool$

**and**

$qAFreshAbs :: varSort \Rightarrow var \Rightarrow (index, bindex, varSort, var, opSym)qAbs \Rightarrow$

$bool$

**where**

$qAFresh\ xs\ x\ (qVar\ ys\ y) = (xs \neq ys \vee x \neq y)$

|

$qAFresh\ xs\ x\ (qOp\ delta\ inp\ binp) =$

$(liftAll\ (qAFresh\ xs\ x)\ inp \wedge liftAll\ (qAFreshAbs\ xs\ x)\ binp)$

|

$qAFreshAbs\ xs\ x\ (qAbs\ ys\ y\ X) = ((xs \neq ys \vee x \neq y) \wedge qAFresh\ xs\ x\ X)$

<proof>

**termination**

<proof>

**lemmas**  $qAFreshAll\text{-simps} = qAFresh.\text{simps } qAFreshAbs.\text{simps}$

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

**function**

$qFresh :: 'varSort \Rightarrow 'var \Rightarrow ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm \Rightarrow bool$

**and**

$qFreshAbs :: 'varSort \Rightarrow 'var \Rightarrow ('index, 'bindex, 'varSort, 'var, 'opSym)qAbs \Rightarrow bool$

**where**

$qFresh\ xs\ x\ (qVar\ ys\ y) = (xs \neq ys \vee x \neq y)$

$qFresh\ xs\ x\ (qOp\ delta\ inp\ binp) =$   
 $(liftAll\ (qFresh\ xs\ x)\ inp \wedge liftAll\ (qFreshAbs\ xs\ x)\ binp)$

$qFreshAbs\ xs\ x\ (qAbs\ ys\ y\ X) = ((xs = ys \wedge x = y) \vee qFresh\ xs\ x\ X)$

$\langle proof \rangle$

**termination**

$\langle proof \rangle$

**lemmas**  $qFreshAll\text{-simps} = qFresh.\text{simps } qFreshAbs.\text{simps}$

## 1.5 Compositional properties of swapping

**lemma**  $qSwapAll\text{-ident}$ :

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**

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

**shows**  $(X \#[[x \wedge x]]\text{-zs}) = X \wedge (A \#[[x \wedge x]]\text{-zs}) = A$

$\langle proof \rangle$

**corollary**  $qSwap\text{-ident}[simp]$ :  $(X \#[[x \wedge x]]\text{-zs}) = X$

$\langle proof \rangle$

**lemma**  $qSwapAll\text{-compose}$ :

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**

$A :: ('index, 'bindex, 'varSort, 'var, 'opSym)qAbs$  **and**  $zs\ x\ y\ x'\ y'$

**shows**

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

$\wedge$   
 $((A \#[[x \wedge y]]\text{-zs}) \#[[x' \wedge y']\text{-zs}']) =$   
 $((A \#[[x' \wedge y']\text{-zs}']) \#[[(x @zs[x' \wedge y']\text{-zs}') \wedge (y @zs[x' \wedge y']\text{-zs}')]]\text{-zs})$

$\langle proof \rangle$

**corollary**  $qSwap\text{-compose}$ :

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

$\langle proof \rangle$

**lemma** *qSwap-commute*:

**assumes**  $zs \neq zs' \vee \{x, y\} \text{ Int } \{x', y'\} = \{\}$

**shows**  $((X \#[[x \wedge y]]-zs) \#[[x' \wedge y']] - zs') = ((X \#[[x' \wedge y']] - zs') \#[[x \wedge y]] - zs)$   
 $\langle \text{proof} \rangle$

**lemma** *qSwapAll-involutive*:

**fixes**  $X::('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**

$A::('index, 'bindex, 'varSort, 'var, 'opSym)qAbs$  **and**  $zs\ x\ y$

**shows**  $((X \#[[x \wedge y]]-zs) \#[[x \wedge y]]-zs) = X \wedge$

$((A \#[[x \wedge y]]-zs) \#[[x \wedge y]]-zs) = A$

$\langle \text{proof} \rangle$

**corollary** *qSwap-involutive[simp]*:

$((X \#[[x \wedge y]]-zs) \#[[x \wedge y]]-zs) = X$

$\langle \text{proof} \rangle$

**lemma** *qSwapAll-sym*:

**fixes**  $X::('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**

$A::('index, 'bindex, 'varSort, 'var, 'opSym)qAbs$  **and**  $zs\ x\ y$

**shows**  $(X \#[[x \wedge y]]-zs) = (X \#[[y \wedge x]]-zs) \wedge$

$(A \#[[x \wedge y]]-zs) = (A \#[[y \wedge x]]-zs)$

$\langle \text{proof} \rangle$

**corollary** *qSwap-sym*:

$(X \#[[x \wedge y]]-zs) = (X \#[[y \wedge x]]-zs)$

$\langle \text{proof} \rangle$

**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**  $(qAFresh\ zs\ z1\ X \wedge qAFresh\ zs\ z2\ X \longrightarrow (X \#[[z1 \wedge z2]]-zs) = X) \wedge$

$(qAFreshAbs\ zs\ z1\ A \wedge qAFreshAbs\ zs\ z2\ A \longrightarrow (A \#[[z1 \wedge z2]]-zs) = A)$

$\langle \text{proof} \rangle$

**corollary** *qAFresh-qSwap-id[simp]*:

$\llbracket qAFresh\ zs\ z1\ X; qAFresh\ zs\ z2\ X \rrbracket \Longrightarrow (X \#[[z1 \wedge z2]]-zs) = X$

$\langle \text{proof} \rangle$

**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**  $(qAFresh\ zs\ y\ X \wedge qAFresh\ zs\ z\ X \longrightarrow$

$((X \#[[y \wedge x]]-zs) \#[[z \wedge y]]-zs) = (X \#[[z \wedge x]]-zs) \wedge$

$(qAFreshAbs\ zs\ y\ A \wedge qAFreshAbs\ zs\ z\ A \longrightarrow$

$((A \#[[y \wedge x]]-zs) \#[[z \wedge y]]-zs) = (A \#[[z \wedge x]]-zs))$

$\langle \text{proof} \rangle$

**corollary** *qAFresh-qSwap-compose*:

$\llbracket qAFresh\ zs\ y\ X; qAFresh\ zs\ z\ X \rrbracket \implies$   
 $((X \# \llbracket y \wedge x \rrbracket -zs) \# \llbracket z \wedge y \rrbracket -zs) = (X \# \llbracket z \wedge x \rrbracket -zs)$   
 $\langle proof \rangle$

## 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**  $qSkel(X \# \llbracket x \wedge y \rrbracket -zs) = qSkel\ X \wedge$   
 $qSkelAbs(A \ \$ \llbracket x \wedge y \rrbracket -zs) = qSkelAbs\ A$

$\langle proof \rangle$

**corollary** *qSkel-qSwap*:  $qSkel(X \# \llbracket x \wedge y \rrbracket -zs) = qSkel\ X$

$\langle proof \rangle$

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*:  $\llbracket (X, Y) \in qSwapped; (Y, Z) \in qSwapped \rrbracket \implies (X, Z) \in qSwapped$

|

*Swap*:  $(X, Y) \in qSwapped \implies (X, Y \# \llbracket x \wedge y \rrbracket -zs) \in qSwapped$

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

**lemma** *qSwap-qSwapped*:  $(X, X \# \llbracket x \wedge y \rrbracket -zs) : qSwapped$

$\langle proof \rangle$

**lemma** *qSwapped-qSkel*:

$(X, Y) \in qSwapped \implies qSkel\ Y = qSkel\ X$

$\langle proof \rangle$

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**  $\phi \phi Abs$   
**assumes**  
 $Var: \bigwedge xs\ x. \phi (qVar\ xs\ x)$  **and**  
 $Op: \bigwedge \delta\ inp\ binp. [\text{liftAll } \phi\ inp; \text{liftAll } \phi Abs\ binp]$   
 $\implies \phi (qOp\ \delta\ inp\ binp)$  **and**  
 $Abs: \bigwedge xs\ x\ X. [\bigwedge Y. (X, Y) \in qSwapped \implies \phi Y;$   
 $\bigwedge Y. qSkel\ Y = qSkel\ X \implies \phi Y]$   
 $\implies \phi Abs (qAbs\ xs\ x\ X)$   
**shows**  $\phi X \wedge \phi Abs\ A$   
 $\langle proof \rangle$

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

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

**lemma**  $qTermQSwappedLess\text{-wf}: wf\ qTermQSwappedLess$   
 $\langle proof \rangle$

## 1.7 More properties connecting swapping and freshness

**lemma**  $qSwap\text{-}3commute$ :  
**assumes**  $*$ :  $qAFresh\ ys\ y\ X$  **and**  $**$ :  $qAFresh\ ys\ y0\ X$   
**and**  $***$ :  $ys \neq zs \vee y0 \notin \{z1, z2\}$   
**shows**  $((X \#[[z1 \wedge z2]]\text{-}zs) \#[[y0 \wedge x \ @ys[z1 \wedge z2]]\text{-}zs])\text{-}ys) =$   
 $((X \#[[y \wedge x]]\text{-}ys) \#[[y0 \wedge y]]\text{-}ys) \#[[z1 \wedge z2]]\text{-}zs)$   
 $\langle proof \rangle$

**lemma**  $qAFreshAll\text{-}imp\text{-}qFreshAll$ :  
**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**  
 $A :: ('index, 'bindex, 'varSort, 'var, 'opSym)qAbs$  **and**  $xs\ x$   
**shows**  $(qAFresh\ xs\ x\ X \longrightarrow qFresh\ xs\ x\ X) \wedge$   
 $(qAFreshAbs\ xs\ x\ A \longrightarrow qFreshAbs\ xs\ x\ A)$   
 $\langle proof \rangle$

**corollary**  $qAFresh\text{-}imp\text{-}qFresh$ :  
 $qAFresh\ xs\ x\ X \implies qFresh\ xs\ x\ X$   
 $\langle proof \rangle$

**lemma**  $qSwapAll\text{-}preserves\text{-}qAFreshAll$ :  
**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**  
 $A :: ('index, 'bindex, 'varSort, 'var, 'opSym)qAbs$  **and**  $ys\ y\ zs\ z1\ z2$   
**shows**  
 $(qAFresh\ ys\ (y \ @ys[z1 \wedge z2]]\text{-}zs) (X \#[[z1 \wedge z2]]\text{-}zs) = qAFresh\ ys\ y\ X) \wedge$   
 $(qAFreshAbs\ ys\ (y \ @ys[z1 \wedge z2]]\text{-}zs) (A \#[[z1 \wedge z2]]\text{-}zs) = qAFreshAbs\ ys\ y\ A)$   
 $\langle proof \rangle$

**corollary** *qSwap-preserves-qAFresh[simp]*:  
 $(qAFresh\ ys\ (y\ @ys[z1\ \wedge\ z2]-zs)\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qAFresh\ ys\ y\ X)$   
 $\langle proof \rangle$

**lemma** *qSwap-preserves-qAFresh-distinct*:  
**assumes**  $ys \neq zs \vee y \notin \{z1, z2\}$   
**shows**  $qAFresh\ ys\ y\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qAFresh\ ys\ y\ X$   
 $\langle proof \rangle$

**lemma** *qAFresh-qSwap-exchange1*:  
 $qAFresh\ zs\ z2\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qAFresh\ zs\ z1\ X$   
 $\langle proof \rangle$

**lemma** *qAFresh-qSwap-exchange2*:  
 $qAFresh\ zs\ z2\ (X\ \#[[z2\ \wedge\ z1]]-zs) = qAFresh\ zs\ z1\ X$   
 $\langle proof \rangle$

**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**  
 $(qFresh\ ys\ (y\ @ys[z1\ \wedge\ z2]-zs)\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qFresh\ ys\ y\ X) \wedge$   
 $(qFreshAbs\ ys\ (y\ @ys[z1\ \wedge\ z2]-zs)\ (A\ \$[[z1\ \wedge\ z2]]-zs) = qFreshAbs\ ys\ y\ A)$   
 $\langle proof \rangle$

**corollary** *qSwap-preserves-qFresh*:  
 $(qFresh\ ys\ (y\ @ys[z1\ \wedge\ z2]-zs)\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qFresh\ ys\ y\ X)$   
 $\langle proof \rangle$

**lemma** *qSwap-preserves-qFresh-distinct*:  
**assumes**  $ys \neq zs \vee y \notin \{z1, z2\}$   
**shows**  $qFresh\ ys\ y\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qFresh\ ys\ y\ X$   
 $\langle proof \rangle$

**lemma** *qFresh-qSwap-exchange1*:  
 $qFresh\ zs\ z2\ (X\ \#[[z1\ \wedge\ z2]]-zs) = qFresh\ zs\ z1\ X$   
 $\langle proof \rangle$

**lemma** *qFresh-qSwap-exchange2*:  
 $qFresh\ zs\ z1\ X = qFresh\ zs\ z2\ (X\ \#[[z2\ \wedge\ z1]]-zs)$   
 $\langle proof \rangle$

**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**  
 $(qGood\ X \longrightarrow qGood\ (X\ \#[[x\ \wedge\ y]]-zs)) \wedge$   
 $(qGoodAbs\ A \longrightarrow qGoodAbs\ (A\ \$[[x\ \wedge\ y]]-zs))$   
 $\langle proof \rangle$

**corollary** *qSwap-preserves-qGood1*:  
 $qGood\ X \Longrightarrow qGood\ (X\ \#[[x\ \wedge\ y]]-zs)$   
 $\langle proof \rangle$

**corollary** *qSwapAbs-preserves-qGoodAbs1*:  
 $qGoodAbs\ A \Longrightarrow qGoodAbs\ (A\ \$[[x\ \wedge\ y]]-zs)$   
 $\langle proof \rangle$

**lemma** *qSwap-preserves-qGood2*:  
**assumes**  $qGood(X\ \#[[x\ \wedge\ y]]-zs)$   
**shows**  $qGood\ X$   
 $\langle proof \rangle$

**lemma** *qSwapAbs-preserves-qGoodAbs2*:  
**assumes**  $qGoodAbs(A\ \$[[x\ \wedge\ y]]-zs)$   
**shows**  $qGoodAbs\ A$   
 $\langle proof \rangle$

**lemma** *qSwap-preserves-qGood*:  $(qGood\ (X\ \#[[x\ \wedge\ y]]-zs)) = (qGood\ X)$   
 $\langle proof \rangle$

**lemma** *qSwapAbs-preserves-qGoodAbs*:  
 $(qGoodAbs\ (A\ \$[[x\ \wedge\ y]]-zs)) = (qGoodAbs\ A)$   
 $\langle proof \rangle$

**lemma** *qSwap-twice-preserves-qGood*:  
 $(qGood\ ((X\ \#[[x\ \wedge\ y]]-zs)\ \#[[x'\ \wedge\ y']-zs'])) = (qGood\ X)$   
 $\langle proof \rangle$

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

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

$$\begin{aligned}
& \text{liftAll } (\lambda X. qGood X \wedge phi X) \text{ inp}; \\
& \text{liftAll } (\lambda A. qGoodAbs A \wedge phiAbs A) \text{ binp}] \\
& \implies phi (qOp \text{ delta inp binp}) \text{ and} \\
\text{Abs: } \bigwedge xs x X. \llbracket qGood X; \bigwedge Y. (X, Y) \in rel \implies phi Y \rrbracket \\
& \implies phiAbs (qAbs xs x X)
\end{aligned}$$

**shows**

$$(qGood X \longrightarrow phi X) \wedge (qGoodAbs A \longrightarrow phiAbs A)$$

*<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**

$Var: \bigwedge xs x. phi (qVar xs x)$  **and**

$$\begin{aligned}
\text{Op: } \bigwedge \text{delta inp binp. } \llbracket \{i. \text{inp } i \neq \text{None}\} \llcorner o \mid UNIV :: 'var \text{ set}; \\
\{i. \text{binp } i \neq \text{None}\} \llcorner o \mid UNIV :: 'var \text{ set}; \\
\text{liftAll } (\lambda X. qGood X \wedge phi X) \text{ inp}; \\
\text{liftAll } (\lambda A. qGoodAbs A \wedge phiAbs A) \text{ binp} \rrbracket \\
\implies phi (qOp \text{ delta inp binp}) \text{ and}
\end{aligned}$$

$\text{Abs: } \bigwedge xs x X. \llbracket qGood X; phi X \rrbracket \implies phiAbs (qAbs xs x X)$

**shows**  $(qGood X \longrightarrow phi X) \wedge (qGoodAbs A \longrightarrow phiAbs A)$

*<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**

$Var: \bigwedge xs x. phi (qVar xs x)$  **and**

$$\begin{aligned}
\text{Op: } \bigwedge \text{delta inp binp. } \llbracket \{i. \text{inp } i \neq \text{None}\} \llcorner o \mid UNIV :: 'var \text{ set}; \\
\{i. \text{binp } i \neq \text{None}\} \llcorner o \mid UNIV :: 'var \text{ set}; \\
\text{liftAll } (\lambda X. qGood X \wedge phi X) \text{ inp}; \\
\text{liftAll } (\lambda A. qGoodAbs A \wedge phiAbs A) \text{ binp} \rrbracket \\
\implies phi (qOp \text{ delta inp binp}) \text{ and}
\end{aligned}$$

$\text{Abs: } \bigwedge xs x X. \llbracket qGood X;$

$$\bigwedge Y. qGood Y \wedge qSkel Y = qSkel X \implies phi Y;$$

$$\bigwedge Y. (X, Y) \in qSwapped \implies phi Y \rrbracket$$

$$\implies phiAbs (qAbs xs x X)$$

**shows**

$$(qGood X \longrightarrow phi X) \wedge (qGoodAbs A \longrightarrow phiAbs A)$$

*<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**

$Var1: \bigwedge xs x. phi1 (qVar xs x)$  **and**

$Var2: \bigwedge xs x. phi2 (qVar xs x)$  **and**

$Op1: \bigwedge \text{delta inp binp. } \llbracket \text{liftAll } (\lambda X. qGood X \wedge \text{phi1 } X) \text{ inp};$   
 $\quad \text{liftAll } (\lambda A. qGoodAbs A \wedge \text{phiAbs1 } A) \text{ binp} \rrbracket$   
 $\implies \text{phi1 } (qOp \text{ delta inp binp}) \text{ and}$

$Op2: \bigwedge \text{delta inp binp. } \llbracket \text{liftAll } (\lambda X. qGood X \wedge \text{phi2 } X) \text{ inp};$   
 $\quad \text{liftAll } (\lambda A. qGoodAbs A \wedge \text{phiAbs2 } A) \text{ binp} \rrbracket$   
 $\implies \text{phi2 } (qOp \text{ delta inp binp}) \text{ and}$

$Abs1: \bigwedge xs x X. \llbracket qGood X;$   
 $\quad \bigwedge Y. qGood Y \wedge qSkel Y = qSkel X \implies \text{phi1 } Y;$   
 $\quad \bigwedge Y. qGood Y \wedge qSkel Y = qSkel X \implies \text{phi2 } Y;$   
 $\quad \bigwedge Y. (X, Y) \in qSwapped \implies \text{phi1 } Y;$   
 $\quad \bigwedge Y. (X, Y) \in qSwapped \implies \text{phi2 } Y \rrbracket$   
 $\implies \text{phiAbs1 } (qAbs xs x X) \text{ and}$

$Abs2: \bigwedge xs x X. \llbracket qGood X;$   
 $\quad \bigwedge Y. qGood Y \wedge qSkel Y = qSkel X \implies \text{phi1 } Y;$   
 $\quad \bigwedge Y. qGood Y \wedge qSkel Y = qSkel X \implies \text{phi2 } Y;$   
 $\quad \bigwedge Y. (X, Y) \in qSwapped \implies \text{phi1 } Y;$   
 $\quad \bigwedge Y. (X, Y) \in qSwapped \implies \text{phi2 } Y;$   
 $\quad \text{phiAbs1 } (qAbs xs x X) \rrbracket$   
 $\implies \text{phiAbs2 } (qAbs xs x X)$

**shows**

$(qGood X \longrightarrow (\text{phi1 } X \wedge \text{phi2 } X)) \wedge$   
 $(qGoodAbs A \longrightarrow (\text{phiAbs1 } A \wedge \text{phiAbs2 } A))$   
 $\langle \text{proof} \rangle$

### 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**

$(qGood X \longrightarrow |\{x. \neg qAFresh xs x X\}| <_o |UNIV :: 'var set|) \wedge$   
 $(qGoodAbs A \longrightarrow |\{x. \neg qAFreshAbs xs x A\}| <_o |UNIV :: 'var set|)$   
 $\langle \text{proof} \rangle$

**corollary** *single-non-qAFresh-ordLess-var:*

$qGood X \implies |\{x. \neg qAFresh xs x X\}| <_o |UNIV :: 'var set|$   
 $\langle \text{proof} \rangle$

**corollary** *single-non-qAFreshAbs-ordLess-var:*

$qGoodAbs A \implies |\{x. \neg qAFreshAbs xs x A\}| <_o |UNIV :: 'var set|$   
 $\langle \text{proof} \rangle$

**lemma** *single-non-qFresh-ordLess-var:*

**assumes**  $qGood X$

**shows**  $|\{x. \neg qFresh xs x X\}| <_o |UNIV :: 'var set|$

$\langle \text{proof} \rangle$

**lemma** *single-non-qFreshAbs-ordLess-var:*

**assumes**  $qGoodAbs A$

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

**lemma** *non-qAFresh-ordLess-var*:

**assumes** *GOOD*:  $\forall X \in XS. qGood\ X$  **and** *Var*:  $|XS| < o\ |UNIV :: 'var\ set|$   
**shows**  $|\{x\ x\ X. X \in XS \wedge \neg qAFresh\ xs\ x\ X\}| < o\ |UNIV :: 'var\ set|$   
 $\langle proof \rangle$

**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**  $|\{x\ x\ X. (x \in V \vee (X \in XS \wedge \neg qAFresh\ xs\ x\ X))\}| < o\ |UNIV :: 'var\ set|$   
 $\langle proof \rangle$

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

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

**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**  $\exists x. x \notin V \wedge (\forall X \in XS. qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$   
 $\langle proof \rangle$

**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**  $\exists x. x \notin V \wedge (\forall X \in XS. qAFresh\ xs\ x\ X \wedge qFresh\ xs\ x\ X)$   
 $\langle proof \rangle$

**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**  $\text{pickQFresh } xs \ V \ XS \notin V \wedge$   
 $(\forall X \in XS. \text{qAFresh } xs \ (\text{pickQFresh } xs \ V \ XS) \ X \wedge \text{qFresh } xs \ (\text{pickQFresh}$   
 $xs \ V \ XS) \ X)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{pickQFresh}$ :

**assumes**  $\text{finite } V \vee |V| < o \ |UNIV| :: 'var \ set$  **and**  $\text{finite } XS \vee |XS| < o \ |UNIV$   
 $:: 'var \ set$  **and**

$\forall X \in XS. \text{qGood } X$

**shows**  $\text{pickQFresh } xs \ V \ XS \notin V \wedge$

$(\forall X \in XS. \text{qAFresh } xs \ (\text{pickQFresh } xs \ V \ XS) \ X \wedge \text{qFresh } xs \ (\text{pickQFresh}$   
 $xs \ V \ XS) \ X)$

$\langle \text{proof} \rangle$

**end**

## 2.4 Alpha-equivalence

### 2.4.1 Definition

**definition**  $\text{aux-alpha-ignoreSecond} ::$

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{qTerm} * (\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{qTerm}$

$+$

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{qAbs} * (\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{qAbs}$

$\Rightarrow$

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{qTermItem}$

**where**

$\text{aux-alpha-ignoreSecond } K ==$

$\text{case } K \text{ of } \text{Inl}(X, Y) \Rightarrow \text{termIn } X$

$\text{Inr}(A, B) \Rightarrow \text{absIn } A$

**lemma**  $\text{aux-alpha-ignoreSecond-qTermLessQSwapped-wf}$ :

$\text{wf}(\text{inv-image } \text{qTermQSwappedLess } \text{aux-alpha-ignoreSecond})$

$\langle \text{proof} \rangle$

**function**

$\text{alpha}$  **and**  $\text{alphaAbs}$

**where**

$\text{alpha } (\text{qVar } xs \ x) (\text{qVar } xs' \ x') \longleftrightarrow xs = xs' \wedge x = x'$

|

$\text{alpha } (\text{qOp } \text{delta } \text{inp } \text{binp}) (\text{qOp } \text{delta}' \ \text{inp}' \ \text{binp}') \longleftrightarrow$

$\text{delta} = \text{delta}' \wedge \text{sameDom } \text{inp } \text{inp}' \wedge \text{sameDom } \text{binp } \text{binp}' \wedge$

$\text{liftAll2 } \text{alpha } \text{inp } \text{inp}' \wedge$

$\text{liftAll2 } \text{alphaAbs } \text{binp } \text{binp}'$

|

$\text{alpha } (\text{qVar } xs \ x) (\text{qOp } \text{delta}' \ \text{inp}' \ \text{binp}') \longleftrightarrow \text{False}$

|

$\text{alpha } (\text{qOp } \text{delta } \text{inp } \text{binp}) (\text{qVar } xs' \ x') \longleftrightarrow \text{False}$

|

$alphaAbs (qAbs\ xs\ x\ X) (qAbs\ xs'\ x'\ X') \longleftrightarrow$   
 $xs = xs' \wedge$   
 $(\exists y. y \notin \{x, x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs'\ y\ X' \wedge$   
 $alpha (X \#[[y \wedge x]]-xs) (X' \#[[y \wedge x']]-xs'))$   
 <proof>  
**termination**  
 <proof>

**abbreviation**  $alpha\text{-abbrev}$  (**infix**  $\# = 50$ ) **where**  $X \# = Y \equiv alpha\ X\ Y$   
**abbreviation**  $alphaAbs\text{-abbrev}$  (**infix**  $\$ = 50$ ) **where**  $A \$ = B \equiv alphaAbs\ A\ B$

**context**  $FixVars$   
**begin**

## 2.4.2 Simplification and elimination rules

**lemma**  $alpha\text{-inp}\text{-None}$ :  
 $qOp\ delta\ inp\ binp\ \# =\ qOp\ delta'\ inp'\ binp' \implies$   
 $(inp\ i = None) = (inp'\ i = None)$   
 <proof>

**lemma**  $alpha\text{-binp}\text{-None}$ :  
 $qOp\ delta\ inp\ binp\ \# =\ qOp\ delta'\ inp'\ binp' \implies$   
 $(binp\ i = None) = (binp'\ i = None)$   
 <proof>

**lemma**  $qVar\text{-alpha}\text{-iff}$ :  
 $qVar\ xs\ x\ \# =\ X' \longleftrightarrow X' = qVar\ xs\ x$   
 <proof>

**lemma**  $alpha\text{-}qVar\text{-iff}$ :  
 $X \# = qVar\ xs'\ x' \longleftrightarrow X = qVar\ xs'\ x'$   
 <proof>

**lemma**  $qOp\text{-alpha}\text{-iff}$ :  
 $qOp\ delta\ inp\ binp\ \# =\ X' \longleftrightarrow$   
 $(\exists inp'\ binp'.$   
 $X' = qOp\ delta\ inp'\ binp' \wedge sameDom\ inp\ inp' \wedge sameDom\ binp\ binp' \wedge$   
 $liftAll2\ (\lambda Y\ Y'. Y \# = Y')\ inp\ inp' \wedge$   
 $liftAll2\ (\lambda A\ A'. A \$ = A')\ binp\ binp')$   
 <proof>

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



*<proof>*

**lemma** *qAbs-alphaAbs-iff*:

$qAbs\ xs\ x\ X\ \$ = A' \longleftrightarrow$

$(\exists\ x'\ y\ X'.\ A' = qAbs\ xs\ x'\ X' \wedge$   
 $y \notin \{x, x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \wedge$   
 $(X\ \#[[y \wedge x]]-xs) \# = (X'\ \#[[y \wedge x']]-xs))$

*<proof>*

**lemma** *alphaAbs-qAbs-iff*:

$A\ \$ = qAbs\ xs'\ x'\ X' \longleftrightarrow$

$(\exists\ x\ y\ X.\ A = qAbs\ xs'\ x\ X \wedge$   
 $y \notin \{x, x'\} \wedge qAFresh\ xs'\ y\ X \wedge qAFresh\ xs'\ y\ X' \wedge$   
 $(X\ \#[[y \wedge x]]-xs') \# = (X'\ \#[[y \wedge x']]-xs'))$

*<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**  $X \# = X' \implies qSkel\ X = qSkel\ X'$

*<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**  $X \# = X' \implies X' \# = X$

*<proof>*

**corollary** *alphaAbs-sym*:

**fixes**  $A A' :: ('index, 'bindex, 'varSort, 'var, 'opSym) qAbs$

**shows**  $A \$= A' \implies A' \$= A$

*<proof>*

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

**lemma** *alphaAll-refl*:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym) qTerm$  **and**

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

**shows**

$(qGood X \longrightarrow X \# = X) \wedge (qGoodAbs A \longrightarrow A \$ = A)$

*<proof>*

**corollary** *alpha-refl*:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym) qTerm$

**shows**  $qGood X \implies X \# = X$

*<proof>*

**corollary** *alphaAbs-refl*:

**fixes**  $A :: ('index, 'bindex, 'varSort, 'var, 'opSym) qAbs$

**shows**  $qGoodAbs A \implies A \$ = A$

*<proof>*

**lemma** *alphaAll-preserves-qGoodAll1*:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym) qTerm$  **and**

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

**shows**

$(qGood X \longrightarrow (\forall X'. X \# = X' \longrightarrow qGood X')) \wedge$   
 $(qGoodAbs A \longrightarrow (\forall A'. A \$ = A' \longrightarrow qGoodAbs A'))$

*<proof>*

**corollary** *alpha-preserves-qGood1*:

$\llbracket X \# = X'; qGood X \rrbracket \implies qGood X'$

*<proof>*

**corollary** *alphaAbs-preserves-qGoodAbs1*:

$\llbracket A \$ = A'; qGoodAbs A \rrbracket \implies qGoodAbs A'$

*<proof>*

**lemma** *alpha-preserves-qGood2*:

$\llbracket X \# = X'; qGood X \rrbracket \implies qGood X$

*<proof>*

**lemma** *alphaAbs-preserves-qGoodAbs2*:

$\llbracket A \$ = A'; qGoodAbs A \rrbracket \implies qGoodAbs A$

*<proof>*

**lemma** *alpha-preserves-qGood*:

$X \# = X' \implies qGood\ X = qGood\ X'$

*<proof>*

**lemma** *alphaAbs-preserves-qGoodAbs*:

$A \$ = A' \implies qGoodAbs\ A = qGoodAbs\ A'$

*<proof>*

**lemma** *alpha-qSwap-preserves-qGood1*:

**assumes** *ALPHA*:  $(X \#[[y \wedge x]]-zs) \# = (X' \#[[y' \wedge x']] -zs')$  **and**  
*GOOD*:  $qGood\ X$

**shows**  $qGood\ X'$

*<proof>*

**lemma** *alpha-qSwap-preserves-qGood2*:

**assumes** *ALPHA*:  $(X \#[[y \wedge x]]-zs) \# = (X' \#[[y' \wedge x']] -zs')$  **and**  
*GOOD'*:  $qGood\ X'$

**shows**  $qGood\ X$

*<proof>*

**lemma** *alphaAbs-qSwapAbs-preserves-qGoodAbs2*:

**assumes** *ALPHA*:  $(A \$[[y \wedge x]]-zs) \$ = (A' \$[[y' \wedge x']] -zs')$  **and**  
*GOOD'*:  $qGoodAbs\ A'$

**shows**  $qGoodAbs\ A$

*<proof>*

**lemma** *alpha-qSwap-preserves-qGood*:

**assumes** *ALPHA*:  $(X \#[[y \wedge x]]-zs) \# = (X' \#[[y' \wedge x']] -zs')$

**shows**  $qGood\ X = qGood\ X'$

*<proof>*

**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**

$(qGood\ X \longrightarrow (\forall X' zs\ z1\ z2. X \# = X' \longrightarrow$   
 $(X \#[[z1 \wedge z2]]-zs) \# = (X' \#[[z1 \wedge z2]] -zs))) \wedge$   
 $(qGoodAbs\ A \longrightarrow (\forall A' zs\ z1\ z2. A \$ = A' \longrightarrow$   
 $(A \$[[z1 \wedge z2]]-zs) \$ = (A' \$[[z1 \wedge z2]] -zs)))$

*<proof>*

**corollary** *qSwap-preserves-alpha*:

**assumes**  $qGood\ X \vee qGood\ X'$  **and**  $X \# = X'$

**shows**  $(X \#[[z1 \wedge z2]]-zs) \# = (X' \#[[z1 \wedge z2]] -zs)$

*<proof>*

**corollary** *qSwapAbs-preserves-alphaAbs*:

**assumes**  $qGoodAbs\ A \vee qGoodAbs\ A'$  **and**  $A \$ = A'$

**shows**  $(A \$[[z1 \wedge z2]]-zs) \$ = (A' \$[[z1 \wedge z2]] -zs)$

*<proof>*

**lemma** *qSwap-twice-preserves-alpha*:

**assumes**  $qGood\ X \vee qGood\ X'$  **and**  $X \# = X'$

**shows**  $((X \#[[z1 \wedge z2]]-zs) \#[[u1 \wedge u2]]-us) \# = ((X' \#[[z1 \wedge z2]]-zs) \#[[u1 \wedge u2]]-us)$

*<proof>*

**lemma** *alphaAll-trans*:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym)qTerm$  **and**

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

**shows**

$(qGood\ X \longrightarrow (\forall\ X'\ X''.\ X \# = X' \wedge X' \# = X'' \longrightarrow X \# = X'')) \wedge$   
 $(qGoodAbs\ A \longrightarrow (\forall\ A'\ A''.\ A \$ = A' \wedge A' \$ = A'' \longrightarrow A \$ = A''))$

*<proof>*

**corollary** *alpha-trans*:

**assumes**  $qGood\ X \vee qGood\ X' \vee qGood\ X''$   $X \# = X'$   $X' \# = X''$

**shows**  $X \# = X''$

*<proof>*

**corollary** *alphaAbs-trans*:

**assumes**  $qGoodAbs\ A \vee qGoodAbs\ A' \vee qGoodAbs\ A''$

**and**  $A \$ = A'$   $A' \$ = A''$

**shows**  $A \$ = A''$

*<proof>*

**lemma** *alpha-trans-twice*:

$\llbracket qGood\ X \vee qGood\ X' \vee qGood\ X'' \vee qGood\ X''';$

$X \# = X'; X' \# = X''; X'' \# = X''' \rrbracket \Longrightarrow X \# = X'''$

*<proof>*

**lemma** *alphaAbs-trans-twice*:

$\llbracket qGoodAbs\ A \vee qGoodAbs\ A' \vee qGoodAbs\ A'' \vee qGoodAbs\ A''';$

$A \$ = A'; A' \$ = A''; A'' \$ = A''' \rrbracket \Longrightarrow A \$ = A'''$

*<proof>*

**lemma** *qAbs-preserves-alpha*:

**assumes** *ALPHA*:  $X \# = X'$  **and** *GOOD*:  $qGood\ X \vee qGood\ X'$

**shows**  $qAbs\ xs\ x\ X \$ = qAbs\ xs\ x\ X'$

*<proof>*

**corollary** *qAbs-preserves-alpha2*:

**assumes** *ALPHA*:  $X \# = X'$  **and** *GOOD*:  $qGoodAbs(qAbs\ xs\ x\ X) \vee qGoodAbs$   
 $(qAbs\ xs\ x\ X')$

**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**  $qAbs\ ys\ x\ X\ \$ = qAbs\ ys\ x'\ (X\ \#[[x' \wedge x]]-ys)$   
*<proof>*

**lemma** *qAbs-ex-qAFresh-rep*:  
**assumes** *GOOD*:  $qGood\ X$  **and** *FRESH*:  $qAFresh\ xs\ x'\ X$   
**shows**  $\exists\ X'.\ qGood\ X' \wedge qAbs\ xs\ x\ X\ \$ = qAbs\ xs\ x'\ X'$   
*<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**  
 $(qGood\ X \longrightarrow$   
 $((\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qFresh\ zs\ z\ X) \longrightarrow$   
 $(\exists\ X'.\ X\ \#\ =\ X' \wedge (\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qAFresh\ zs\ z\ X')))) \wedge$   
 $(qGoodAbs\ A \longrightarrow$   
 $((\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qFreshAbs\ zs\ z\ A) \longrightarrow$   
 $(\exists\ A'.\ A\ \$ =\ A' \wedge (\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qAFreshAbs\ zs\ z\ A'))))$   
*<proof>*

**corollary** *qFresh-imp-ex-qAFresh*:  
**assumes** *finite V* **and**  $qGood\ X$  **and**  $\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qFresh\ zs\ z\ X$   
**shows**  $\exists\ X'.\ qGood\ X' \wedge X\ \#\ =\ X' \wedge (\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qAFresh\ zs\ z\ X')$   
*<proof>*

**corollary** *qFreshAbs-imp-ex-qAFreshAbs*:  
**assumes** *finite V* **and**  $qGoodAbs\ A$  **and**  $\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qFreshAbs\ zs\ z\ A$   
**shows**  $\exists\ A'.\ qGoodAbs\ A' \wedge A\ \$ =\ A' \wedge (\forall\ z \in V.\ \forall\ zs \in fZs\ z.\ qAFreshAbs\ zs\ z\ A')$   
*<proof>*

**lemma** *qFresh-imp-ex-qAFresh1*:  
**assumes**  $qGood\ X$  **and**  $qFresh\ zs\ z\ X$   
**shows**  $\exists\ X'.\ qGood\ X' \wedge X\ \#\ =\ X' \wedge qAFresh\ zs\ z\ X'$   
*<proof>*

**lemma** *qFreshAbs-imp-ex-qAFreshAbs1*:  
**assumes** *finite V* **and**  $qGoodAbs\ A$  **and**  $qFreshAbs\ zs\ z\ A$   
**shows**  $\exists\ A'.\ qGoodAbs\ A' \wedge A\ \$ =\ A' \wedge qAFreshAbs\ zs\ z\ A'$   
*<proof>*

**lemma** *qFresh-imp-ex-qAFresh2*:  
**assumes**  $qGood\ X$  **and**  $qFresh\ xs\ x\ X$  **and**  $qFresh\ ys\ y\ X$

**shows**  $\exists X'. qGood\ X' \wedge X \# = X' \wedge qAFresh\ xs\ x\ X' \wedge qAFresh\ ys\ y\ X'$   
 ⟨proof⟩

**lemma** *qFreshAbs-imp-ex-qFreshAbs2*:

**assumes** *finite V and qGoodAbs A and qFreshAbs xs x A and qFreshAbs ys y A*  
**shows**  $\exists A'. qGoodAbs\ A' \wedge A \$ = A' \wedge qAFreshAbs\ xs\ x\ A' \wedge qAFreshAbs\ ys\ y\ A'$   
 ⟨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**

$(qGood\ X \longrightarrow$   
 $(qAFresh\ zs\ z\ X \longrightarrow (\forall X'. X \# = X' \longrightarrow qFresh\ zs\ z\ X')) \wedge$   
 $(qGoodAbs\ A \longrightarrow$   
 $(qAFreshAbs\ zs\ z\ A \longrightarrow (\forall A'. A \$ = A' \longrightarrow qFreshAbs\ zs\ z\ A'))))$   
 ⟨proof⟩

**corollary** *qAFresh-qFresh-preserves-alpha*:

$\llbracket qGood\ X; qAFresh\ zs\ z\ X; X \# = X' \rrbracket \Longrightarrow qFresh\ zs\ z\ X'$   
 ⟨proof⟩

**corollary** *qAFreshAbs-imp-qFreshAbs-preserves-alphaAbs*:

$\llbracket qGoodAbs\ A; qAFreshAbs\ zs\ z\ A; A \$ = A' \rrbracket \Longrightarrow qFreshAbs\ zs\ z\ A'$   
 ⟨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**  $qGood\ X \vee qGood\ X'$  **and**  $X \# = X'$

**shows**  $qFresh\ zs\ z\ X \longleftrightarrow qFresh\ zs\ z\ X'$

⟨proof⟩

**lemma** *qFreshAbs-preserves-alphaAbs*:

**assumes**  $qGoodAbs\ A \vee qGoodAbs\ A'$  **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**  $(X \# \llbracket z1 \wedge z2 \rrbracket - zs) \# = X$

*<proof>*

**lemma** *alphaAbs-qFreshAbs-qSwapAbs-id:*

**assumes** *qGoodAbs A and qFreshAbs zs z1 A and qFreshAbs zs z2 A*

**shows**  $(A \text{ \$}[[z1 \wedge z2]]\text{-zs}) \text{ \$} = A$

*<proof>*

**lemma** *alpha-qFresh-qSwap-compose:*

**assumes** *GOOD: qGood X and qFresh zs y X and qFresh zs z X*

**shows**  $((X \text{ \#}[[y \wedge x]]\text{-zs}) \text{ \#}[[z \wedge y]]\text{-zs}) \text{ \#} = (X \text{ \#}[[z \wedge x]]\text{-zs})$

*<proof>*

**lemma** *qAbs-alphaAbs-qSwap-qFresh:*

**assumes** *GOOD: qGood X and FRESH: qFresh xs x' X*

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

*<proof>*

**lemma** *alphaAbs-qAbs-ex-qFresh-rep:*

**assumes** *GOOD: qGood X and FRESH: qFresh xs x' X*

**shows**  $\exists X'. (X, X') \in qSwapped \wedge qGood X' \wedge qAbs \text{ xs } x \text{ X } \$ = qAbs \text{ xs } x' X'$

*<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**

$alphaAbs\text{-ex-equal-or-qAFresh } xs \ x \ X \ xs' \ x' \ X' ==$

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

**definition** *alphaAbs-ex-qAFresh*

**where**

$alphaAbs\text{-ex-qAFresh } xs \ x \ X \ xs' \ x' \ X' ==$

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

**definition** *alphaAbs-ex-distinct-qAFresh*

**where**

$alphaAbs\text{-ex-distinct-qAFresh } xs \ x \ X \ xs' \ x' \ X' ==$

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

**definition** *alphaAbs-all-equal-or-qAFresh*

**where**

$$\begin{aligned} & \text{alphaAbs-all-equal-or-qAFresh } xs \ x \ X \ xs' \ x' \ X' == \\ & (xs = xs' \wedge \\ & (\forall y. (y = x \vee qAFresh \ xs \ y \ X) \wedge (y = x' \vee qAFresh \ xs \ y \ X') \longrightarrow \\ & \quad (X \ #[[y \wedge x]]-xs) \# = (X' \ #[[y \wedge x']]-xs))) \end{aligned}$$

**definition** *alphaAbs-all-qAFresh*

**where**

$$\begin{aligned} & \text{alphaAbs-all-qAFresh } xs \ x \ X \ xs' \ x' \ X' == \\ & (xs = xs' \wedge \\ & (\forall y. qAFresh \ xs \ y \ X \wedge qAFresh \ xs \ y \ X' \longrightarrow \\ & \quad (X \ #[[y \wedge x]]-xs) \# = (X' \ #[[y \wedge x']]-xs))) \end{aligned}$$

**definition** *alphaAbs-all-distinct-qAFresh*

**where**

$$\begin{aligned} & \text{alphaAbs-all-distinct-qAFresh } xs \ x \ X \ xs' \ x' \ X' == \\ & (xs = xs' \wedge \\ & (\forall y. y \notin \{x, x'\} \wedge qAFresh \ xs \ y \ X \wedge qAFresh \ xs \ y \ X' \longrightarrow \\ & \quad (X \ #[[y \wedge x]]-xs) \# = (X' \ #[[y \wedge x']]-xs))) \end{aligned}$$

**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:*

$(qAbs \ xs \ x \ X \ \$ = qAbs \ xs' \ x' \ X') = \text{alphaAbs-ex-distinct-qAFresh } xs \ x \ X \ xs' \ x' \ X'$

*<proof>*

**corollary** *alphaAbs-qAbs-iff-ex-distinct-qAFresh:*

$(qAbs \ xs \ x \ X \ \$ = qAbs \ xs' \ x' \ X') =$

$(xs = xs' \wedge$

$(\exists y. y \notin \{x, x'\} \wedge qAFresh \ xs \ y \ X \wedge qAFresh \ xs \ y \ X' \wedge$



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

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

**assumes** *qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$\alpha Abs\ ex\ equal\ or\ qAFresh\ xs\ x\ X\ xs'\ x'\ X'$

⟨proof⟩

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

**assumes** *qGood X*

**shows**

$(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\exists y. (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X')) \wedge$

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

⟨proof⟩

**lemma** *alphaAbs-qAbs-iff-alphaAbs-ex-qAFresh:*

**assumes** *qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') = \alpha Abs\ ex\ qAFresh\ xs\ x\ X\ xs'\ x'\ X'$

⟨proof⟩

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

**assumes** *qGood X*

**shows**

$(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\exists y. qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \wedge$

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

⟨proof⟩

**lemma** *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qAFresh:*

**assumes** *qGood X* **and**  $qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X'$

**shows**  $\alpha Abs\ all\ equal\ or\ qAFresh\ xs\ x\ X\ xs'\ x'\ X'$

⟨proof⟩

**corollary** *alphaAbs-qAbs-imp-all-equal-or-qAFresh:*

**assumes** *qGood X* **and**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X')$

**shows**

$(xs = xs' \wedge$

$(\forall y. (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X')) \longrightarrow$

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

⟨proof⟩

**lemma** *alphaAbs-qAbs-iff-alphaAbs-all-equal-or-qAFresh:*

**assumes** *qGood X* **and** *qGood X'*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$\alpha Abs\ all\ equal\ or\ qAFresh\ xs\ x\ X\ xs'\ x'\ X'$

*<proof>*

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

**assumes** *qGood X and qGood X'*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\forall y. (y = x \vee qAFresh\ xs\ y\ X) \wedge (y = x' \vee qAFresh\ xs\ y\ X') \longrightarrow$

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

*<proof>*

**lemma** *alphaAbs-qAbs-imp-alphaAbs-all-qAFresh:*

**assumes** *qGood X and qAbs xs x X \$ = qAbs xs' x' X'*

**shows** *alphaAbs-all-qAFresh xs x X xs' x' X'*

*<proof>*

**corollary** *alphaAbs-qAbs-imp-all-qAFresh:*

**assumes** *qGood X and (qAbs xs x X \$ = qAbs xs' x' X')*

**shows**

$(xs = xs' \wedge$

$(\forall y. qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$

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

*<proof>*

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

**assumes** *qGood X and qGood X'*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') = \text{alphaAbs-all-qAFresh}\ xs\ x\ X\ xs'\ x'\ X'$

*<proof>*

**corollary** *alphaAbs-qAbs-iff-all-qAFresh:*

**assumes** *qGood X and qGood X'*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\forall y. qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$

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

*<proof>*

**lemma** *alphaAbs-qAbs-imp-alphaAbs-all-distinct-qAFresh:*

**assumes** *qGood X and qAbs xs x X \$ = qAbs xs' x' X'*

**shows** *alphaAbs-all-distinct-qAFresh xs x X xs' x' X'*

*<proof>*

**corollary** *alphaAbs-qAbs-imp-all-distinct-qAFresh:*

**assumes** *qGood X and (qAbs xs x X \$ = qAbs xs' x' X')*

**shows**

$(xs = xs' \wedge$

$(\forall y. y \notin \{x, x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$

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

*<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**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\forall y. y \notin \{x, x'\} \wedge qAFresh\ xs\ y\ X \wedge qAFresh\ xs\ y\ X' \longrightarrow$   
 $(X \#[[y \wedge x]]-xs) \# = (X' \#[[y \wedge x']]-xs)))$

*<proof>*

## 2.6.2 Then for “qFresh”

**definition** *alphaAbs-ex-equal-or-qFresh*

**where**

$alphaAbs-ex-equal-or-qFresh\ xs\ x\ X\ xs'\ x'\ X' ==$   
 $(xs = xs' \wedge$   
 $(\exists y. (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \wedge$   
 $(X \#[[y \wedge x]]-xs) \# = (X' \#[[y \wedge x']]-xs)))$

**definition** *alphaAbs-ex-qFresh*

**where**

$alphaAbs-ex-qFresh\ xs\ x\ X\ xs'\ x'\ X' ==$   
 $(xs = xs' \wedge$   
 $(\exists y. qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$   
 $(X \#[[y \wedge x]]-xs) \# = (X' \#[[y \wedge x']]-xs)))$

**definition** *alphaAbs-ex-distinct-qFresh*

**where**

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

**definition** *alphaAbs-all-equal-or-qFresh*

**where**

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

**definition** *alphaAbs-all-qFresh*

**where**

$alphaAbs-all-qFresh\ xs\ x\ X\ xs'\ x'\ X' ==$   
 $(xs = xs' \wedge$   
 $(\forall y. qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \longrightarrow$

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

**definition** *alphaAbs-all-distinct-qFresh*

**where**

$$\begin{aligned} & \text{alphaAbs-all-distinct-qFresh } xs \ x \ X \ xs' \ x' \ X' = \\ & (xs = xs' \wedge \\ & (\forall y. y \notin \{x, x'\} \wedge \text{qFresh } xs \ y \ X \wedge \text{qFresh } xs \ y \ X' \longrightarrow \\ & (X \#[[y \wedge x]]-xs) \# = (X' \#[[y \wedge x']] - xs))) \end{aligned}$$

**lemma** *alphaAbs-ex-equal-or-qAFresh-imp-qFresh:*

$$\begin{aligned} & \text{alphaAbs-ex-equal-or-qAFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-ex-equal-or-qFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-ex-distinct-qAFresh-imp-qFresh:*

$$\begin{aligned} & \text{alphaAbs-ex-distinct-qAFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-ex-distinct-qFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-ex-qAFresh-imp-qFresh:*

$$\begin{aligned} & \text{alphaAbs-ex-qAFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-ex-qFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-all-equal-or-qFresh-imp-qAFresh:*

$$\begin{aligned} & \text{alphaAbs-all-equal-or-qFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-all-equal-or-qAFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-all-distinct-qFresh-imp-qAFresh:*

$$\begin{aligned} & \text{alphaAbs-all-distinct-qFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-all-distinct-qAFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-all-qFresh-imp-qAFresh:*

$$\begin{aligned} & \text{alphaAbs-all-qFresh } xs \ x \ X \ xs' \ x' \ X' \Longrightarrow \\ & \text{alphaAbs-all-qAFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-ex-equal-or-qFresh-imp-alphaAbs-qAbs:*

$$\begin{aligned} & \text{assumes } \text{GOOD}: \text{qGood } X \text{ and } \text{alphaAbs-ex-equal-or-qFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \text{shows } \text{qAbs } xs \ x \ X \ \$ = \text{qAbs } xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *alphaAbs-qAbs-iff-alphaAbs-ex-equal-or-qFresh:*

$$\begin{aligned} & \text{assumes } \text{GOOD}: \text{qGood } X \\ & \text{shows } (\text{qAbs } xs \ x \ X \ \$ = \text{qAbs } xs' \ x' \ X') = \\ & \quad \text{alphaAbs-ex-equal-or-qFresh } xs \ x \ X \ xs' \ x' \ X' \\ & \langle \text{proof} \rangle \end{aligned}$$

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

**assumes** *GOOD: qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\exists y. (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \wedge$

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

*<proof>*

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

**assumes** *GOOD: qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$alphaAbs-ex-qFresh\ xs\ x\ X\ xs'\ x'\ X'$

*<proof>*

**corollary** *alphaAbs-qAbs-iff-ex-qFresh:*

**assumes** *GOOD: qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\exists y. qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$

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

*<proof>*

**lemma** *alphaAbs-qAbs-iff-alphaAbs-ex-distinct-qFresh:*

**assumes** *GOOD: qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$alphaAbs-ex-distinct-qFresh\ xs\ x\ X\ xs'\ x'\ X'$

*<proof>*

**corollary** *alphaAbs-qAbs-iff-ex-distinct-qFresh:*

**assumes** *GOOD: qGood X*

**shows**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$

$(xs = xs' \wedge$

$(\exists y. y \notin \{x, x'\} \wedge qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \wedge$

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

*<proof>*

**lemma** *alphaAbs-qAbs-imp-alphaAbs-all-equal-or-qFresh:*

**assumes** *qGood X and qAbs xs x X \$ = qAbs xs' x' X'*

**shows**  $alphaAbs-all-equal-or-qFresh\ xs\ x\ X\ xs'\ x'\ X'$

*<proof>*

**corollary** *alphaAbs-qAbs-imp-all-equal-or-qFresh:*

**assumes** *qGood X and (qAbs xs x X \$ = qAbs xs' x' X')*

**shows**

$(xs = xs' \wedge$

$(\forall y. (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \longrightarrow$

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

*<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**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\forall y. (y = x \vee qFresh\ xs\ y\ X) \wedge (y = x' \vee qFresh\ xs\ y\ X') \longrightarrow$   
 $(X\ \#[[y \wedge x]]-xs) \# = (X'\ \#[[y \wedge x']]-xs)))$

*<proof>*

**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**

$(xs = xs' \wedge$   
 $(\forall y. qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \longrightarrow$   
 $(X\ \#[[y \wedge x]]-xs) \# = (X'\ \#[[y \wedge x']]-xs)))$

*<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**  $(qAbs\ xs\ x\ X\ \$ = qAbs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\forall y. qFresh\ xs\ y\ X \wedge qFresh\ xs\ y\ X' \longrightarrow$   
 $(X\ \#[[y \wedge x]]-xs) \# = (X'\ \#[[y \wedge x']]-xs)))$

*<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 (λX X'. X ≠ X') (rho xs) (rho' xs)}
```

```
abbreviation alphaEnv-abbrev ::
```

```
('index,'bindex,'varSort,'var,'opSym)qEnv =>
  ('index,'bindex,'varSort,'var,'opSym)qEnv => bool (infix &= 50)
```

```
where
```

```
rho &= rho' == (rho,rho') ∈ alphaEnv
```

**definition** *pickQFreshEnv*

**where**

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

**lemma** *qGoodEnv-imp-card-of-qTerm*:

**assumes** *qGoodEnv rho*

**shows**  $|\{X. \exists y. \text{rho } \text{ys } y = \text{Some } X\}| <_o |UNIV :: 'var \text{ set}|$   
<proof>

**lemma** *qGoodEnv-imp-card-of-qTerm2*:

**assumes** *qGoodEnv rho*

**shows**  $|\{X. \exists \text{ys } y. \text{rho } \text{ys } y = \text{Some } X\}| <_o |UNIV :: 'var \text{ set}|$   
<proof>

**lemma** *qGoodEnv-iff*:

*qGoodEnv rho* =  
 $(\forall \text{xs}. \text{liftAll } q\text{Good } (\text{rho } \text{xs})) \wedge$   
 $(\forall \text{ys}. |\{y. \text{rho } \text{ys } y \neq \text{None}\}| <_o |UNIV :: 'var \text{ set}|) \wedge$   
 $|\{X. \exists \text{ys } y. \text{rho } \text{ys } y = \text{Some } X\}| <_o |UNIV :: 'var \text{ set}|$   
<proof>

**lemma** *alphaEnv-refl*:

*qGoodEnv rho*  $\implies$  *rho*  $\&=$  *rho*  
<proof>

**lemma** *alphaEnv-sym*:

*rho*  $\&=$  *rho'*  $\implies$  *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:  $\forall X \in XS. q\text{Good } X$*  **and**

*Rhovar: |Rho| <\_o |UNIV :: 'var set|* **and** *RhoGood:  $\forall \text{rho} \in \text{Rho}. q\text{GoodEnv}$*

*rho*

**shows**

*pickQFreshEnv xs V XS Rho*  $\notin V \wedge$   
 $(\forall X \in XS. q\text{Fresh } xs (pickQFreshEnv xs V XS Rho) X) \wedge$   
 $(\forall \text{rho} \in \text{Rho}. q\text{FreshEnv } xs (pickQFreshEnv xs V XS Rho) \text{rho})$   
<proof>



**lemma** *pickQFreshEnv*:  
**assumes**  $Vvar: |V| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ V$   
**and**  $XSvar: |XS| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ XS$   
**and**  $good: \forall X \in XS. qGood\ X$   
**and**  $Rhovar: |Rho| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ Rho$   
**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>*

**corollary** *obtain-qFreshEnv*:  
**fixes**  $XS::('index,'bindex,'varSort,'var,'opSym)qTerm\ set$  **and**  
 $Rho::('index,'bindex,'varSort,'var,'opSym)qEnv\ set$  **and**  $rho$   
**assumes**  $Vvar: |V| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ V$   
**and**  $XSvar: |XS| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ XS$   
**and**  $good: \forall X \in XS. qGood\ X$   
**and**  $Rhovar: |Rho| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ Rho$   
**and**  $RhoGood: \forall rho \in Rho. qGoodEnv\ rho$   
**shows**  
 $\exists z. z \notin V \wedge$   
 $(\forall X \in XS. qFresh\ xs\ z\ X) \wedge (\forall rho \in Rho. qFreshEnv\ xs\ z\ rho)$   
*<proof>*

### 3.2 Parallel substitution

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

**lemma** *aux-qPsubst-ignoreFirst-qTermLessQSwapped-wf*:  
 $wf(inv-image\ qTermQSwappedLess\ aux-qPsubst-ignoreFirst)$   
*<proof>*

**function**  
 $qPsubst ::$   
 $( 'index,'bindex,'varSort,'var,'opSym)qEnv \Rightarrow ( 'index,'bindex,'varSort,'var,'opSym)qTerm$   
 $\Rightarrow$   
 $( 'index,'bindex,'varSort,'var,'opSym)qTerm$   
**and**  
 $qPsubstAbs ::$

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qEnv \Rightarrow (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$   
 $\Rightarrow$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$   
**where**  
 $qPsubst\ rho\ (qVar\ xs\ x) = (\text{case}\ rho\ xs\ x\ \text{of}\ None \Rightarrow qVar\ xs\ x\ |\ Some\ X \Rightarrow X)$   
 $|$   
 $qPsubst\ rho\ (qOp\ delta\ inp\ binp) =$   
 $qOp\ delta\ (\text{lift}\ (qPsubst\ rho)\ inp)\ (\text{lift}\ (qPsubstAbs\ rho)\ binp)$   
 $|$   
 $qPsubstAbs\ rho\ (qAbs\ xs\ x\ X) =$   
 $(\text{let}\ x' = \text{pickQFreshEnv}\ xs\ \{x\}\ \{X\}\ \{\rho\}\ \text{in}\ qAbs\ xs\ x'\ (qPsubst\ rho\ (X\ \#[[x' \wedge x]]-xs)))$   
 $\langle \text{proof} \rangle$   
**termination**  
 $\langle \text{proof} \rangle$

**abbreviation**  $qPsubst\text{-abbrev} ::$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm \Rightarrow (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qEnv$   
 $\Rightarrow$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm\ (-\ \#[[-]])$   
**where**  $X\ \#[[\rho]] == qPsubst\ rho\ X$

**abbreviation**  $qPsubstAbs\text{-abbrev} ::$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs \Rightarrow (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qEnv$   
 $\Rightarrow$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs\ (-\ \$[[-]])$   
**where**  $A\ \$[[\rho]] == qPsubstAbs\ rho\ A$

**lemma**  $qPsubstAll\text{-preserves-}qGoodAll:$   
**fixes**  $X :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm$  **and**  
 $A :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$  **and**  $\rho$   
**assumes**  $GOOD\text{-ENV}: qGoodEnv\ \rho$   
**shows**  
 $(qGood\ X \longrightarrow qGood\ (X\ \#[[\rho]])) \wedge (qGoodAbs\ A \longrightarrow qGoodAbs\ (A\ \$[[\rho]]))$   
 $\langle \text{proof} \rangle$

**corollary**  $qPsubst\text{-preserves-}qGood:$   
 $\llbracket qGoodEnv\ \rho; qGood\ X \rrbracket \Longrightarrow qGood\ (X\ \#[[\rho]])$   
 $\langle \text{proof} \rangle$

**corollary**  $qPsubstAbs\text{-preserves-}qGoodAbs:$   
 $\llbracket qGoodEnv\ \rho; qGoodAbs\ A \rrbracket \Longrightarrow qGoodAbs\ (A\ \$[[\rho]])$   
 $\langle \text{proof} \rangle$

**lemma**  $qPsubstAll\text{-preserves-}qFreshAll:$   
**fixes**  $X :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qTerm$  **and**  
 $A :: (\text{'index, 'bindex, 'varSort, 'var, 'opSym})qAbs$  **and**  $\rho$   
**assumes**  $GOOD\text{-ENV}: qGoodEnv\ \rho$   
**shows**

$(qFresh\ zs\ z\ X \longrightarrow$   
 $(qGood\ X \wedge qFreshEnv\ zs\ z\ rho \longrightarrow qFresh\ zs\ z\ (X\ \#[[rho]]))) \wedge$   
 $(qFreshAbs\ zs\ z\ A \longrightarrow$   
 $(qGoodAbs\ A \wedge qFreshEnv\ zs\ z\ rho \longrightarrow qFreshAbs\ zs\ z\ (A\ \$[[rho]])))$   
 $\langle proof \rangle$

**lemma**  $qPsubst\text{-}preserves\text{-}qFresh$ :  
 $\llbracket qGood\ X; qGoodEnv\ rho; qFresh\ zs\ z\ X; qFreshEnv\ zs\ z\ rho \rrbracket$   
 $\implies qFresh\ zs\ z\ (X\ \#[[rho]])$   
 $\langle proof \rangle$

**lemma**  $qPsubstAbs\text{-}preserves\text{-}qFreshAbs$ :  
 $\llbracket qGoodAbs\ A; qGoodEnv\ rho; qFreshAbs\ zs\ z\ A; qFreshEnv\ zs\ z\ rho \rrbracket$   
 $\implies qFreshAbs\ zs\ z\ (A\ \$[[rho]])$   
 $\langle proof \rangle$

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**  $qPsubstAll\text{-}preserves\text{-}alphaAll\text{-}qSwapAll$ :  
**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**  
 $(qGood\ X \longrightarrow$   
 $(\forall Y. X\ \# = Y \longrightarrow (X\ \#[[rho]])\ \# = (Y\ \#[[rho]])) \wedge$   
 $(\forall xs\ z1\ z2. qFreshEnv\ xs\ z1\ rho \wedge qFreshEnv\ xs\ z2\ rho \longrightarrow$   
 $((X\ \#[[z1 \wedge z2]]\text{-}xs)\ \#[[rho]])\ \# = ((X\ \#[[rho]])\ \#[[z1 \wedge z2]]\text{-}xs)) \wedge$   
 $(qGoodAbs\ A \longrightarrow$   
 $(\forall B. A\ \$ = B \longrightarrow (A\ \$[[rho]])\ \$ = (B\ \$[[rho]])) \wedge$   
 $(\forall xs\ z1\ z2. qFreshEnv\ xs\ z1\ rho \wedge qFreshEnv\ xs\ z2\ rho \longrightarrow$   
 $((A\ \$[[z1 \wedge z2]]\text{-}xs)\ \$[[rho]])\ \$ = ((A\ \$[[rho]])\ \$[[z1 \wedge z2]]\text{-}xs)))$   
 $\langle proof \rangle$

**corollary**  $qPsubst\text{-}preserves\text{-}alpha1$ :  
**assumes**  $qGoodEnv\ rho$  **and**  $qGood\ X \vee qGood\ Y$  **and**  $X\ \# = Y$   
**shows**  $(X\ \#[[rho]])\ \# = (Y\ \#[[rho]])$   
 $\langle proof \rangle$

**corollary**  $qPsubstAbs\text{-}preserves\text{-}alphaAbs1$ :  
**assumes**  $qGoodEnv\ rho$  **and**  $qGoodAbs\ A \vee qGoodAbs\ B$  **and**  $A\ \$ = B$   
**shows**  $(A\ \$[[rho]])\ \$ = (B\ \$[[rho]])$   
 $\langle proof \rangle$

**corollary**  $alpha\text{-}qFreshEnv\text{-}qSwap\text{-}qPsubst\text{-}commute$ :  
 $\llbracket qGoodEnv\ rho; qGood\ X; qFreshEnv\ zs\ z1\ rho; qFreshEnv\ zs\ z2\ rho \rrbracket \implies$   
 $((X\ \#[[z1 \wedge z2]]\text{-}zs)\ \#[[rho]])\ \# = ((X\ \#[[rho]])\ \#[[z1 \wedge z2]]\text{-}zs)$   
 $\langle proof \rangle$

**corollary** *alphaAbs-qFreshEnv-qSwapAbs-qPsubstAbs-commute:*  
 $\llbracket qGoodEnv\ rho; qGoodAbs\ A; qFreshEnv\ zs\ z1\ rho; qFreshEnv\ zs\ z2\ rho \rrbracket \implies$   
 $((A\ \$\llbracket z1\ \wedge\ z2 \rrbracket -zs)\ \$\llbracket rho \rrbracket) \$ = ((A\ \$\llbracket rho \rrbracket)\ \$\llbracket z1\ \wedge\ z2 \rrbracket -zs)$   
 $\langle proof \rangle$

**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**  $rho'\text{-alpha-}rho''$ :  $rho' \&= rho''$  **and**  
 $goodRho'$ :  $qGoodEnv\ rho'$  **and**  $goodRho''$ :  $qGoodEnv\ rho''$   
**shows**  
 $(qGood\ X \longrightarrow (X\ \#\llbracket rho \rrbracket) \# = (X\ \#\llbracket rho' \rrbracket)) \wedge$   
 $(qGoodAbs\ A \longrightarrow (A\ \$\llbracket rho \rrbracket) \$ = (A\ \$\llbracket rho' \rrbracket))$   
 $\langle proof \rangle$

**corollary** *qPsubst-preserves-alpha2:*  
 $\llbracket qGood\ X; qGoodEnv\ rho'; qGoodEnv\ rho''; rho' \&= rho'' \rrbracket$   
 $\implies (X\ \#\llbracket rho \rrbracket) \# = (X\ \#\llbracket rho' \rrbracket)$   
 $\langle proof \rangle$

**corollary** *qPsubstAbs-preserves-alphaAbs2:*  
 $\llbracket qGoodAbs\ A; qGoodEnv\ rho'; qGoodEnv\ rho''; rho' \&= rho'' \rrbracket$   
 $\implies (A\ \$\llbracket rho \rrbracket) \$ = (A\ \$\llbracket rho' \rrbracket)$   
 $\langle proof \rangle$

**lemma** *qPsubst-preserves-alpha:*  
**assumes**  $qGood\ X \vee qGood\ X'$  **and**  $qGoodEnv\ rho$  **and**  $qGoodEnv\ rho'$   
**and**  $X \# = X'$  **and**  $rho \&= rho'$   
**shows**  $(X\ \#\llbracket rho \rrbracket) \# = (X'\ \#\llbracket rho \rrbracket)$   
 $\langle proof \rangle$

**lemma** *qPsubstAbs-preserves-alphaAbs:*  
**assumes**  $qGoodAbs\ A \vee qGoodAbs\ A'$  **and**  $qGoodEnv\ rho$  **and**  $qGoodEnv\ rho'$   
**and**  $A \$ = A'$  **and**  $rho \&= rho'$   
**shows**  $(A\ \$\llbracket rho \rrbracket) \$ = (A'\ \$\llbracket rho \rrbracket)$   
 $\langle proof \rangle$

**lemma** *qFresh-qPsubst-commute-qAbs:*  
**assumes**  $good\text{-}X$ :  $qGood\ X$  **and**  $good\text{-}rho$ :  $qGoodEnv\ rho$  **and**  
 $x\text{-fresh-}rho$ :  $qFreshEnv\ xs\ x\ rho$   
**shows**  $((qAbs\ xs\ x\ X)\ \$\llbracket rho \rrbracket) \$ = qAbs\ xs\ x\ (X\ \#\llbracket rho \rrbracket)$   
 $\langle proof \rangle$

**end**

**end**

**theory** *Pick imports Main*

**begin**

**definition** *pick*  $X \equiv \text{SOME } x. x \in X$

**lemma** *pick[simp]*:  $x \in X \implies \text{pick } X \in X$   
*<proof>*

**lemma** *pick-NE[simp]*:  $X \neq \{\}$   $\implies \text{pick } X \in X$  *<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]*:  $S2P\ (\text{Collect } \varphi) = \varphi$   
*<proof>*

**lemma** *Collect-S2P[simp]*:  $\text{Collect } (S2P\ r) = r$   
*<proof>*

Binary predicates vs. relations:

**definition** *P2R*  $\varphi \equiv \{(x,y). \varphi\ x\ y\}$

**definition** *R2P*  $r \equiv \lambda x\ y. (x,y) \in r$

**lemma** *in-P2R[simp]*:  $xy \in P2R\ \varphi \longleftrightarrow \varphi\ (\text{fst } xy)\ (\text{snd } xy)$   
*<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]*:  $R2P\ (P2R\ \varphi) = \varphi$   
*<proof>*

**lemma** *P2R-R2P[simp]*:  $P2R\ (R2P\ r) = r$   
*<proof>*

**definition**  $\text{reflP } P \ \varphi \equiv (\forall x \ y. \ \varphi \ x \ y \vee \varphi \ y \ x \longrightarrow P \ x) \wedge (\forall x. \ P \ x \longrightarrow \varphi \ x \ x)$

**definition**  $\text{symP } \varphi \equiv \forall x \ y. \ \varphi \ x \ y \longrightarrow \varphi \ y \ x$

**definition**  $\text{transP}$  **where**  $\text{transP } \varphi \equiv \forall x \ y \ z. \ \varphi \ x \ y \wedge \varphi \ y \ z \longrightarrow \varphi \ x \ z$

**definition**  $\text{equivP } A \ \varphi \equiv \text{reflP } A \ \varphi \wedge \text{symP } \varphi \wedge \text{transP } \varphi$

**lemma**  $\text{refl-on-P2R[simp]}$ :  $\text{refl-on } (\text{Collect } P) (P2R \ \varphi) \longleftrightarrow \text{reflP } P \ \varphi$   
*<proof>*

**lemma**  $\text{reflP-R2P[simp]}$ :  $\text{reflP } (S2P \ A) (R2P \ r) \longleftrightarrow \text{refl-on } A \ r$   
*<proof>*

**lemma**  $\text{sym-P2R[simp]}$ :  $\text{sym } (P2R \ \varphi) \longleftrightarrow \text{symP } \varphi$   
*<proof>*

**lemma**  $\text{symP-R2P[simp]}$ :  $\text{symP } (R2P \ r) \longleftrightarrow \text{sym } r$   
*<proof>*

**lemma**  $\text{trans-P2R[simp]}$ :  $\text{trans } (P2R \ \varphi) \longleftrightarrow \text{transP } \varphi$   
*<proof>*

**lemma**  $\text{transP-R2P[simp]}$ :  $\text{transP } (R2P \ r) \longleftrightarrow \text{trans } r$   
*<proof>*

**lemma**  $\text{equiv-P2R[simp]}$ :  $\text{equiv } (\text{Collect } P) (P2R \ \varphi) \longleftrightarrow \text{equivP } P \ \varphi$   
*<proof>*

**lemma**  $\text{equivP-R2P[simp]}$ :  $\text{equivP } (S2P \ A) (R2P \ r) \longleftrightarrow \text{equiv } A \ r$   
*<proof>*

**lemma**  $\text{in-P2R-Im-singl[simp]}$ :  $y \in P2R \ \varphi \ \text{“} \{x\} \longleftrightarrow \varphi \ x \ y$  *<proof>*

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

**lemma**  $\text{proj-P2R}$ :  $\text{proj } \varphi \ x = P2R \ \varphi \ \text{“} \{x\}$  *<proof>*

**lemma**  $\text{proj-P2R-raw}$ :  $\text{proj } \varphi = (\lambda x. \ P2R \ \varphi \ \text{“} \{x\})$   
*<proof>*

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

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

**lemma**  $\text{proj-preserves}$ :  
 $P \ x \Longrightarrow (P \ /'/' \ \varphi) (\text{proj } \varphi \ x)$

$\langle proof \rangle$

**lemma** *proj-in-iff*:

**assumes** *equivP*  $P \ \varphi$

**shows**  $(P//\varphi) \ (proj \ \varphi \ x) \ \longleftrightarrow \ P \ x$

$\langle proof \rangle$

**lemma** *proj-iff[simp]*:

$\llbracket equivP \ P \ \varphi; \ P \ x; \ P \ y \rrbracket \Longrightarrow \ proj \ \varphi \ x = proj \ \varphi \ y \longleftrightarrow \ \varphi \ x \ y$

$\langle proof \rangle$

**lemma** *in-proj[simp]*:  $\llbracket equivP \ P \ \varphi; \ P \ x \rrbracket \Longrightarrow \ x \in proj \ \varphi \ x$

$\langle proof \rangle$

**lemma** *proj-image[simp]*:  $(proj \ \varphi) \ ` \ (Collect \ P) = Collect \ (P//\varphi)$

$\langle proof \rangle$

**lemma** *in-quotientP-imp-non-empty*:

**assumes** *equivP*  $P \ \varphi$  **and**  $(P//\varphi) \ X$

**shows**  $X \neq \{\}$

$\langle proof \rangle$

**lemma** *in-quotientP-imp-in-rel*:

$\llbracket equivP \ P \ \varphi; \ (P//\varphi) \ X; \ x \in X; \ y \in X \rrbracket \Longrightarrow \ \varphi \ x \ y$

$\langle proof \rangle$

**lemma** *in-quotientP-imp-closed*:

$\llbracket equivP \ P \ \varphi; \ (P//\varphi) \ X; \ x \in X; \ \varphi \ x \ y \rrbracket \Longrightarrow \ y \in X$

$\langle proof \rangle$

**lemma** *in-quotientP-imp-subset*:

**assumes** *equivP*  $P \ \varphi$  **and**  $(P//\varphi) \ X$

**shows**  $X \subseteq Collect \ P$

$\langle proof \rangle$

**lemma** *equivP-pick-in*:

**assumes** *equivP*  $P \ \varphi$  **and**  $(P//\varphi) \ X$

**shows** *pick*  $X \in X$

$\langle proof \rangle$

**lemma** *equivP-pick-preserves*:

**assumes** *equivP*  $P \ \varphi$  **and**  $(P//\varphi) \ X$

**shows**  $P \ (pick \ X)$

$\langle proof \rangle$

**lemma** *proj-pick*:

**assumes**  $\varphi: equivP \ P \ \varphi$  **and**  $X: (P//\varphi) \ X$

**shows**  $proj \ \varphi \ (pick \ X) = X$

$\langle proof \rangle$

**lemma** *pick-proj*:  
**assumes** *equivP P φ* **and** *P x*  
**shows**  $\varphi$  (*pick* (*proj φ x*)) *x*  
*<proof>*

**lemma** *equivP-pick-iff[simp]*:  
**assumes**  $\varphi$ : *equivP P φ* **and** *X: (P///φ) X* **and** *Y: (P///φ) Y*  
**shows**  $\varphi$  (*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*  $\equiv \forall x y. \varphi x y \longrightarrow f x = f y$

**abbreviation** *RESPECTS-P* (**infix** *respectsP 80*) **where**  
*f respectsP r*  $\equiv$  *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**  $\bigwedge x. P x \implies G$  (*proj φ x*) = *f x*  
**shows**  $\forall X. (P///\varphi) X \longrightarrow G X = \text{univ } f X$   
*<proof>*

**lemma** *univ-preserves*:  
**assumes** *equivP P φ* **and** *f respectsP φ* **and**  $\bigwedge x. P x \implies f x \in B$   
**shows**  $\forall X. (P///\varphi) X \longrightarrow \text{univ } f X \in B$   
*<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-inputs.

**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* (- % $[[ - \wedge - ]]$ '-- 200) **where**  
 $(qinp\ \%[[z1\ \wedge\ z2]]-zs) == qSwapInp\ zs\ z1\ z2\ qinp$

**abbreviation** *qSwapBinp-abbrev* (- % $[[ - \wedge - ]]$ '-- 200) **where**  
 $(qbinp\ \%[[z1\ \wedge\ z2]]-zs) == qSwapBinp\ zs\ z1\ z2\ qbinp$

**lemma** *qSwap-qSwapInp*:  
 $((qOp\ delta\ qinp\ qbinp)\ \#[[x\ \wedge\ y]]-xs) =$   
 $qOp\ delta\ (qinp\ \%[[x\ \wedge\ y]]-xs)\ (qbinp\ \%[[x\ \wedge\ y]]-xs)$   
(*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**  $qPsubst-qPsubstInp$ :  
 $((qOp \ delta \ qinp \ qbinp) \ #[[rho]]) = qOp \ delta \ (qinp \ \%[[rho]]) \ (qbinp \ \%[[rho]])$   
 $\langle proof \rangle$

**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**  $qSkel-qSkelInp$ :  
 $qSkel \ (qOp \ delta \ qinp \ qbinp) =$   
 $Branch \ (qSkelInp \ qinp) \ (qSkelBinp \ qbinp)$   
 $\langle proof \rangle$

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

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

**definition**  $qFreshInp$  ::  
 $'varSort \Rightarrow 'var \Rightarrow ('index, ('index, 'bindex, 'varSort, 'var, 'opSym) \ qTerm) \ input \Rightarrow$

*bool*  
**where**  
 $qFreshInp\ xs\ x\ qinp == liftAll\ (qFresh\ xs\ x)\ qinp$

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

**lemma**  $qFresh-qFreshInp:$   
 $qFresh\ xs\ x\ (qOp\ delta\ qinp\ qbinp) =$   
 $(qFreshInp\ xs\ x\ qinp \wedge qFreshBinp\ xs\ x\ qbinp)$   
 $\langle proof \rangle$

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

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

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

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

**lemma**  $qGood-qGoodInp:$   
 $qGood\ (qOp\ delta\ qinp\ qbinp) = (qGoodInp\ qinp \wedge qGoodBinp\ qbinp)$   
 $\langle proof \rangle$

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

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

**definition**  $alphaInp$  **where**  
 $alphaInp ==$   
 $\{(qinp, qinp').\ sameDom\ qinp\ qinp' \wedge liftAll2\ (\lambda qX\ qX'.\ qX \# = qX')\ qinp\ qinp'\}$

**definition** *alphaBinp* where

*alphaBinp* ==  
{(*qbinp*,*qbinp'*). *sameDom* *qbinp* *qbinp'* ∧ *liftAll2* (λ*qA* *qA'*. *qA* \$= *qA'*) *qbinp* *qbinp'*}

**abbreviation** *alphaInp-abbrev* (infix %= 50) where

*qinp* %= *qinp'* == (*qinp*,*qinp'*) ∈ *alphaInp*

**abbreviation** *alphaBinp-abbrev* (infix %%= 50) where

*qbinp* %%= *qbinp'* == (*qbinp*,*qbinp'*) ∈ *alphaBinp*

**lemma** *alpha-alphaInp*:

(*qOp* *delta* *qinp* *qbinp* #= *qOp* *delta'* *qinp'* *qbinp'*) =  
(*delta* = *delta'* ∧ *qinp* %= *qinp'* ∧ *qbinp* %%= *qbinp'*)  
{*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  $\wedge$  x2]]-xs)*

*<proof>*

**lemma** *qSwapBinp-preserves-qGoodBinp:*

**assumes** *qGoodBinp qbinp*

**shows** *qGoodBinp (qbinp %[[x1  $\wedge$  x2]]-xs)*

*<proof>*

**lemma** *qSwapInp-preserves-alphaInp:*

**assumes** *qGoodInp qinp  $\vee$  qGoodInp qinp' and qinp % = qinp'*

**shows** *(qinp %[[x1  $\wedge$  x2]]-xs) % = (qinp' %[[x1  $\wedge$  x2]]-xs)*

*<proof>*

**lemma** *qSwapBinp-preserves-alphaBinp:*

**assumes** *qGoodBinp qbinp  $\vee$  qGoodBinp qbinp' and qbinp % = qbinp'*

**shows** *(qbinp %[[x1  $\wedge$  x2]]-xs) % = (qbinp' %[[x1  $\wedge$  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  $\vee$  qGoodInp qinp' and qGoodEnv qrho and qinp % = qinp'*

**shows** *(qinp %[[qrho]]) % = (qinp' %[[qrho]])*

*<proof>*

**lemma** *qPsubstBinp-preserves-alphaBinp:*

**assumes** *qGoodBinp qbinp  $\vee$  qGoodBinp qbinp' and qGoodEnv qrho and qbinp % = qbinp'*

**shows** *(qbinp %[[qrho]]) % = (qbinp' %[[qrho]])*

*<proof>*

**lemma** *qFreshInp-preserves-alphaInp-aux*:  
**assumes** *good*:  $qGoodInp\ qinp \vee qGoodInp\ qinp'$  **and** *alpha*:  $qinp \approx qinp'$   
**and** *fresh*:  $qFreshInp\ xs\ x\ qinp$   
**shows**  $qFreshInp\ xs\ x\ qinp'$   
*<proof>*

**lemma** *qFreshBinp-preserves-alphaBinp-aux*:  
**assumes** *good*:  $qGoodBinp\ qbinp \vee qGoodBinp\ qbinp'$  **and** *alpha*:  $qbinp \approx\approx qbinp'$   
**and** *fresh*:  $qFreshBinp\ xs\ x\ qbinp$   
**shows**  $qFreshBinp\ xs\ x\ qbinp'$   
*<proof>*

**lemma** *qFreshInp-preserves-alphaInp*:  
**assumes**  $qGoodInp\ qinp \vee qGoodInp\ qinp'$  **and**  $qinp \approx qinp'$   
**shows**  $qFreshInp\ xs\ x\ qinp \longleftrightarrow qFreshInp\ xs\ x\ qinp'$   
*<proof>*

**lemma** *qFreshBinp-preserves-alphaBinp*:  
**assumes**  $qGoodBinp\ qbinp \vee qGoodBinp\ qbinp'$  **and**  $qbinp \approx\approx qbinp'$   
**shows**  $qFreshBinp\ xs\ x\ qbinp \longleftrightarrow qFreshBinp\ xs\ x\ qbinp'$   
*<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  $\Rightarrow$  'var  $\Rightarrow$  ('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*

$(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{term list}$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{abs list}$   
 $(\text{'index, 'bindex, 'varSort, 'var, 'opSym})\text{env list}$

**fun varsOf where**  
 $\text{varsOf (Par } xL \text{ - - )} = \text{set } xL$

**fun termsOf where**  
 $\text{termsOf (Par - } XL \text{ - )} = \text{set } XL$

**fun absOf where**  
 $\text{absOf (Par - - } AL \text{ - )} = \text{set } AL$

**fun envsOf where**  
 $\text{envsOf (Par - - - } rhoL \text{ )} = \text{set } rhoL$

**context FixVars**  
**begin**

**definition**  $\text{alphaGood} \equiv \lambda qX qY. qGood qX \wedge qGood qY \wedge qX \# = qY$

**definition**  $\text{alphaAbsGood} \equiv \lambda qA qB. qGoodAbs qA \wedge qGoodAbs qB \wedge qA \$ = qB$

**definition**  $\text{good} \equiv qGood \text{ /// } \text{alphaGood}$

**definition**  $\text{goodAbs} \equiv qGoodAbs \text{ /// } \text{alphaAbsGood}$

**definition goodInp where**  
 $\text{goodInp inp} ==$   
 $\text{liftAll good inp} \wedge$   
 $|\{i. \text{inp } i \neq \text{None}\}| < o \text{ |UNIV :: 'var set|}$

**definition goodBinp where**  
 $\text{goodBinp binp} ==$   
 $\text{liftAll goodAbs binp} \wedge$   
 $|\{i. \text{binp } i \neq \text{None}\}| < o \text{ |UNIV :: 'var set|}$

**definition goodEnv where**  
 $\text{goodEnv rho} ==$   
 $(\forall \text{ys. } \text{liftAll good (rho ys)}) \wedge$   
 $(\forall \text{ys. } |\{y. \text{rho ys } y \neq \text{None}\}| < o \text{ |UNIV :: 'var set|})$

**definition asTerm where**  
 $\text{asTerm } qX \equiv \text{proj } \text{alphaGood } qX$

**definition asAbs where**  
 $\text{asAbs } qA \equiv \text{proj } \text{alphaAbsGood } qA$

**definition pickInp where**  
 $\text{pickInp inp} \equiv \text{lift pick inp}$

**definition** *pickBinp* **where**  
*pickBinp binp*  $\equiv$  *lift pick binp*

**definition** *asInp* **where**  
*asInp qinp*  $\equiv$  *lift asTerm qinp*

**definition** *asBinp* **where**  
*asBinp qbinp*  $\equiv$  *lift asAbs qbinp*

**definition** *pickE* **where**  
*pickE rho*  $\equiv$   $\lambda$  *xs*. *lift pick (rho xs)*

**definition** *asEnv* **where**  
*asEnv qrho*  $\equiv$   $\lambda$  *xs*. *lift asTerm (qrho xs)*

**definition** *Var* **where**  
*Var xs x*  $\equiv$  *asTerm (qVar xs x)*

**definition** *Op* **where**  
*Op delta inp binp*  $\equiv$  *asTerm (qOp delta (pickInp inp) (pickBinp binp))*

**definition** *Abs* **where**  
*Abs xs x X*  $\equiv$  *asAbs (qAbs xs x (pick X))*

**definition** *skel* **where**  
*skel X*  $\equiv$  *qSkel (pick X)*

**definition** *skelAbs* **where**  
*skelAbs A*  $\equiv$  *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*  $(- \#[- \wedge -]'-- 200)$  **where**

$(X \#[z1 \wedge z2]-zs) \equiv swap\ zs\ z1\ z2\ X$

**definition** *swapAbs* **where**

$swapAbs\ xs\ x\ y\ A = asAbs\ (qSwapAbs\ xs\ x\ y\ (pick\ A))$

**abbreviation** *swapAbs-abbrev*  $(- \$[- \wedge -]'-- 200)$  **where**

$(A \$[z1 \wedge z2]-zs) \equiv swapAbs\ zs\ z1\ z2\ A$

**definition** *swapInp* **where**

$swapInp\ xs\ x\ y\ inp \equiv lift\ (swap\ xs\ x\ y)\ inp$

**definition** *swapBinp* **where**

$swapBinp\ xs\ x\ y\ binp \equiv lift\ (swapAbs\ xs\ x\ y)\ binp$

**abbreviation** *swapInp-abbrev*  $(- \%[- \wedge -]'-- 200)$  **where**

$(inp\ \%[z1 \wedge z2]-zs) \equiv swapInp\ zs\ z1\ z2\ inp$

**abbreviation** *swapBinp-abbrev*  $(- \%%[- \wedge -]'-- 200)$  **where**

$(binp\ \%%[z1 \wedge z2]-zs) \equiv swapBinp\ zs\ z1\ z2\ binp$

**definition** *swapEnvDom* **where**

$swapEnvDom\ xs\ x\ y\ rho \equiv \lambda zs\ z.\ rho\ zs\ (z\ @zs[x \wedge y]-xs)$

**definition** *swapEnvIm* **where**

$swapEnvIm\ xs\ x\ y\ rho \equiv \lambda zs.\ lift\ (swap\ xs\ x\ y)\ (rho\ zs)$

**definition** *swapEnv* **where**

$swapEnv\ xs\ x\ y \equiv swapEnvIm\ xs\ x\ y\ o\ swapEnvDom\ xs\ x\ y$

**abbreviation** *swapEnv-abbrev*  $(- \&[- \wedge -]'-- 200)$  **where**

$(rho\ \&[z1 \wedge z2]-zs) \equiv swapEnv\ zs\ z1\ z2\ rho$

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

**inductive-set** *swapped* **where**

*Refl*:  $(X, X) \in swapped$

|

*Trans*:  $\llbracket (X, Y) \in swapped; (Y, Z) \in swapped \rrbracket \implies (X, Z) \in swapped$

|

*Swap*:  $(X, Y) \in swapped \implies (X, Y \#[x \wedge y]-zs) \in swapped$

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

**definition** *fresh* **where**

$fresh\ xs\ x\ X \equiv qFresh\ xs\ x\ (pick\ X)$

**definition** *freshAbs* **where**  
 $freshAbs\ xs\ x\ A \equiv qFreshAbs\ xs\ x\ (pick\ A)$

**definition** *freshInp* **where**  
 $freshInp\ xs\ x\ inp \equiv liftAll\ (fresh\ xs\ x)\ inp$

**definition** *freshBinp* **where**  
 $freshBinp\ xs\ x\ binp \equiv liftAll\ (freshAbs\ xs\ x)\ binp$

**definition** *freshEnv* **where**  
 $freshEnv\ xs\ x\ rho ==$   
 $rho\ xs\ x = None \wedge (\forall\ ys.\ liftAll\ (fresh\ xs\ x)\ (rho\ ys))$

**definition** *psubst* **where**  
 $psubst\ rho\ X \equiv asTerm(qPsubst\ (pickE\ rho)\ (pick\ X))$

**abbreviation** *psubst-abbrev*  $(- \#[-])$  **where**  
 $(X \#[rho]) \equiv psubst\ rho\ X$

**definition** *psubstAbs* **where**  
 $psubstAbs\ rho\ A \equiv asAbs(qPsubstAbs\ (pickE\ rho)\ (pick\ A))$

**abbreviation** *psubstAbs-abbrev*  $(- \$[-])$  **where**  
 $A \$[rho] \equiv psubstAbs\ rho\ A$

**definition** *psubstInp* **where**  
 $psubstInp\ rho\ inp \equiv lift\ (psubst\ rho)\ inp$

**definition** *psubstBinp* **where**  
 $psubstBinp\ rho\ binp \equiv lift\ (psubstAbs\ rho)\ binp$

**abbreviation** *psubstInp-abbrev*  $(- \%[-])$  **where**  
 $inp \%[rho] \equiv psubstInp\ rho\ inp$

**abbreviation** *psubstBinp-abbrev*  $(- \%\%[-])$  **where**  
 $binp \%\%[rho] \equiv psubstBinp\ rho\ binp$

**definition** *psubstEnv* **where**  
 $psubstEnv\ rho\ rho' \equiv$   
 $\lambda\ xs\ x.\ case\ rho'\ xs\ x\ of\ None \Rightarrow rho\ xs\ x$   
 $\quad\quad\quad | Some\ X \Rightarrow Some\ (X \#[rho])$

**abbreviation** *psubstEnv-abbrev*  $(- \&[-])$  **where**  
 $rho \&[rho'] \equiv psubstEnv\ rho'\ rho$

**definition** *idEnv* **where**  
 $idEnv \equiv \lambda xs.\ Map.empty$

**definition** *updEnv* ::  
 ('index,'bindex,'varSort,'var,'opSym)env  $\Rightarrow$   
 'var  $\Rightarrow$  ('index,'bindex,'varSort,'var,'opSym)term  $\Rightarrow$  'varSort  $\Rightarrow$   
 ('index,'bindex,'varSort,'var,'opSym)env  
 (- [-  $\leftarrow$  -]'-- 200) **where**  
 (rho [x  $\leftarrow$  X]-xs)  $\equiv$   $\lambda$  ys y. (if ys = xs  $\wedge$  y = x then Some X else rho ys y)

(Unary) substitution:

**definition** *subst* **where**  
*subst* xs X x  $\equiv$  *psubst* (idEnv [x  $\leftarrow$  X]-xs)

**abbreviation** *subst-abbrev* (- #[- '/' -]'-- 200) **where**  
 (Y #[X / x]-xs)  $\equiv$  *subst* xs X x Y

**definition** *substAbs* **where**  
*substAbs* xs X x  $\equiv$  *psubstAbs* (idEnv [x  $\leftarrow$  X]-xs)

**abbreviation** *substAbs-abbrev* (- \$[- '/' -]'-- 200) **where**  
 (A \$[X / x]-xs)  $\equiv$  *substAbs* xs X x A

**definition** *substInp* **where**  
*substInp* xs X x  $\equiv$  *psubstInp* (idEnv [x  $\leftarrow$  X]-xs)

**definition** *substBinp* **where**  
*substBinp* xs X x  $\equiv$  *psubstBinp* (idEnv [x  $\leftarrow$  X]-xs)

**abbreviation** *substInp-abbrev* (- %[- '/' -]'-- 200) **where**  
 (inp %[X / x]-xs)  $\equiv$  *substInp* xs X x inp

**abbreviation** *substBinp-abbrev* (- %%[- '/' -]'-- 200) **where**  
 (binp %%[X / x]-xs)  $\equiv$  *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  $\equiv$  *psubstEnv* (idEnv [x  $\leftarrow$  X]-xs)

**abbreviation** *substEnv-abbrev* (- &[- '/' -]'-- 200) **where**  
 (Y &[X / x]-xs)  $\equiv$  *substEnv* xs X x Y

**theorem** *substEnv-def2*:  
 (rho &[Y / y]-ys) =  
 ( $\lambda$ xs x. case rho xs x of

$None \Rightarrow \text{if } (xs = ys \wedge x = y) \text{ then } Some\ Y \text{ else } None$   
 $|Some\ X \Rightarrow Some\ (X \#[Y / y]-ys)$

$\langle proof \rangle$

Variable-for-variable substitution:

**definition** *vsubst* **where**

$vsubst\ ys\ y1\ y2 \equiv subst\ ys\ (Var\ ys\ y1)\ y2$

**abbreviation** *vsubst-abbrev* ( $- \#[- \ '/' / -]'- 200$ ) **where**

$(X \#[y1 / / y2]-ys) \equiv vsubst\ ys\ y1\ y2\ X$

**definition** *vsubstAbs* **where**

$vsubstAbs\ ys\ y1\ y2 \equiv substAbs\ ys\ (Var\ ys\ y1)\ y2$

**abbreviation** *vsubstAbs-abbrev* ( $- \$[- \ '/' / -]'- 200$ ) **where**

$(A \$[y1 / / y2]-ys) \equiv vsubstAbs\ ys\ y1\ y2\ A$

**definition** *vsubstInp* **where**

$vsubstInp\ ys\ y1\ y2 \equiv substInp\ ys\ (Var\ ys\ y1)\ y2$

**definition** *vsubstBinp* **where**

$vsubstBinp\ ys\ y1\ y2 \equiv substBinp\ ys\ (Var\ ys\ y1)\ y2$

**abbreviation** *vsubstInp-abbrev* ( $- \%[- \ '/' / -]'- 200$ ) **where**

$(inp\ \%[y1 / / y2]-ys) \equiv vsubstInp\ ys\ y1\ y2\ inp$

**abbreviation** *vsubstBinp-abbrev* ( $- \% \%[- \ '/' / -]'- 200$ ) **where**

$(binp\ \% \%[y1 / / y2]-ys) \equiv vsubstBinp\ ys\ y1\ y2\ binp$

**lemma** *vsubstInp-def2*:

$(inp\ \%[y1 / / y2]-ys) = lift\ (vsubst\ ys\ y1\ y2)\ inp$

$\langle proof \rangle$

**lemma** *vsubstBinp-def2*:

$(binp\ \% \%[y1 / / y2]-ys) = lift\ (vsubstAbs\ ys\ y1\ y2)\ binp$

$\langle proof \rangle$

**definition** *vsubstEnv* **where**

$vsubstEnv\ ys\ y1\ y2 \equiv substEnv\ ys\ (Var\ ys\ y1)\ y2$

**abbreviation** *vsubstEnv-abbrev* ( $- \&[- \ '/' / -]'- 200$ ) **where**

$(rho\ \&[y1 / / y2]-ys) \equiv vsubstEnv\ ys\ y1\ y2\ rho$

**theorem** *vsubstEnv-def2*:

$(rho\ \&[y1 / / y]-ys) =$

$(\lambda xs\ x.\ case\ rho\ xs\ x\ of$

$None \Rightarrow \text{if } (xs = ys \wedge x = y) \text{ then } Some\ (Var\ ys\ y1) \text{ else } None$

$|Some\ X \Rightarrow Some\ (X \#[y1 / / y]-ys))$

$\langle proof \rangle$

**definition** *goodPar* **where**  
 $goodPar\ P \equiv (\forall X \in termsOf\ P. good\ X) \wedge$   
 $(\forall A \in absOf\ P. goodAbs\ A) \wedge$   
 $(\forall rho \in envsOf\ P. goodEnv\ rho)$

**lemma** *Par-preserves-good[simp]*:  
**assumes** !!  $X. X \in set\ XL \implies good\ X$   
**and** !!  $A. A \in set\ AL \implies goodAbs\ A$   
**and** !!  $rho. rho \in set\ rhoL \implies goodEnv\ rho$   
**shows**  $goodPar\ (Par\ xL\ XL\ AL\ rhoL)$   
 $\langle proof \rangle$

**lemma** *termsOf-preserves-good[simp]*:  
**assumes**  $goodPar\ P$  **and**  $X : termsOf\ P$   
**shows**  $good\ X$   
 $\langle proof \rangle$

**lemma** *absOf-preserves-good[simp]*:  
**assumes**  $goodPar\ P$  **and**  $A : absOf\ P$   
**shows**  $goodAbs\ A$   
 $\langle proof \rangle$

**lemma** *envsOf-preserves-good[simp]*:  
**assumes**  $goodPar\ P$  **and**  $rho : envsOf\ P$   
**shows**  $goodEnv\ rho$   
 $\langle proof \rangle$

**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*:  $equivP\ qGood\ alphaGood$   
 $\langle proof \rangle$

**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**  $pick-injective-good[simp]$ :  
 $\llbracket good\ X; good\ Y \rrbracket \implies (pick\ X = pick\ Y) = (X = Y)$   
*<proof>*

**lemma**  $good-imp-qGood-pick$ :  
 $good\ X \implies qGood\ (pick\ X)$   
*<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**  $pick-alpha$ :  $good\ X \implies pick\ X \# = pick\ X$   
*<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**  $qGood\ qX \vee qGood\ qY$   
**shows**  $(asTerm\ qX = asTerm\ qY) = (qX \# = qY)$

*<proof>*

**lemma** *pick-alpha-iff-equal*:  
**assumes** *good X and good Y*  
**shows**  $\text{pick } X \# = \text{pick } Y \longleftrightarrow X = Y$   
*<proof>*

**lemma** *pick-swap-qSwap*:  
**assumes** *good X*  
**shows**  $\text{pick } (X \#[x1 \wedge x2]-xs) \# = ((\text{pick } X) \#[[x1 \wedge x2]]-xs)$   
*<proof>*

**lemma** *asTerm-qSwap-swap*:  
**assumes** *qGood qX*  
**shows**  $\text{asTerm } (qX \#[[x1 \wedge x2]]-xs) = ((\text{asTerm } qX) \#[x1 \wedge x2]-xs)$   
*<proof>*

**lemma** *fresh-asTerm-qFresh*:  
**assumes** *qGood qX*  
**shows**  $\text{fresh } xs \ x \ (\text{asTerm } qX) = \text{qFresh } xs \ x \ qX$   
*<proof>*

**lemma** *skel-asTerm-qSkel*:  
**assumes** *qGood qX*  
**shows**  $\text{skel } (\text{asTerm } qX) = \text{qSkel } qX$   
*<proof>*

**lemma** *double-swap-qSwap*:  
**assumes** *good X*  
**shows**  $qGood \ ((\text{pick } X) \#[[x \wedge y]]-zs) \#[[x' \wedge y']]-zs' \wedge$   
 $((X \#[x \wedge y]-zs) \#[x' \wedge y']-zs') = \text{asTerm } ((\text{pick } X) \#[[x \wedge y]]-zs) \#[[x' \wedge$   
 $y']] -zs'$   
*<proof>*

**lemma** *fresh-swap-qFresh-qSwap*:  
**assumes** *good X*  
**shows**  $\text{fresh } xs \ x \ (X \#[y1 \wedge y2]-ys) = \text{qFresh } xs \ x \ ((\text{pick } X) \#[[y1 \wedge y2]]-ys)$   
*<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**  $\text{univ } fAbs \ (\text{asAbs } A) = fAbs \ A$

*<proof>*

**corollary** *univ-asAbs-alphaAbs[simp]*:

**assumes** \*: *fAbs respects P 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]*:

$\llbracket \text{goodAbs } A; \text{goodAbs } B \rrbracket \implies \text{pick } A = \text{pick } B \longleftrightarrow A = B$

*<proof>*

**lemma** *goodAbs-imp-qGoodAbs-pick*:

*goodAbs A  $\implies$  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  $\implies$  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  $\vee$  qGoodAbs qB*

**shows** *(asAbs qA = asAbs qB) = (qA  $\$=$  qB)*

*<proof>*



**lemma** *pick-alphaAbs-iff-equal*:  
**assumes** *goodAbs A and goodAbs B*  
**shows**  $(\text{pick } A \ \$ = \text{pick } B) = (A = B)$   
 $\langle \text{proof} \rangle$

**lemma** *pick-swapAbs-qSwapAbs*:  
**assumes** *goodAbs A*  
**shows**  $\text{pick } (A \ \$[x1 \ \wedge \ x2]-xs) \ \$ = ((\text{pick } A) \ \$[[x1 \ \wedge \ x2]]-xs)$   
 $\langle \text{proof} \rangle$

**lemma** *asAbs-qSwapAbs-swapAbs*:  
**assumes** *qGoodAbs qA*  
**shows**  $\text{asAbs } (qA \ \$[[x1 \ \wedge \ x2]]-xs) = ((\text{asAbs } qA) \ \$[x1 \ \wedge \ x2]-xs)$   
 $\langle \text{proof} \rangle$

**lemma** *freshAbs-asAbs-qFreshAbs*:  
**assumes** *qGoodAbs qA*  
**shows**  $\text{freshAbs } xs \ x \ (\text{asAbs } qA) = \text{qFreshAbs } xs \ x \ qA$   
 $\langle \text{proof} \rangle$

**lemma** *skelAbs-asAbs-qSkelAbs*:  
**assumes** *qGoodAbs qA*  
**shows**  $\text{skelAbs } (\text{asAbs } qA) = \text{qSkelAbs } qA$   
 $\langle \text{proof} \rangle$

### 5.3.3 For inputs

For unbound inputs:

**lemma** *pickInp-inj-on-goodInp*: *inj-on pickInp (Collect goodInp)*  
 $\langle \text{proof} \rangle$

**lemma** *goodInp-imp-qGoodInp-pickInp*:  
**assumes** *goodInp inp*  
**shows** *qGoodInp (pickInp inp)*  
 $\langle \text{proof} \rangle$

**lemma** *qGoodInp-iff-goodInp-asInp*:  
 $\text{goodInp } (\text{asInp } qinp) = \text{qGoodInp } qinp$   
 $\langle \text{proof} \rangle$

**lemma** *pickInp-asInp*:  
**assumes** *qGoodInp qinp*  
**shows**  $\text{pickInp } (\text{asInp } qinp) \ \% = \ qinp$   
 $\langle \text{proof} \rangle$

**lemma** *asInp-pickInp*:  
**assumes** *goodInp inp*  
**shows**  $\text{asInp } (\text{pickInp } inp) = \text{inp}$

*<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*  $\implies$  (*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*  $\% [x1 \wedge x2]$  *-xs)*  $\% =$  (*(pickInp inp)*  $\% [[x1 \wedge x2]]$  *-xs)*  
*<proof>*

**lemma** *asInp-qSwapInp-swapInp*:  
assumes *qGoodInp qinp*  
shows *asInp (qinp*  $\% [[x1 \wedge x2]]$  *-xs)* = (*(asInp qinp)*  $\% [x1 \wedge x2]$  *-xs)*  
*<proof>*

**lemma** *swapInp-def2*:  
*(inp*  $\% [x1 \wedge x2]$  *-xs)* = *asInp ((pickInp inp)*  $\% [[x1 \wedge x2]]$  *-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  $\implies$  (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  $\wedge$  x2]-xs) %%= ((pickBinp binp) %%[[x1  $\wedge$  x2]]-xs)*  
{proof}

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

**shows**  $asBinp (qbinp \% \% [[x1 \wedge x2]] -xs) = ((asBinp qbinp) \% \% [x1 \wedge x2] -xs)$   
(proof)

**lemma** *swapBinp-def2*:  
 $(binp \% \% [x1 \wedge x2] -xs) = asBinp ((pickBinp binp) \% \% [[x1 \wedge x2]] -xs)$   
(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*:  
 $qGoodEnv qrho \implies (asEnv qrho = asEnv qrho') = (qrho \&= qrho')$   
(proof)

**lemma** *pickE-alphaEnv-iff-equal*:  
**assumes** *goodEnv rho and goodEnv rho'*  
**shows**  $(\text{pickE } rho \ \&= \ \text{pickE } rho') = (rho = rho')$   
 $\langle \text{proof} \rangle$

**lemma** *freshEnv-def2*:  
 $\text{freshEnv } xs \ x \ rho = \text{qFreshEnv } xs \ x \ (\text{pickE } rho)$   
 $\langle \text{proof} \rangle$

**lemma** *pick-psubst-qPsubst*:  
**assumes** *good X and goodEnv rho*  
**shows**  $\text{pick } (X \ \#[rho]) \ \# = ((\text{pick } X) \ \#[[\text{pickE } rho]])$   
 $\langle \text{proof} \rangle$

**lemma** *pick-psubstAbs-qPsubstAbs*:  
**assumes** *goodAbs A and goodEnv rho*  
**shows**  $\text{pick } (A \ \$[rho]) \ \$ = ((\text{pick } A) \ \$[[\text{pickE } rho]])$   
 $\langle \text{proof} \rangle$

**lemma** *pickInp-psubstInp-qPsubstInp*:  
**assumes** *good: goodInp inp and good-rho: goodEnv rho*  
**shows**  $\text{pickInp } (inp \ \%[rho]) \ \% = ((\text{pickInp } inp) \ \%[[\text{pickE } rho]])$   
 $\langle \text{proof} \rangle$

**lemma** *pickBinp-psubstBinp-qPsubstBinp*:  
**assumes** *good: goodBinp binp and good-rho: goodEnv rho*  
**shows**  $\text{pickBinp } (binp \ \%[rho]) \ \% = ((\text{pickBinp } binp) \ \%[[\text{pickE } rho]])$   
 $\langle \text{proof} \rangle$

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

**lemma** *pick-Var-qVar*:  
 $\text{pick } (\text{Var } xs \ x) \ \# = \text{qVar } xs \ x$   
 $\langle \text{proof} \rangle$

**lemma** *Op-asInp-asTerm-qOp*:  
**assumes** *qGoodInp qinp and qGoodBinp qbinp*  
**shows**  $\text{Op } \text{delta } (asInp \ qinp) \ (asBinp \ qbinp) = asTerm \ (qOp \ \text{delta } \ qinp \ qbinp)$   
 $\langle \text{proof} \rangle$

**lemma** *qOp-pickInp-pick-Op*:  
**assumes** *goodInp inp and goodBinp binp*  
**shows**  $\text{qOp } \text{delta } (\text{pickInp } inp) \ (\text{pickBinp } binp) \ \# = \text{pick } (\text{Op } \text{delta } inp \ binp)$   
 $\langle \text{proof} \rangle$

**lemma** *Abs-asTerm-asAbs-qAbs*:  
**assumes** *qGood qX*  
**shows**  $\text{Abs } xs \ x \ (asTerm \ qX) = asAbs \ (qAbs \ xs \ x \ qX)$

*<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  $\wedge$  y]-xs)*  
*<proof>*

**lemma** *swapAbs-preserves-good[simp]*:  
**assumes** *goodAbs A*  
**shows** *goodAbs(A \$[x  $\wedge$  y]-xs)*  
*<proof>*

**lemma** *swapInp-preserves-good[simp]*:  
**assumes** *goodInp inp*  
**shows** *goodInp (inp %[x  $\wedge$  y]-xs)*  
*<proof>*

**lemma** *swapBinp-preserves-good[simp]*:  
**assumes** *goodBinp binp*  
**shows** *goodBinp (binp %[%[x  $\wedge$  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  $\wedge$  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 ← 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*  $\wedge$  *x = y*)  
 ⟨*proof*⟩

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

**and**  $goodInp\ inp'$  **and**  $goodBinp\ binp'$

**shows**

$(Op\ delta\ inp\ binp = Op\ delta'\ inp'\ binp') =$   
 $(delta = delta' \wedge inp = inp' \wedge binp = binp')$   
 $\langle proof \rangle$

“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**

$(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\forall y. (y = x \vee fresh\ xs\ y\ X) \wedge (y = x' \vee fresh\ xs\ y\ X') \longrightarrow$   
 $(X\ \#[y \wedge x]-xs) = (X'\ \#[y \wedge x']-xs)))$   
 $\langle proof \rangle$

**lemma**  $Abs-ainj-ex$ :

**assumes**  $good: good\ X$  **and**  $good': good\ X'$

**shows**

$(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\exists y. y \notin \{x, x'\} \wedge fresh\ xs\ y\ X \wedge fresh\ xs\ y\ X' \wedge$   
 $(X\ \#[y \wedge x]-xs) = (X'\ \#[y \wedge x']-xs)))$   
 $\langle proof \rangle$

**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**  $eq: (X\ \#[y \wedge x]-xs) = (X'\ \#[y \wedge x']-xs)$

**shows**  $Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$

$\langle proof \rangle$

**lemma**  $Abs-swap-fresh$ :

**assumes**  $good-X: good\ X$  **and**  $fresh: fresh\ xs\ x'\ X$

**shows**  $Abs\ xs\ x\ X = Abs\ xs\ x'\ (X\ \#[x' \wedge x]-xs)$

$\langle proof \rangle$

**lemma**  $Var-diff-Op[simp]$ :

$Var\ xs\ x \neq Op\ delta\ inp\ binp$

$\langle proof \rangle$

**lemma**  $Op-diff-Var[simp]$ :

$Op\ delta\ inp\ binp \neq Var\ xs\ x$

$\langle proof \rangle$

**theorem** *term-nchotomy*:

**assumes** *good X*

**shows**

$(\exists xs x. X = \text{Var } xs \ x) \vee$

$(\exists \text{delta } inp \ \text{binp}. \text{goodInp } inp \wedge \text{goodBinp } \text{binp} \wedge X = \text{Op } \text{delta } \text{inp } \text{binp})$

*<proof>*

**theorem** *abs-nchotomy*:

**assumes** *goodAbs A*

**shows**  $\exists xs \ x \ X. \text{good } X \wedge A = \text{Abs } xs \ x \ X$

*<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]*:

$((\text{Var } xs \ x) \#[y1 \wedge y2]-ys) = \text{Var } xs \ (x \ @xs[y1 \wedge y2]-ys)$

*<proof>*

**lemma** *swap-Op-simp[simp]*:

**assumes** *goodInp inp goodBinp binp*

**shows**  $((\text{Op } \text{delta } \text{inp } \text{binp}) \#[x1 \wedge x2]-xs) =$

$\text{Op } \text{delta } (\text{inp } \%[x1 \wedge x2]-xs) (\text{binp } \%[x1 \wedge x2]-xs)$

*<proof>*

**lemma** *swapAbs-simp[simp]*:

**assumes** *good X*

**shows**  $((\text{Abs } xs \ x \ X) \$[y1 \wedge y2]-ys) = \text{Abs } xs \ (x \ @xs[y1 \wedge y2]-ys) (X \ #[y1 \wedge y2]-ys)$

*<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)  $\longleftrightarrow$*

*(ys  $\neq$  xs  $\vee$  y  $\neq$  x)*

*<proof>*

**lemma** *fresh-Op-simp[simp]*:

**assumes** *goodInp inp goodBinp binp*

**shows**

*fresh xs x (Op delta inp binp)  $\longleftrightarrow$*

*(freshInp xs x inp  $\wedge$  freshBinp xs x binp)*

*<proof>*

**lemma** *freshAbs-simp[simp]*:  
**assumes** *good X*  
**shows**  $\text{freshAbs } ys \ y \ (\text{Abs } xs \ x \ X) \longleftrightarrow (ys = xs \wedge y = x \vee \text{fresh } ys \ y \ X)$   
 $\langle \text{proof} \rangle$

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

**theorem** *skel-Var-simp[simp]*:  
 $\text{skel } (\text{Var } xs \ x) = \text{Branch } \text{Map.empty } \text{Map.empty}$   
 $\langle \text{proof} \rangle$

**lemma** *skel-Op-simp[simp]*:  
**assumes** *goodInp inp and goodBinp binp*  
**shows**  $\text{skel } (\text{Op } \text{delta } \text{inp } \text{binp}) = \text{Branch } (\text{skelInp } \text{inp}) (\text{skelBinp } \text{binp})$   
 $\langle \text{proof} \rangle$

**lemma** *skelAbs-simp[simp]*:  
**assumes** *good X*  
**shows**  $\text{skelAbs } (\text{Abs } xs \ x \ X) = \text{Branch } (\lambda i. \text{Some } (\text{skel } X)) \ \text{Map.empty}$   
 $\langle \text{proof} \rangle$

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

**lemma** *psubst-Var*:  
**assumes** *goodEnv rho*  
**shows**  $((\text{Var } xs \ x) \#[\text{rho}]) =$   
 $\quad (\text{case } \text{rho } xs \ x \ \text{of } \text{None} \Rightarrow \text{Var } xs \ x$   
 $\quad \quad \quad \quad \quad \quad \quad | \text{Some } X \Rightarrow X)$   
 $\langle \text{proof} \rangle$

**corollary** *psubst-Var-simp1[simp]*:  
**assumes** *goodEnv rho and rho xs x = None*  
**shows**  $((\text{Var } xs \ x) \#[\text{rho}]) = \text{Var } xs \ x$   
 $\langle \text{proof} \rangle$

**corollary** *psubst-Var-simp2[simp]*:  
**assumes** *goodEnv rho and rho xs x = Some X*  
**shows**  $((\text{Var } xs \ x) \#[\text{rho}]) = X$   
 $\langle \text{proof} \rangle$

**lemma** *psubst-Op-simp[simp]*:  
**assumes** *good-inp: goodInp inp goodBinp binp*  
**and** *good-rho: goodEnv rho*  
**shows**  
 $((\text{Op } \text{delta } \text{inp } \text{binp}) \#[\text{rho}]) = \text{Op } \text{delta } (\text{inp } \%[\text{rho}]) (\text{binp } \%[\text{rho}])$   
 $\langle \text{proof} \rangle$

**lemma** *psubstAbs-simp[simp]*:  
**assumes** *good-X: good X and good-rho: goodEnv rho and*  
*x-fresh-rho: freshEnv xs x rho*  
**shows**  $((Abs\ xs\ x\ X)\ \$[rho]) = Abs\ xs\ x\ (X\ \#[rho])$   
 $\langle proof \rangle$

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

**theorem** *getEnv-idEnv[simp]*:  $idEnv\ xs\ x = None$   
 $\langle proof \rangle$

**lemma** *getEnv-updEnv[simp]*:  
 $(rho\ [x \leftarrow X]-xs)\ ys\ y = (if\ ys = xs \wedge y = x\ then\ Some\ X\ else\ rho\ ys\ y)$   
 $\langle proof \rangle$

**theorem** *getEnv-updEnv1*:  
 $ys \neq xs \vee y \neq x \implies (rho\ [x \leftarrow X]-xs)\ ys\ y = rho\ ys\ y$   
 $\langle proof \rangle$

**theorem** *getEnv-updEnv2*:  
 $(rho\ [x \leftarrow X]-xs)\ xs\ x = Some\ X$   
 $\langle proof \rangle$

**lemma** *subst-Var-simp1[simp]*:  
**assumes** *good Y*  
**and**  $ys \neq xs \vee y \neq x$   
**shows**  $((Var\ xs\ x)\ \#[Y / y]-ys) = Var\ xs\ x$   
 $\langle proof \rangle$

**lemma** *subst-Var-simp2[simp]*:  
**assumes** *good Y*  
**shows**  $((Var\ xs\ x)\ \#[Y / x]-xs) = Y$   
 $\langle proof \rangle$

**lemma** *subst-Op-simp[simp]*:  
**assumes** *good Y*  
**and** *goodInp inp and goodBinp binp*  
**shows**  
 $((Op\ delta\ inp\ binp)\ \#[Y / y]-ys) =$   
 $Op\ delta\ (inp\ \%[Y / y]-ys)\ (binp\ \%[Y / y]-ys)$   
 $\langle proof \rangle$

**lemma** *substAbs-simp[simp]*:  
**assumes** *good: good Y and good-X: good X and*  
*x-dif-y: xs \neq ys \vee x \neq 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]*:  
 $((\text{Var } xs \ x) \#[y1 \ // \ y]-ys) = \text{Var } xs \ (x \ @xs[y1 \ / \ y]-ys)$   
*<proof>*

**lemma** *vsubst-Op-simp[simp]*:  
**assumes** *goodInp inp* **and** *goodBinp binp*  
**shows**  
 $((\text{Op } delta \ inp \ binp) \#[y1 \ // \ y]-ys) =$   
 $\text{Op } delta \ (inp \ \%[y1 \ // \ y]-ys) \ (binp \ \%[y1 \ // \ y]-ys)$   
*<proof>*

**lemma** *vsubstAbs-simp[simp]*:  
**assumes** *good X* **and**  
 $xs \neq ys \vee x \notin \{y, y1\}$   
**shows**  $((\text{Abs } xs \ x \ X) \$[y1 \ // \ y]-ys) = \text{Abs } xs \ x \ (X \ #[y1 \ // \ y]-ys)$   
*<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*:  
 $good \ X \implies |\{x. \neg \text{fresh } xs \ x \ X\}| <_o \ |UNIV :: 'var \ set|$   
*<proof>*

**lemma** *single-non-freshAbs-ordLess-var*:  
 $goodAbs \ A \implies |\{x. \neg \text{freshAbs } xs \ x \ A\}| <_o \ |UNIV :: 'var \ set|$   
*<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**  $Vvar: |V| <_o \ |UNIV :: 'var \ set| \vee \text{finite } V$  **and**  $XSvar: |XS| <_o \ |UNIV$   
 $:: 'var \ set| \vee \text{finite } XS$  **and**

$good: \forall X \in XS. good\ X$  **and**  
 $RhoVar: |Rho| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ Rho$  **and**  $RhoGood: \forall rho \in$   
 $Rho. goodEnv\ rho$   
**shows**  
 $\exists z. z \notin V \wedge$   
 $(\forall X \in XS. fresh\ xs\ z\ X) \wedge$   
 $(\forall rho \in Rho. freshEnv\ xs\ z\ rho)$   
 $\langle proof \rangle$

**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**  $Vvar: |V| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ V$  **and**

$XSvar: |XS| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ XS$  **and**

$ASvar: |AS| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ AS$  **and**

$RhoVar: |Rho| < o \mid UNIV :: 'var\ set \mid \vee\ finite\ Rho$  **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**

$\exists z. z \notin V \wedge$   
 $(\forall X \in XS. fresh\ xs\ z\ X) \wedge$   
 $(\forall A \in AS. freshAbs\ xs\ z\ A) \wedge$   
 $(\forall rho \in Rho. freshEnv\ xs\ z\ rho)$   
 $\langle proof \rangle$

### 5.5.3 Compositionality

**lemma** *swap-ident[simp]*:

**assumes**  $good\ X$

**shows**  $(X \#[x \wedge x]-zs) = X$

$\langle proof \rangle$

**lemma** *swap-compose*:

**assumes**  $good-X: good\ X$

**shows**  $((X \#[x \wedge y]-zs) \#[x' \wedge y']-zs') =$

$((X \#[x' \wedge y']-zs') \#[(x \ @zs[x' \wedge y']-zs') \wedge (y \ @zs[x' \wedge y']-zs')]-zs)$

$\langle proof \rangle$

**lemma** *swap-commute*:

$\llbracket good\ X; zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\} \rrbracket \implies$

$((X \#[x \wedge y]-zs) \#[x' \wedge y']-zs') = ((X \#[x' \wedge y']-zs') \#[x \wedge y]-zs)$

$\langle proof \rangle$

**lemma** *swap-involutive[simp]*:

**assumes**  $good-X: good\ X$

**shows**  $((X \#[x \wedge y]-zs) \#[x \wedge y]-zs) = X$

*<proof>*

**theorem** *swap-sym*:  $(X \#[x \wedge y]-zs) = (X \#[y \wedge x]-zs)$   
*<proof>*

**lemma** *swap-involutive2*[simp]:  
**assumes** *good X*  
**shows**  $((X \#[x \wedge y]-zs) \#[y \wedge x]-zs) = X$   
*<proof>*

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

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

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

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

**lemma** *fresh-swap-id*[simp]:  
**assumes** *good X* **and** *fresh xs x1 X* *fresh xs x2 X*  
**shows**  $(X \#[x1 \wedge x2]-xs) = X$   
*<proof>*

**lemma** *freshAbs-swapAbs-id*[simp]:  
**assumes** *goodAbs A* *freshAbs xs x1 A* *freshAbs xs x2 A*  
**shows**  $(A \ \$[x1 \wedge x2]-xs) = A$   
*<proof>*

**lemma** *fresh-swap-compose*:  
**assumes** *good X* *fresh xs y X* *fresh xs z X*  
**shows**  $((X \#[y \wedge x]-xs) \#[z \wedge y]-xs) = (X \#[z \wedge x]-xs)$   
*<proof>*

**lemma** *skel-swap*:



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

### 5.5.4 Compositionality for environments

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

**lemma** *swapEnv-compose*:  
**assumes** *good: goodEnv rho*  
**shows**  $((rho \&[x \wedge y]-zs) \&[x' \wedge y']-zs') =$   
 $((rho \&[x' \wedge y']-zs') \&[(x @zs[x' \wedge y']-zs') \wedge (y @zs[x' \wedge y']-zs')]-zs)$   
 ⟨*proof*⟩

**lemma** *swapEnv-commute*:  
 $\llbracket goodEnv rho; \{x,y\} \subseteq var zs; zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\} \rrbracket \implies$   
 $((rho \&[x \wedge y]-zs) \&[x' \wedge y']-zs') = ((rho \&[x' \wedge y']-zs') \&[x \wedge y]-zs)$   
 ⟨*proof*⟩

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

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

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

**lemma** *swapEnv-preserves-freshEnv[simp]*:  
**assumes** *good: goodEnv rho*  
**shows**  $freshEnv xs (x @xs[y1 \wedge y2]-ys) (rho \&[y1 \wedge y2]-ys) = freshEnv xs x rho$   
 ⟨*proof*⟩

**lemma** *swapEnv-preserves-freshEnv-distinct*:  
**assumes** *goodEnv rho* **and**  
 $xs \neq ys \vee x \notin \{y1,y2\}$   
**shows**  $freshEnv xs x (rho \&[y1 \wedge y2]-ys) = freshEnv xs x rho$   
 ⟨*proof*⟩

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

*<proof>*

**lemma** *freshEnv-swapEnv-exchange2*:

**assumes** *goodEnv rho*

**shows**  $\text{freshEnv } xs \ x2 \ (\rho \ \&[x2 \wedge x1]-xs) = \text{freshEnv } xs \ x1 \ \rho$

*<proof>*

**lemma** *freshEnv-swapEnv-id[simp]*:

**assumes** *good: goodEnv rho and*

*fresh: freshEnv xs x1 rho freshEnv xs x2 rho*

**shows**  $(\rho \ \&[x1 \wedge x2]-xs) = \rho$

*<proof>*

**lemma** *freshEnv-swapEnv-compose*:

**assumes** *good: goodEnv rho and*

*fresh: freshEnv xs y rho freshEnv xs z rho*

**shows**  $((\rho \ \&[y \wedge x]-xs) \ \&[z \wedge y]-xs) = (\rho \ \&[z \wedge x]-xs)$

*<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*:  $(X, X \ \#[x \wedge y]-zs) \in \text{swapped}$

*<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**  $\text{skel } Y = \text{skel } X$

*<proof>*

**lemma** *obtain-rep*:

**assumes** *GOOD: good X and FRESH: fresh xs x' X*

**shows**  $\exists X'. (X, X') \in \text{swapped} \wedge \text{good } X' \wedge \text{Abs } xs \ x \ X = \text{Abs } xs \ x' \ X'$

*<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**  
 $rel: \bigwedge X\ Y. \llbracket good\ X; (X, Y) \in rel \rrbracket \implies good\ Y \wedge skel\ Y = skel\ X$  **and**  
 $var: \bigwedge xs\ x. phi\ (Var\ xs\ x)$  **and**  
 $op: \bigwedge delta\ inp\ binp. \llbracket goodInp\ inp; goodBinp\ binp; liftAll\ phi\ inp; liftAll\ phiAbs\ binp \rrbracket$   
 $\implies phi\ (Op\ delta\ inp\ binp)$  **and**  
 $abs: \bigwedge xs\ x\ X. \llbracket good\ X; \bigwedge Y. (X, Y) \in rel \implies phi\ Y \rrbracket$   
 $\implies phiAbs\ (Abs\ xs\ x\ X)$   
**shows**  $(good\ X \implies phi\ X) \wedge (goodAbs\ A \implies phiAbs\ A)$   
 $\langle proof \rangle$

**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**  
 $Op: \bigwedge delta\ inp\ binp. \llbracket goodInp\ inp; goodBinp\ binp; liftAll\ phi\ inp; liftAll\ phiAbs\ binp \rrbracket$   
 $\implies phi\ (Op\ delta\ inp\ binp)$  **and**  
 $Abs: \bigwedge xs\ x\ X. \llbracket good\ X; phi\ X \rrbracket \implies phiAbs\ (Abs\ xs\ x\ X)$   
**shows**  $(good\ X \implies phi\ X) \wedge (goodAbs\ A \implies phiAbs\ A)$   
 $\langle proof \rangle$

**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**  
 $Op: \bigwedge delta\ inp\ binp. \llbracket goodInp\ inp; goodBinp\ binp; liftAll\ phi\ inp; liftAll\ phiAbs\ binp \rrbracket$   
 $\implies phi\ (Op\ delta\ inp\ binp)$  **and**  
 $Abs: \bigwedge xs\ x\ X. \llbracket good\ X;$   
 $\bigwedge Y. (X, Y) \in swapped \implies phi\ Y;$   
 $\bigwedge Y. \llbracket good\ Y; skel\ Y = skel\ X \rrbracket \implies phi\ Y \rrbracket$   
 $\implies phiAbs\ (Abs\ xs\ x\ X)$   
**shows**  $(good\ X \implies phi\ X) \wedge (goodAbs\ A \implies phiAbs\ A)$   
 $\langle proof \rangle$

## 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:**  $\bigwedge p \text{ xs. } ( |\text{varsOf xs } p| < o \text{ |UNIV::'var set| } ) \text{ and}$   
**var:**  $\bigwedge \text{xs } x \text{ p. } \text{phi } ( \text{Var } \text{xs } x ) \text{ p and}$   
**op:**  $\bigwedge \text{delta inp binp p.}$   
 $\llbracket \{i. \text{inp } i \neq \text{None}\} | < o \text{ |UNIV::'var set|}; \{i. \text{binp } i \neq \text{None}\} | < o \text{ |UNIV::'var set|};$   
 $\text{liftAll } ( \lambda X. \text{good } X \wedge ( \forall q. \text{phi } X \text{ p} ) ) \text{ inp}; \text{liftAll } ( \lambda A. \text{goodAbs } A \wedge ( \forall q. \text{phiAbs } A \text{ p} ) ) \text{ binp} \rrbracket$   
 $\implies \text{phi } ( \text{Op } \text{delta inp binp} ) \text{ p and}$   
**abs:**  $\bigwedge \text{xs } x \text{ X p. } \llbracket \text{good } X; x \notin \text{varsOf } p \text{ xs}; \text{phi } X \text{ p} \rrbracket \implies \text{phiAbs } ( \text{Abs } \text{xs } x \text{ X} ) \text{ p}$   
**shows**  $( \text{good } X \longrightarrow ( \forall p. \text{phi } X \text{ p} ) ) \wedge ( \text{goodAbs } A \longrightarrow ( \forall p. \text{phiAbs } A \text{ p} ) )$   
 $\langle \text{proof} \rangle$

**lemma term-templateInduct-fresh[case-names PAR Var Op Abs]:**

**fixes**  $X::('index, 'bindex, 'varSort, 'var, 'opSym)\text{term}$  **and**

$A::('index, 'bindex, 'varSort, 'var, 'opSym)\text{abs}$  **and**

$\text{rel}$  **and**  $\text{phi}$  **and**  $\text{phiAbs}$  **and**

$\text{vars}::'varSort \Rightarrow 'var \text{ set}$  **and**

$\text{terms}::('index, 'bindex, 'varSort, 'var, 'opSym)\text{term set}$  **and**

$\text{abs}::('index, 'bindex, 'varSort, 'var, 'opSym)\text{abs set}$  **and**

$\text{envs}::('index, 'bindex, 'varSort, 'var, 'opSym)\text{env set}$

**assumes**

**PAR:**

$\bigwedge \text{xs.}$

$( |\text{vars } \text{xs}| < o \text{ |UNIV}::'var \text{ set}| \vee \text{finite } ( \text{vars } \text{xs} ) ) \wedge$

$( |\text{terms}| < o \text{ |UNIV}::'var \text{ set}| \vee \text{finite } \text{terms} ) \wedge ( \forall X \in \text{terms. good } X ) \wedge$

$( |\text{abs}| < o \text{ |UNIV}::'var \text{ set}| \vee \text{finite } \text{abs} ) \wedge ( \forall A \in \text{abs. goodAbs } A ) \wedge$

$( |\text{envs}| < o \text{ |UNIV}::'var \text{ set}| \vee \text{finite } \text{envs} ) \wedge ( \forall \text{rho} \in \text{envs. goodEnv } \text{rho} ) \text{ and}$

**rel:**  $\bigwedge X \text{ Y. } \llbracket \text{good } X; (X, Y) \in \text{rel} \rrbracket \implies \text{good } Y \wedge \text{skel } Y = \text{skel } X \text{ and}$

**Var:**  $\bigwedge \text{xs } x. \text{phi } ( \text{Var } \text{xs } x ) \text{ and}$

**Op:**

$\bigwedge \text{delta inp binp.}$

$\llbracket \text{goodInp } \text{inp}; \text{goodBinp } \text{binp};$

$\text{liftAll } \text{phi } \text{inp}; \text{liftAll } \text{phiAbs } \text{binp} \rrbracket$

$\implies \text{phi } ( \text{Op } \text{delta inp binp} ) \text{ and}$

**abs:**

$\bigwedge \text{xs } x \text{ X.}$

$\llbracket \text{good } X;$

$x \notin \text{vars } \text{xs};$

$\bigwedge Y. Y \in \text{terms} \implies \text{fresh } \text{xs } x \text{ Y};$

$\bigwedge A. A \in \text{abs} \implies \text{freshAbs } \text{xs } x \text{ A};$

$\bigwedge \text{rho. } \text{rho} \in \text{envs} \implies \text{freshEnv } \text{xs } x \text{ rho};$

$\bigwedge Y. (X, Y) \in \text{rel} \implies \text{phi } Y \rrbracket$

$\implies \text{phiAbs } ( \text{Abs } \text{xs } x \text{ X} )$

**shows**

$( \text{good } X \longrightarrow \text{phi } X ) \wedge$

$( \text{goodAbs } A \longrightarrow \text{phiAbs } A )$

$\langle \text{proof} \rangle$

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**  
 $vars :: 'varSort \Rightarrow 'var set$  **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*:

$\bigwedge xs.$

( $|vars\ xs| < o\ |UNIV :: 'var\ set| \vee finite\ (vars\ xs)$ )  $\wedge$   
( $|terms| < o\ |UNIV :: 'var\ set| \vee finite\ terms$ )  $\wedge (\forall X \in terms. good\ X) \wedge$   
( $|abs| < o\ |UNIV :: 'var\ set| \vee finite\ abs$ )  $\wedge (\forall A \in abs. goodAbs\ A) \wedge$   
( $|envs| < o\ |UNIV :: 'var\ set| \vee finite\ envs$ )  $\wedge (\forall rho \in envs. goodEnv\ rho)$  **and**

*Var*:  $\bigwedge xs\ x. phi\ (Var\ xs\ x)$  **and**

*Op*:

$\bigwedge delta\ inp\ binp.$

$\llbracket goodInp\ inp; goodBinp\ binp;$   
 $liftAll\ phi\ inp; liftAll\ phiAbs\ binp \rrbracket$   
 $\implies phi\ (Op\ delta\ inp\ binp)$  **and**

*Abs*:

$\bigwedge xs\ x\ X.$

$\llbracket good\ X;$   
 $x \notin vars\ xs;$   
 $\bigwedge Y. Y \in terms \implies fresh\ xs\ x\ Y;$   
 $\bigwedge A. A \in abs \implies freshAbs\ xs\ x\ A;$   
 $\bigwedge rho. rho \in envs \implies freshEnv\ xs\ x\ rho;$   
 $phi\ X \rrbracket$   
 $\implies phiAbs\ (Abs\ xs\ x\ X)$

**shows**

$(good\ X \longrightarrow phi\ X) \wedge$

$(goodAbs\ A \longrightarrow phiAbs\ A)$

*<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*:  $\bigwedge xs\ x. phi\ (Var\ xs\ x)$  **and**

*Op*:

$\bigwedge delta\ inp\ binp.$

$\llbracket goodInp\ inp; goodBinp\ binp;$   
 $liftAll\ phi\ inp; liftAll\ phiAbs\ binp \rrbracket$

```

     $\implies \text{phi } (Op \text{ delta inp binp})$  and
  Abs:
   $\wedge xs \ x \ X.$ 
   $\llbracket \text{good } X;$ 
   $x \notin \text{varsOf } P;$ 
   $\wedge Y. Y \in \text{termsOf } P \implies \text{fresh } xs \ x \ Y;$ 
   $\wedge A. A \in \text{absOf } P \implies \text{freshAbs } xs \ x \ A;$ 
   $\wedge rho. rho \in \text{envsOf } P \implies \text{freshEnv } xs \ x \ rho;$ 
   $\text{phi } X \rrbracket$ 
   $\implies \text{phiAbs } (Abs \ xs \ x \ X)$ 
shows
   $(\text{good } X \longrightarrow \text{phi } X) \wedge$ 
   $(\text{goodAbs } A \longrightarrow \text{phiAbs } A)$ 
   $\langle \text{proof} \rangle$ 

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-terms).

```

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]:
   $(\text{idEnv } [x \leftarrow X]\text{-xs}) \ y \ y = (\text{if } (ys = xs \wedge y = x) \text{ then } \text{Some } X \text{ else } \text{None})$ 
   $\langle \text{proof} \rangle$ 

```

```

theorem subst-psubst-idEnv:
   $(X \ \#[Y / y]\text{-ys}) = (X \ \#[\text{idEnv } [y \leftarrow Y]\text{-ys}]$ 
   $\langle \text{proof} \rangle$ 

```

```

theorem vsubst-psubst-idEnv:

```

$(X \#[z // y]-ys) = (X \#[idEnv [y \leftarrow Var\ ys\ z]-ys])$   
 ⟨proof⟩

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

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

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

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

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

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

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

**lemma** *psubstAll-idEnv*:

**fixes**  $X::('index, 'bindex, 'varSort, 'var, 'opSym)term$  **and**  
 $A::('index, 'bindex, 'varSort, 'var, 'opSym)abs$

**shows**

$(good\ X \longrightarrow (X \#[idEnv]) = X) \wedge$   
 $(goodAbs\ A \longrightarrow (A \ \$[idEnv]) = A)$   
 ⟨proof⟩

**lemma** *psubst-idEnv[simp]*:  
 $good\ X \Longrightarrow (X \#[idEnv]) = X$   
 ⟨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]*:  $((rho [x \leftarrow X]-xs) [x \leftarrow X^\uparrow]-xs) = (rho [x \leftarrow X^\uparrow]-xs)$   
 ⟨proof⟩

**theorem** *updEnv-commute*:

**assumes**  $xs \neq ys \vee x \neq y$

**shows**  $((\text{rho } [x \leftarrow X]\text{-xs}) [y \leftarrow Y]\text{-ys}) = ((\text{rho } [y \leftarrow Y]\text{-ys}) [x \leftarrow X]\text{-xs})$   
*<proof>*

**theorem** *freshEnv-updEnv-E1*:

**assumes** *freshEnv*  $xs$   $y$   $(\text{rho } [x \leftarrow X]\text{-xs})$

**shows**  $y \neq x$   
*<proof>*

**theorem** *freshEnv-updEnv-E2*:

**assumes** *freshEnv*  $ys$   $y$   $(\text{rho } [x \leftarrow X]\text{-xs})$

**shows** *fresh*  $ys$   $y$   $X$   
*<proof>*

**theorem** *freshEnv-updEnv-E3*:

**assumes** *freshEnv*  $ys$   $y$   $(\text{rho } [x \leftarrow X]\text{-xs})$

**shows**  $\text{rho } ys$   $y = \text{None}$   
*<proof>*

**theorem** *freshEnv-updEnv-E4*:

**assumes** *freshEnv*  $ys$   $y$   $(\text{rho } [x \leftarrow X]\text{-xs})$

**and**  $zs \neq xs \vee z \neq x$  **and**  $\text{rho } zs$   $z = \text{Some } Z$

**shows** *fresh*  $ys$   $y$   $Z$   
*<proof>*

**theorem** *freshEnv-updEnv-I*:

**assumes**  $ys \neq xs \vee y \neq x$  **and** *fresh*  $ys$   $y$   $X$  **and**  $\text{rho } ys$   $y = \text{None}$

**and**  $\bigwedge zs$   $z$   $Z. \llbracket zs \neq xs \vee z \neq x; \text{rho } zs$   $z = \text{Some } Z \rrbracket \implies \text{fresh } ys$   $y$   $Z$

**shows** *freshEnv*  $ys$   $y$   $(\text{rho } [x \leftarrow X]\text{-xs})$   
*<proof>*

**theorem** *swapEnv-updEnv*:

$((\text{rho } [x \leftarrow X]\text{-xs}) \&[y1 \wedge y2]\text{-ys}) =$

$((\text{rho } \&[y1 \wedge y2]\text{-ys}) [(x @xs[y1 \wedge y2]\text{-ys}) \leftarrow (X \#[y1 \wedge y2]\text{-ys})]\text{-xs})$   
*<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**  $((\text{rho } [x \leftarrow X]\text{-xs}) \&[y1 \wedge y2]\text{-ys}) =$   
 $((\text{rho } \&[y1 \wedge y2]\text{-ys}) [x \leftarrow X]\text{-xs})$   
*<proof>*

**theorem** *psubstEnv-updEnv*:

$((\text{rho } [x \leftarrow X]\text{-xs}) \&[\text{rho}']) = ((\text{rho } \&[\text{rho}']) [x \leftarrow (X \#[\text{rho}'])]\text{-xs})$   
*<proof>*

**theorem** *psubstEnv-updEnv-idEnv*:



$((idEnv [x \leftarrow X]-xs) \&[rho]) = (rho [x \leftarrow (X \#[rho])]-xs)$   
 $\langle proof \rangle$

**theorem** *substEnv-updEnv*:

$((rho [x \leftarrow X]-xs) \&[Y / y]-ys) = ((rho \&[Y / y]-ys) [x \leftarrow (X \#[Y / y]-ys)]-xs)$   
 $\langle proof \rangle$

**theorem** *vsubstEnv-updEnv*:

$((rho [x \leftarrow X]-xs) \&[y1 // y]-ys) = ((rho \&[y1 // y]-ys) [x \leftarrow (X \#[y1 // y]-ys)]-xs)$   
 $\langle proof \rangle$

### 6.3 Environment “get” versus other operators

Currently, “get” is just function application. While the next properties are immediate consequences of the definitions, it is worth stating them because of their abstract character (since later, concrete terms inferred from abstract terms by a presumptive package, “get” will no longer be function application).

**theorem** *getEnv-ext*:

**assumes**  $\bigwedge xs x. rho \ xs \ x = rho' \ xs \ x$

**shows**  $rho = rho'$

$\langle proof \rangle$

**theorem** *freshEnv-getEnv1[simp]*:

$\llbracket freshEnv \ ys \ y \ rho; rho \ xs \ x = Some \ X \rrbracket \implies ys \neq xs \vee y \neq x$

$\langle proof \rangle$

**theorem** *freshEnv-getEnv2[simp]*:

$\llbracket freshEnv \ ys \ y \ rho; rho \ xs \ x = Some \ X \rrbracket \implies fresh \ ys \ y \ X$

$\langle proof \rangle$

**theorem** *freshEnv-getEnv[simp]*:

$freshEnv \ ys \ y \ rho \implies rho \ ys \ y = None$

$\langle proof \rangle$

**theorem** *getEnv-swapEnv1[simp]*:

**assumes**  $rho \ xs \ (x \ @xs \ [z1 \ \wedge \ z2]-zs) = None$

**shows**  $(rho \ \&[z1 \ \wedge \ z2]-zs) \ xs \ x = None$

$\langle proof \rangle$

**theorem** *getEnv-swapEnv2[simp]*:

**assumes**  $rho \ xs \ (x \ @xs \ [z1 \ \wedge \ z2]-zs) = Some \ X$

**shows**  $(rho \ \&[z1 \ \wedge \ z2]-zs) \ xs \ x = Some \ (X \ \#[z1 \ \wedge \ z2]-zs)$

$\langle proof \rangle$

**theorem** *getEnv-psubstEnv-None[simp]*:

**assumes**  $rho \ xs \ x = None$

**shows**  $(\rho \&[\rho']) \text{ xs } x = \rho' \text{ xs } x$   
*<proof>*

**theorem** *getEnv-psubstEnv-Some[simp]*:  
**assumes**  $\rho \text{ xs } x = \text{Some } X$   
**shows**  $(\rho \&[\rho']) \text{ xs } x = \text{Some } (X \#[\rho'])$   
*<proof>*

**theorem** *getEnv-substEnv1[simp]*:  
**assumes**  $ys \neq xs \vee y \neq x$  **and**  $\rho \text{ xs } x = \text{None}$   
**shows**  $(\rho \&[Y / y]\text{-ys}) \text{ xs } x = \text{None}$   
*<proof>*

**theorem** *getEnv-substEnv2[simp]*:  
**assumes**  $ys \neq xs \vee y \neq x$  **and**  $\rho \text{ xs } x = \text{Some } X$   
**shows**  $(\rho \&[Y / y]\text{-ys}) \text{ xs } x = \text{Some } (X \#[Y / y]\text{-ys})$   
*<proof>*

**theorem** *getEnv-substEnv3[simp]*:  
 $\llbracket ys \neq xs \vee y \neq x; \text{freshEnv } \text{xs } x \ \rho \rrbracket$   
 $\implies (\rho \&[Y / y]\text{-ys}) \text{ xs } x = \text{None}$   
*<proof>*

**theorem** *getEnv-substEnv4[simp]*:  
 $\text{freshEnv } ys \ y \ \rho \implies (\rho \&[Y / y]\text{-ys}) \text{ ys } y = \text{Some } Y$   
*<proof>*

**theorem** *getEnv-vsubstEnv1[simp]*:  
**assumes**  $ys \neq xs \vee y \neq x$  **and**  $\rho \text{ xs } x = \text{None}$   
**shows**  $(\rho \&[y1 // y]\text{-ys}) \text{ xs } x = \text{None}$   
*<proof>*

**theorem** *getEnv-vsubstEnv2[simp]*:  
**assumes**  $ys \neq xs \vee y \neq x$  **and**  $\rho \text{ xs } x = \text{Some } X$   
**shows**  $(\rho \&[y1 // y]\text{-ys}) \text{ xs } x = \text{Some } (X \#[y1 // y]\text{-ys})$   
*<proof>*

**theorem** *getEnv-vsubstEnv3[simp]*:  
 $\llbracket ys \neq xs \vee y \neq x; \text{freshEnv } \text{xs } x \ \rho \rrbracket$   
 $\implies (\rho \&[z // y]\text{-ys}) \text{ xs } x = \text{None}$   
*<proof>*

**theorem** *getEnv-vsubstEnv4[simp]*:  
 $\text{freshEnv } ys \ y \ \rho \implies (\rho \&[z // y]\text{-ys}) \text{ ys } y = \text{Some } (\text{Var } ys \ z)$   
*<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**

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

**corollary** *fresh-psubst*:

**assumes** *good X* **and** *goodEnv rho*

**shows**

$fresh\ zs\ z\ (X\ \#[rho]) =$   
 $(\forall\ ys\ y.\ fresh\ ys\ y\ X \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)$   
 $\langle proof \rangle$

**corollary** *fresh-psubst-E1*:

**assumes** *good X* **and** *goodEnv rho*

**and**  $\rho\ ys\ y = None$  **and**  $fresh\ zs\ z\ (X\ \#[rho])$

**shows**  $fresh\ ys\ y\ X \vee (ys \neq zs \vee y \neq z)$

$\langle proof \rangle$

**corollary** *fresh-psubst-E2*:

**assumes** *good X* **and** *goodEnv rho*

**and**  $\rho\ ys\ y = Some\ Y$  **and**  $fresh\ zs\ z\ (X\ \#[rho])$

**shows**  $fresh\ ys\ y\ X \vee fresh\ zs\ z\ Y$

$\langle proof \rangle$

**corollary** *fresh-psubst-I1*:

**assumes** *good X* **and** *goodEnv rho*

**and**  $fresh\ zs\ z\ X$  **and**  $freshEnv\ zs\ z\ rho$

**shows**  $fresh\ zs\ z\ (X\ \#[rho])$

$\langle proof \rangle$

**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  $\implies$  fresh zs z X and*  
 $\wedge$  *ys y Y. rho ys y = Some Y  $\implies$  fresh ys y X  $\vee$  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) =*  
 $((zs = ys \wedge z = y) \vee \text{fresh } zs \ z \ X) \wedge (\text{fresh } ys \ y \ X \vee \text{fresh } zs \ z \ Y)$   
⟨*proof*⟩

**lemma** *fresh-ubst*:  
**assumes** *good X*  
**shows** *fresh zs z (X #[y1 // y]-ys) =*  
 $((zs = ys \wedge z = y) \vee \text{fresh } zs \ z \ X) \wedge (\text{fresh } ys \ y \ X \vee (zs \neq ys \vee z \neq y1))$   
⟨*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  $\neq$  ys  $\vee$  z  $\neq$  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  $\neq$  ys  $\vee$  z  $\neq$  y*  
**shows** *freshEnv zs z (rho &[Y / y]-ys)*  
⟨*proof*⟩

**lemma** *ubst-preserves-fresh*:  
**assumes** *good X*  
**and** *fresh zs z X and zs  $\neq$  ys  $\vee$  z  $\neq$  y1*  
**shows** *fresh zs z (X #[y1 // y]-ys)*

*<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*  $\vee$  *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**

$(\text{good } X \longrightarrow$   
   $(\forall \text{ rho rho}'. \{ \text{rho}, \text{rho}' \} \subseteq \text{envsOf } P \longrightarrow$   
     $(\forall \text{ ys}. \forall \text{ y}. \text{fresh } \text{ys } \text{y } X \vee \text{rho } \text{ys } \text{y} = \text{rho}' \text{ys } \text{y}) \longrightarrow$   
       $(X \#[\text{rho}] = (X \#[\text{rho}'])))$   
 $\wedge$   
   $(\text{goodAbs } A \longrightarrow$   
     $(\forall \text{ rho rho}'. \{ \text{rho}, \text{rho}' \} \subseteq \text{envsOf } P \longrightarrow$   
       $(\forall \text{ ys}. \forall \text{ y}. \text{freshAbs } \text{ys } \text{y } A \vee \text{rho } \text{ys } \text{y} = \text{rho}' \text{ys } \text{y}) \longrightarrow$   
         $(A \#[\text{rho}] = (A \#[\text{rho}'])))$   
 $\langle \text{proof} \rangle$

**corollary** *psubst-cong*[*fundef-cong*]:

**assumes** *good*  $X$  **and** *goodEnv*  $\text{rho}$  **and** *goodEnv*  $\text{rho}'$   
**and**  $\bigwedge \text{ys } \text{y}. \text{fresh } \text{ys } \text{y } X \vee \text{rho } \text{ys } \text{y} = \text{rho}' \text{ys } \text{y}$   
**shows**  $(X \#[\text{rho}] = (X \#[\text{rho}']))$   
 $\langle \text{proof} \rangle$

**lemma** *fresh-psubst-updEnv*:

**assumes** *good*  $X$  **and** *good*  $Y$  **and** *goodEnv*  $\text{rho}$   
**and** *fresh*  $\text{xs } x Y$   
**shows**  $(Y \#[\text{rho } [x \leftarrow X]\text{-xs}] = (Y \#[\text{rho}]))$   
 $\langle \text{proof} \rangle$

**lemma** *psubstAll-ident*:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym) \text{term}$  **and**  
   $A :: ('index, 'bindex, 'varSort, 'var, 'opSym) \text{abs}$  **and**  
   $P :: ('index, 'bindex, 'varSort, 'var, 'opSym) \text{Transition-QuasiTerms-Terms.param}$   
**assumes**  $P$ : *goodPar*  $P$   
**shows**  
 $(\text{good } X \longrightarrow$   
   $(\forall \text{ rho} \in \text{envsOf } P.$   
     $(\forall \text{ zs } z. \text{freshEnv } \text{zs } z \text{ rho} \vee \text{fresh } \text{zs } z X)$   
       $\longrightarrow (X \#[\text{rho}] = X))$   
 $\wedge$   
   $(\text{goodAbs } A \longrightarrow$   
     $(\forall \text{ rho} \in \text{envsOf } P.$   
       $(\forall \text{ zs } z. \text{freshEnv } \text{zs } z \text{ rho} \vee \text{freshAbs } \text{zs } z A)$   
         $\longrightarrow (A \#[\text{rho}] = A))$   
 $\langle \text{proof} \rangle$

**corollary** *freshEnv-psubst-ident*[*simp*]:

**fixes**  $X :: ('index, 'bindex, 'varSort, 'var, 'opSym) \text{term}$   
**assumes** *good*  $X$  **and** *goodEnv*  $\text{rho}$   
**and**  $\bigwedge \text{zs } z. \text{freshEnv } \text{zs } z \text{ rho} \vee \text{fresh } \text{zs } z X$   
**shows**  $(X \#[\text{rho}] = X)$   
 $\langle \text{proof} \rangle$

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

**corollary** *substEnv-updEnv-fresh*:  
**assumes** *good X and good Y and fresh ys y X*  
**shows**  $((rho [x \leftarrow X]-xs) \&[Y / y]-ys) = ((rho \&[Y / y]-ys) [x \leftarrow X]-xs)$   
 $\langle proof \rangle$

**lemma** *fresh-substEnv-updEnv[simp]*:  
**assumes** *rho: goodEnv rho and Y: good Y*  
**and** *\*: freshEnv ys y rho*  
**shows**  $(rho \&[Y / y]-ys) = (rho [y \leftarrow Y]-ys)$   
 $\langle proof \rangle$

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

**corollary** *vsubstEnv-updEnv-fresh*:  
**assumes** *good X and fresh ys y X*  
**shows**  $((rho [x \leftarrow X]-xs) \&[y1 // y]-ys) = ((rho \&[y1 // y]-ys) [x \leftarrow X]-xs)$   
 $\langle proof \rangle$

**lemma** *fresh-vsubstEnv-updEnv[simp]*:  
**assumes** *rho: goodEnv rho*  
**and** *\*: freshEnv ys y rho*  
**shows**  $(rho \&[y1 // y]-ys) = (rho [y \leftarrow Var ys y1]-ys)$   
 $\langle proof \rangle$

**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**  
 $(good X \longrightarrow$   
 $(\forall rho z1 z2. rho \in envsOf P \wedge \{z1, z2\} \subseteq varsOf P \longrightarrow$   
 $((X \#[rho]) \#[z1 \wedge z2]-zs) = ((X \#[z1 \wedge z2]-zs) \#[rho \&[z1 \wedge$   
 $z2]-zs))))$   
 $\wedge$   
 $(goodAbs A \longrightarrow$   
 $(\forall rho z1 z2. rho \in envsOf P \wedge \{z1, z2\} \subseteq varsOf P \longrightarrow$   
 $((A \$[rho]) \$[z1 \wedge z2]-zs) = ((A \$[z1 \wedge z2]-zs) \$[rho \&[z1 \wedge z2]-zs))))$   
 $\langle proof \rangle$

**lemma** *swap-psubst*:

**assumes** *good X and goodEnv rho*

**shows**  $((X \#[\text{rho}]) \#[z1 \wedge z2]\text{-zs}) = ((X \#[z1 \wedge z2]\text{-zs}) \#[\text{rho} \&[z1 \wedge z2]\text{-zs}])$   
*<proof>*

**lemma** *swap-subst*:

**assumes** *good X and good Y*

**shows**  $((X \#[Y / y]\text{-ys}) \#[z1 \wedge z2]\text{-zs}) =$   
 $((X \#[z1 \wedge z2]\text{-zs}) \#[(Y \#[z1 \wedge z2]\text{-zs}) / (y \text{@ys}[z1 \wedge z2]\text{-zs})]\text{-ys})$   
*<proof>*

**lemma** *swap-vsubst*:

**assumes** *good X*

**shows**  $((X \#[y1 // y]\text{-ys}) \#[z1 \wedge z2]\text{-zs}) =$   
 $((X \#[z1 \wedge z2]\text{-zs}) \#[(y1 \text{@ys}[z1 \wedge z2]\text{-zs}) // (y \text{@ys}[z1 \wedge z2]\text{-zs})]\text{-ys})$   
*<proof>*

**lemma** *swapEnv-psubstEnv*:

**assumes** *goodEnv rho and goodEnv rho'*

**shows**  $((\text{rho} \&[\text{rho}']) \&[z1 \wedge z2]\text{-zs}) = ((\text{rho} \&[z1 \wedge z2]\text{-zs}) \&[\text{rho}' \&[z1 \wedge z2]\text{-zs}])$   
*<proof>*

**lemma** *swapEnv-substEnv*:

**assumes** *good Y and goodEnv rho*

**shows**  $((\text{rho} \&[Y / y]\text{-ys}) \&[z1 \wedge z2]\text{-zs}) =$   
 $((\text{rho} \&[z1 \wedge z2]\text{-zs}) \&[(Y \#[z1 \wedge z2]\text{-zs}) / (y \text{@ys}[z1 \wedge z2]\text{-zs})]\text{-ys})$   
*<proof>*

**lemma** *swapEnv-vsubstEnv*:

**assumes** *goodEnv rho*

**shows**  $((\text{rho} \&[y1 // y]\text{-ys}) \&[z1 \wedge z2]\text{-zs}) =$   
 $((\text{rho} \&[z1 \wedge z2]\text{-zs}) \&[(y1 \text{@ys}[z1 \wedge z2]\text{-zs}) // (y \text{@ys}[z1 \wedge z2]\text{-zs})]\text{-ys})$   
*<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**

$(good X \longrightarrow$

$(\forall rho rho'. \{rho, rho'\} \subseteq envsOf P \longrightarrow ((X \#[\text{rho}]) \#[\text{rho}']) = (X \#[(\text{rho} \&[\text{rho}'])]))$

$\wedge$

$(goodAbs A \longrightarrow$

$(\forall rho rho'. \{rho, rho'\} \subseteq envsOf P \longrightarrow ((A \$[\text{rho}]) \$[\text{rho}']) = (A \$[(\text{rho} \&[\text{rho}'])]))$ )  
*<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**  $((X \#[Y / y]-ys) \#[rho]) = (X \#[(rho [y \leftarrow (Y \#[rho])]-ys)])$

*<proof>*

**lemma** *psubstEnv-substEnv-compose*:

**assumes** *goodEnv rho and good Y and goodEnv rho'*

**shows**  $((rho \&[Y / y]-ys) \&[rho']) = (rho \&[(rho' [y \leftarrow (Y \#[rho'])]-ys)])$

*<proof>*

**lemma** *psubst-vsubst-compose*:

**assumes** *good X and goodEnv rho*

**shows**  $((X \#[y1 // y]-ys) \#[rho]) = (X \#[(rho [y \leftarrow ((Var ys y1) \#[rho])]-ys)])$

*<proof>*

**lemma** *psubstEnv-vsubstEnv-compose*:

**assumes** *goodEnv rho and goodEnv rho'*

**shows**  $((rho \&[y1 // y]-ys) \&[rho']) = (rho \&[(rho' [y \leftarrow ((Var ys y1) \#[rho'])]-ys)])$

*<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**  $((\rho \&[Y / y]\text{-ys}) \&[\rho']) = ((\rho \&[\rho']) \&[(Y \#[\rho']) / y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubst-psubst-compose*:

**assumes** *good X and goodEnv rho*

**shows**  $((X \#[\rho]) \#[y1 // y]\text{-ys}) = (X \#[(\rho \&[y1 // y]\text{-ys})])$   
 $\langle \text{proof} \rangle$

**lemma** *vsubstEnv-psubstEnv-compose*:

**assumes** *goodEnv rho and goodEnv rho'*

**shows**  $((\rho \&[\rho']) \&[y1 // y]\text{-ys}) = (\rho \&[(\rho' \&[y1 // y]\text{-ys})])$   
 $\langle \text{proof} \rangle$

**lemma** *subst-compose1*:

**assumes** *good X and good Y1 and good Y2*

**shows**  $((X \#[Y1 / y]\text{-ys}) \#[Y2 / y]\text{-ys}) = (X \#[(Y1 \#[Y2 / y]\text{-ys}) / y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *substEnv-compose1*:

**assumes** *goodEnv rho and good Y1 and good Y2*

**shows**  $((\rho \&[Y1 / y]\text{-ys}) \&[Y2 / y]\text{-ys}) = (\rho \&[(Y1 \#[Y2 / y]\text{-ys}) / y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *subst-vsubst-compose1*:

**assumes** *good X and good Y and y ≠ y1*

**shows**  $((X \#[y1 // y]\text{-ys}) \#[Y / y]\text{-ys}) = (X \#[y1 // y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *substEnv-vsubstEnv-compose1*:

**assumes** *goodEnv rho and good Y and y ≠ y1*

**shows**  $((\rho \&[y1 // y]\text{-ys}) \&[Y / y]\text{-ys}) = (\rho \&[y1 // y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubst-subst-compose1*:

**assumes** *good X and good Y*

**shows**  $((X \#[Y / y]\text{-ys}) \#[y1 // y]\text{-ys}) = (X \#[(Y \#[y1 // y]\text{-ys}) / y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubstEnv-substEnv-compose1*:

**assumes** *goodEnv rho and good Y*

**shows**  $((\rho \&[Y / y]\text{-ys}) \&[y1 // y]\text{-ys}) = (\rho \&[(Y \#[y1 // y]\text{-ys}) / y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubst-compose1*:

**assumes** *good X*

**shows**  $((X \#[y1 // y]\text{-ys}) \#[y2 // y]\text{-ys}) = (X \#[(y1 @ys[y2 / y]\text{-ys}) // y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubstEnv-compose1*:

**assumes** *goodEnv rho*  
**shows**  $((\text{rho} \ \&[y1 \ // \ y]\text{-ys}) \ \&[y2 \ // \ y]\text{-ys}) = (\text{rho} \ \&[(y1 \ @\text{ys}[y2 \ // \ y]\text{-ys}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *subst-compose2*:  
**assumes** *good X and good Y and good Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh: fresh ys y Z*  
**shows**  $((X \ \#[Y \ // \ y]\text{-ys}) \ \#[Z \ // \ z]\text{-zs}) = ((X \ \#[Z \ // \ z]\text{-zs}) \ \#[(Y \ \#[Z \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *substEnv-compose2*:  
**assumes** *goodEnv rho and good Y and good Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh: fresh ys y Z*  
**shows**  $((\text{rho} \ \&[Y \ // \ y]\text{-ys}) \ \&[Z \ // \ z]\text{-zs}) = ((\text{rho} \ \&[Z \ // \ z]\text{-zs}) \ \&[(Y \ \#[Z \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *subst-vsubst-compose2*:  
**assumes** *good X and good Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh: fresh ys y Z*  
**shows**  $((X \ \#[y1 \ // \ y]\text{-ys}) \ \#[Z \ // \ z]\text{-zs}) = ((X \ \#[Z \ // \ z]\text{-zs}) \ \#[((\text{Var } ys \ y1) \ \#[Z \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *substEnv-vsubstEnv-compose2*:  
**assumes** *goodEnv rho and good Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh: fresh ys y Z*  
**shows**  $((\text{rho} \ \&[y1 \ // \ y]\text{-ys}) \ \&[Z \ // \ z]\text{-zs}) = ((\text{rho} \ \&[Z \ // \ z]\text{-zs}) \ \&[((\text{Var } ys \ y1) \ \#[Z \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubst-subst-compose2*:  
**assumes** *good X and good Y*  
**and**  $ys \neq zs \vee y \notin \{z, z1\}$   
**shows**  $((X \ \#[Y \ // \ y]\text{-ys}) \ \#[z1 \ // \ z]\text{-zs}) = ((X \ \#[z1 \ // \ z]\text{-zs}) \ \#[(Y \ \#[z1 \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubstEnv-substEnv-compose2*:  
**assumes** *goodEnv rho and good Y*  
**and**  $ys \neq zs \vee y \notin \{z, z1\}$   
**shows**  $((\text{rho} \ \&[Y \ // \ y]\text{-ys}) \ \&[z1 \ // \ z]\text{-zs}) = ((\text{rho} \ \&[z1 \ // \ z]\text{-zs}) \ \&[(Y \ \#[z1 \ // \ z]\text{-zs}) \ // \ y]\text{-ys})$   
 $\langle \text{proof} \rangle$

**lemma** *vsubst-compose2*:  
**assumes** *good X*  
**and**  $ys \neq zs \vee y \notin \{z, z1\}$

**shows**  $((X \#[y1 // y]-ys) \#[z1 // z]-zs) =$   
 $((X \#[z1 // z]-zs) \#[(y1 @ys[z1 / z]-zs) // y]-ys)$   
 $\langle proof \rangle$

**lemma** *vsubstEnv-compose2*:

**assumes** *goodEnv rho*

**and**  $ys \neq zs \vee y \notin \{z, z1\}$

**shows**  $((rho \&[y1 // y]-ys) \&[z1 // z]-zs) =$   
 $((rho \&[z1 // z]-zs) \&[(y1 @ys[z1 / z]-zs) // y]-ys)$   
 $\langle proof \rangle$

## 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**

$(good X \longrightarrow$   
 $(\forall z. z \in varsOf P \longrightarrow (X \#[z // z]-zs) = X))$   
 $\wedge$   
 $(goodAbs A \longrightarrow$   
 $(\forall z. z \in varsOf P \longrightarrow (A \$[z // z]-zs) = A))$   
 $\langle proof \rangle$

**corollary** *vsubst-ident[simp]*:

**assumes** *good X*

**shows**  $(X \#[z // z]-zs) = X$

$\langle proof \rangle$

**corollary** *subst-ident[simp]*:

**assumes** *good X*

**shows**  $(X \#[(Var zs z) / z]-zs) = X$

$\langle proof \rangle$

**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**

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

$\langle proof \rangle$

**corollary** *vsubst-eq-swap*:

**assumes** *good*  $X$  **and**  $y1 = y2 \vee \text{fresh } ys \ y1 \ X$   
**shows**  $(X \#[y1 \ / \ / \ y2]-ys) = (X \#[y1 \ \wedge \ y2]-ys)$   
 $\langle proof \rangle$

**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**

$(good \ X \longrightarrow$   
 $(\forall \ y1 \ y2. \ \{y1, y2\} \subseteq varsOf \ P \longrightarrow$   
 $skel \ (X \#[y1 \ / \ / \ y2]-ys) = skel \ X)$   
 $\wedge$   
 $(goodAbs \ A \longrightarrow$   
 $(\forall \ y1 \ y2. \ \{y1, y2\} \subseteq varsOf \ P \longrightarrow$   
 $skelAbs \ (A \ \$[y1 \ / \ / \ y2]-ys) = skelAbs \ A)$   
 $\langle proof \rangle$

**corollary** *skel-vsubst*:

**assumes** *good*  $X$   
**shows**  $skel \ (X \#[y1 \ / \ / \ y2]-ys) = skel \ X$   
 $\langle proof \rangle$

**lemma** *subst-vsubst-trans*:

**assumes** *good*  $X$  **and** *good*  $Y$  **and** *fresh*  $ys \ y1 \ X$   
**shows**  $((X \#[y1 \ / \ / \ y]-ys) \#[Y \ / \ y1]-ys) = (X \#[Y \ / \ y]-ys)$   
 $\langle proof \rangle$

**lemma** *vsubst-trans*:

**assumes** *good*  $X$  **and** *fresh*  $ys \ y1 \ X$   
**shows**  $((X \#[y1 \ / \ / \ y]-ys) \#[y2 \ / \ / \ y1]-ys) = (X \#[y2 \ / \ / \ y]-ys)$   
 $\langle proof \rangle$

**lemma** *vsubst-commute*:

**assumes**  $X$ : *good*  $X$   
**and**  $xs \neq xs' \vee \{x, y\} \cap \{x', y'\} = \{\}$  **and** *fresh*  $xs \ x \ X$  **and** *fresh*  $xs' \ x' \ X$   
**shows**  $((X \#[x \ / \ / \ y]-xs) \#[x' \ / \ / \ y']-xs') = ((X \#[x' \ / \ / \ y']-xs') \#[x \ / \ / \ y]-xs)$   
 $\langle proof \rangle$

## 6.6 Abstraction versions of the properties

Environment identity and update versus other operators:

**lemma** *psubstAbs-idEnv[simp]*:

*goodAbs*  $A \implies (A \ \$[idEnv]) = A$   
 $\langle proof \rangle$

Substitution versus other operators:

**corollary** *freshAbs-psubstAbs*:

**assumes** *goodAbs A* **and** *goodEnv rho*

**shows**

*freshAbs zs z (A \$[rho]) =*

$(\forall ys y. \text{freshAbs } ys y A \vee \text{freshImEnvAt } zs z \text{ rho } ys y)$

*<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*  $\vee$   $(ys \neq zs \vee y \neq z)$

*<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*  $\vee$  *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*  $\implies$  *freshAbs zs z A* **and**

$\bigwedge ys y Y. \text{rho } ys y = \text{Some } Y \implies \text{freshAbs } ys y A \vee \text{fresh } zs z Y$

**shows** *freshAbs zs z (A \$[rho])*

*<proof>*

**lemma** *freshAbs-substAbs*:

**assumes** *goodAbs A* **and** *good Y*

**shows** *freshAbs zs z (A \$[Y / y]-ys) =*

$((zs = ys \wedge z = y) \vee \text{freshAbs } zs z A) \wedge (\text{freshAbs } ys y A \vee \text{fresh } zs z Y)$

*<proof>*

**lemma** *freshAbs-vsubstAbs*:

**assumes** *goodAbs A*

**shows** *freshAbs zs z (A \$[y1 // y]-ys) =*

$((zs = ys \wedge z = y) \vee \text{freshAbs } zs z A) \wedge$   
 $(\text{freshAbs } ys y A \vee (zs \neq ys \vee z \neq y1))$

*<proof>*

**lemma** *substAbs-preserves-freshAbs*:

**assumes** *goodAbs A* **and** *good Y*

**and** *freshAbs zs z A* **and** *fresh zs z Y*

**shows**  $\text{freshAbs } zs \ z \ (A \ \$[Y / y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{vsubstAbs-preserves-freshAbs}$ :  
**assumes**  $\text{goodAbs } A$   
**and**  $\text{freshAbs } zs \ z \ A$  **and**  $zs \neq ys \vee z \neq y1$   
**shows**  $\text{freshAbs } zs \ z \ (A \ \$[y1 // y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{fresh-freshAbs-substAbs[simp]}$ :  
**assumes**  $\text{good } Y$  **and**  $\text{goodAbs } A$   
**and**  $\text{fresh } ys \ y \ Y$   
**shows**  $\text{freshAbs } ys \ y \ (A \ \$[Y / y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{diff-freshAbs-vsubstAbs[simp]}$ :  
**assumes**  $\text{goodAbs } A$   
**and**  $y \neq y1$   
**shows**  $\text{freshAbs } ys \ y \ (A \ \$[y1 // y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

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

**lemma**  $\text{freshAbs-vsubstAbs-E1}$ :  
**assumes**  $\text{goodAbs } A$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[y1 // y]\text{-}ys)$  **and**  $zs \neq ys \vee z \neq y$   
**shows**  $\text{freshAbs } zs \ z \ A$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{freshAbs-substAbs-E2}$ :  
**assumes**  $\text{goodAbs } A$  **and**  $\text{good } Y$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[Y / y]\text{-}ys)$   
**shows**  $\text{freshAbs } ys \ y \ A \vee \text{fresh } zs \ z \ Y$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{freshAbs-vsubstAbs-E2}$ :  
**assumes**  $\text{goodAbs } A$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[y1 // y]\text{-}ys)$   
**shows**  $\text{freshAbs } ys \ y \ A \vee zs \neq ys \vee z \neq y1$   
 $\langle \text{proof} \rangle$

**corollary**  $\text{psubstAbs-cong[fundef-cong]}$ :  
**assumes**  $\text{goodAbs } A$  **and**  $\text{goodEnv } rho$  **and**  $\text{goodEnv } rho'$   
**and**  $\bigwedge ys \ y. \text{freshAbs } ys \ y \ A \vee rho \ ys \ y = rho' \ ys \ y$   
**shows**  $(A \ \$[rho]) = (A \ \$[rho'])$

$\langle proof \rangle$

**lemma** *freshAbs-psubstAbs-updEnv*:  
assumes *good X and goodAbs A and goodEnv rho*  
and *freshAbs xs x A*  
shows  $(A \ \$[rho \ [x \leftarrow X]-xs]) = (A \ \$[rho])$   
 $\langle proof \rangle$

**corollary** *freshEnv-psubstAbs-ident[simp]*:  
fixes  $A :: ('index, 'bindex, 'varSort, 'var, 'opSym)abs$   
assumes *goodAbs A and goodEnv rho*  
and  $\bigwedge z s z. \text{freshEnv } z s z \ \rho \vee \text{freshAbs } z s z \ A$   
shows  $(A \ \$[rho]) = A$   
 $\langle proof \rangle$

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

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

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

**lemma** *swapAbs-psubstAbs*:  
assumes *goodAbs A and goodEnv rho*  
shows  $((A \ \$[rho]) \ \$[z1 \ \wedge \ z2]-zs) = ((A \ \$[z1 \ \wedge \ z2]-zs) \ \$[rho \ \&[z1 \ \wedge \ z2]-zs])$   
 $\langle proof \rangle$

**lemma** *swapAbs-substAbs*:  
assumes *goodAbs A and good Y*  
shows  $((A \ \$[Y \ / \ y]-ys) \ \$[z1 \ \wedge \ z2]-zs) =$   
 $((A \ \$[z1 \ \wedge \ z2]-zs) \ \$[(Y \ \#[z1 \ \wedge \ z2]-zs) \ / \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**lemma** *swapAbs-vsubstAbs*:  
assumes *goodAbs A*  
shows  $((A \ \$[y1 \ /\ / \ y]-ys) \ \$[z1 \ \wedge \ z2]-zs) =$   
 $((A \ \$[z1 \ \wedge \ z2]-zs) \ \$[(y1 \ @ys[z1 \ \wedge \ z2]-zs) \ /\ / \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

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



**shows**  $((A \ \$[rho]) \ \$[rho']) = (A \ \$[(rho \ \&[rho')])$   
 $\langle proof \rangle$

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

**lemma** *psubstAbs-usbstAbs-compose*:  
**assumes** *goodAbs A and goodEnv rho*  
**shows**  $((A \ \$[y1 \ /\ y]-ys) \ \$[rho]) = (A \ \$[(rho \ [y \ \leftarrow \ ((Var \ ys \ y1) \ \#[rho])]-ys)])$   
 $\langle proof \rangle$

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

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

**lemma** *usbstAbs-psubstAbs-compose*:  
**assumes** *goodAbs A and goodEnv rho*  
**shows**  $((A \ \$[rho]) \ \$[y1 \ /\ y]-ys) = (A \ \$[(rho \ \&[y1 \ /\ y]-ys)])$   
 $\langle proof \rangle$

**lemma** *substAbs-compose1*:  
**assumes** *goodAbs A and good Y1 and good Y2*  
**shows**  $((A \ \$[Y1 \ / \ y]-ys) \ \$[Y2 \ / \ y]-ys) = (A \ \$[(Y1 \ \#[Y2 \ / \ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**lemma** *substAbs-usbstAbs-compose1*:  
**assumes** *goodAbs A and good Y and y  $\neq$  y1*  
**shows**  $((A \ \$[y1 \ /\ y]-ys) \ \$[Y \ / \ y]-ys) = (A \ \$[y1 \ /\ y]-ys)$   
 $\langle proof \rangle$

**lemma** *usbstAbs-substAbs-compose1*:  
**assumes** *goodAbs A and good Y*  
**shows**  $((A \ \$[Y \ / \ y]-ys) \ \$[y1 \ /\ y]-ys) = (A \ \$[(Y \ \#[y1 \ /\ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**lemma** *usbstAbs-compose1*:  
**assumes** *goodAbs A*  
**shows**  $((A \ \$[y1 \ /\ y]-ys) \ \$[y2 \ /\ y]-ys) = (A \ \$[(y1 \ @ys[y2 \ / \ y]-ys) \ /\ y]-ys)$   
 $\langle proof \rangle$

**lemma** *substAbs-compose2*:

**assumes** *goodAbs A and good Y and good Z*

**and**  $ys \neq zs \vee y \neq z$  **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>*

**lemma** *substAbs-vsubstAbs-compose2*:

**assumes** *goodAbs A and good Z*

**and**  $ys \neq zs \vee y \neq z$  **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>*

**lemma** *vsubstAbs-substAbs-compose2*:

**assumes** *goodAbs A and good Y*

**and**  $ys \neq zs \vee y \notin \{z, z1\}$

**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**  $ys \neq zs \vee y \notin \{z, z1\}$

**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**  $(A \ \$[y1 // y2]-ys) = (A \ \$[y1 \wedge y2]-ys)$   
*<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**  $((A \text{ \$}[y1 \ // \ y]-ys) \text{ \$}[Y / y1]-ys) = (A \text{ \$}[Y / y]-ys)$   
*<proof>*

**lemma** *vsubstAbs-trans:*

**assumes** *goodAbs A and freshAbs ys y1 A*  
**shows**  $((A \text{ \$}[y1 \ // \ y]-ys) \text{ \$}[y2 \ // \ y1]-ys) = (A \text{ \$}[y2 \ // \ y]-ys)$   
*<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-allOpsers-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-allOpsers-simps*  
*good-allOpsers-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) <sup>1</sup> – 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, 'binde $x$ , 'varSort, 'sort, 'opSym)signature =
  varSortAsSort :: 'varSort  $\Rightarrow$  'sort
  wlsOpSym :: 'opSym  $\Rightarrow$  bool
  sortOf :: 'opSym  $\Rightarrow$  'sort
  arityOf :: 'opSym  $\Rightarrow$  ('index, 'sort)input
  barityOf :: 'opSym  $\Rightarrow$  ('binde $x$ , '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 ==
 $\forall$  delta.
```

```
wlsOpSym Delta delta  $\longrightarrow$  |\{i. arityOf Delta delta i  $\neq$  None\}| < o |UNIV :: 'var set|
```

**definition** *barityOf-lt-var* **where**

---

<sup>1</sup>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  $\Rightarrow$   
 ('varSort \* 'sort)  $\Rightarrow$  ('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  $\Rightarrow$  ('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 ==  $\lambda i.$  None

**abbreviation** empAr :: ('index,'sort)input  
**where** empAr ==  $\lambda i.$  None

**abbreviation** empBinp :: ('bindex,('index,'bindex,'varSort,'var,'opSym)abs)input  
**where** empBinp ==  $\lambda i.$  None

**abbreviation** empBar :: ('bindex,'varSort \* 'sort)input  
**where** empBar ==  $\lambda i.$  None

**lemma** freshInp-empInp[*simp*]:  
 freshInp xs x empInp  
 {proof}

**lemma** *swapInp-empInp[simp]*:  
 $(empInp \%[x1 \wedge x2]-xs) = empInp$   
 $\langle proof \rangle$

**lemma** *psubstInp-empInp[simp]*:  
 $(empInp \%[rho]) = empInp$   
 $\langle proof \rangle$

**lemma** *substInp-empInp[simp]*:  
 $(empInp \%[Y / y]-ys) = empInp$   
 $\langle proof \rangle$

**lemma** *vsubstInp-empInp[simp]*:  
 $(empInp \%[y1 // y]-ys) = empInp$   
 $\langle proof \rangle$

**lemma** *freshBinp-empBinp[simp]*:  
 $freshBinp\ xs\ x\ empBinp$   
 $\langle proof \rangle$

**lemma** *swapBinp-empBinp[simp]*:  
 $(empBinp \%[x1 \wedge x2]-xs) = empBinp$   
 $\langle proof \rangle$

**lemma** *psubstBinp-empBinp[simp]*:  
 $(empBinp \%[rho]) = empBinp$   
 $\langle proof \rangle$

**lemma** *substBinp-empBinp[simp]*:  
 $(empBinp \%[Y / y]-ys) = empBinp$   
 $\langle proof \rangle$

**lemma** *vsubstBinp-empBinp[simp]*:  
 $(empBinp \%[y1 // y]-ys) = empBinp$   
 $\langle proof \rangle$

**lemmas** *empInp-simps =*  
 $freshInp-empInp\ swapInp-empInp\ psubstInp-empInp\ substInp-empInp\ vsubstInp-empInp$   
 $freshBinp-empBinp\ swapBinp-empBinp\ psubstBinp-empBinp\ substBinp-empBinp\ vsub-$   
 $stBinp-empBinp$

### 7.3.4 Inner versions of the locale assumptions

**lemma** *varSortAsSort-inj-INNER: inj asSort*  
 $\langle proof \rangle$

**lemma** *asSort-inj[simp]*:  
 $(asSort\ xs = asSort\ ys) = (xs = ys)$   
 $\langle proof \rangle$

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

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

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

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

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

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

### 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 ==*  
 $\exists \text{ delta } i. \text{wlsOpS delta} \wedge \text{barOf delta } i = \text{Some } xs-s$

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*:  $\llbracket wlsInp \text{ delta } inp; wlsBinp \text{ delta } binp \rrbracket \implies wls \text{ (stOf delta) (Op delta inp binp)}$   
 |  
*Inp*:  
 $\llbracket wlsOpS \text{ delta};$   
 $\bigwedge i. (arOf \text{ delta } i = None \wedge inp \text{ } i = None) \vee$   
 $(\exists s \ X. arOf \text{ delta } i = Some \ s \wedge inp \text{ } i = Some \ X \wedge wls \ s \ X) \rrbracket$   
 $\implies wlsInp \text{ delta } inp$   
 |  
*Binp*:  
 $\llbracket wlsOpS \text{ delta};$   
 $\bigwedge i. (barOf \text{ delta } i = None \wedge binp \text{ } i = None) \vee$   
 $(\exists us \ s \ A. barOf \text{ delta } i = Some \ (us, s) \wedge binp \text{ } i = Some \ A \wedge wlsAbs \ (us, s)$   
 $A) \rrbracket$   
 $\implies wlsBinp \text{ delta } binp$   
 |  
*Abs*:  $\llbracket isInBar \ (xs, s); wls \ s \ X \rrbracket \implies wlsAbs \ (xs, s) \ (Abs \ xs \ x \ X)$

**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* ::  $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'var}, \text{'opSym})env \Rightarrow bool$

**where**

*wlsEnv rho* ==

$(\forall ys. \text{liftAll} \ (wls \ (asSort \ ys)) \ (rho \ ys)) \wedge$

$(\forall ys. |\{y. rho \ ys \ y \neq None\}| < o \ |UNIV :: \text{'var set}|)$

**definition** *wlsPar* ::  $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'var}, \text{'opSym}, \text{'sort})paramS \Rightarrow bool$

**where**

*wlsPar P* ==

$(\forall s. \forall X \in \text{termsOfS} \ P \ s. wls \ s \ X) \wedge$

$(\forall xs \ s. \forall A \in \text{absOfS} \ P \ (xs, s). wlsAbs \ (xs, s) \ A) \wedge$

$(\forall rho \in \text{envsOfS} \ P. wlsEnv \ rho)$

**lemma** *ParS-preserves-wls[simp]*:

**assumes**  $\bigwedge s \ X. X \in \text{set} \ (XLF \ s) \implies wls \ s \ X$

**and**  $\bigwedge xs\ s\ A. A \in set\ (ALF\ (xs,s)) \implies wlsAbs\ (xs,s)\ A$   
**and**  $\bigwedge rho. rho \in set\ rhoF \implies wlsEnv\ rho$   
**shows**  $wlsPar\ (ParS\ xLF\ XLF\ ALF\ rhoF)$   
 $\langle proof \rangle$

**lemma** *termsOfS-preserves-wls[simp]*:  
**assumes**  $wlsPar\ P$  **and**  $X : termsOfS\ P\ s$   
**shows**  $wls\ s\ X$   
 $\langle proof \rangle$

**lemma** *absOfS-preserves-wls[simp]*:  
**assumes**  $wlsPar\ P$  **and**  $A : absOfS\ P\ (us,s)$   
**shows**  $wlsAbs\ (us,s)\ A$   
 $\langle proof \rangle$

**lemma** *envsOfS-preserves-wls[simp]*:  
**assumes**  $wlsPar\ P$  **and**  $rho : envsOfS\ P$   
**shows**  $wlsEnv\ rho$   
 $\langle proof \rangle$

**lemma** *not-isInBar-absOfS-empty[simp]*:  
**assumes**  $*$ :  $\neg\ isInBar\ (us,s)$  **and**  $**$ :  $wlsPar\ P$   
**shows**  $absOfS\ P\ (us,s) = \{\}$   
 $\langle proof \rangle$

**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*:  
 $wlsInp\ delta\ inp =$   
 $(wlsOpS\ delta \wedge sameDom\ (arOf\ delta)\ inp \wedge liftAll2\ wls\ (arOf\ delta)\ inp)$   
 $\langle proof \rangle$

**lemma** *wlsBinp-iff*:  
 $wlsBinp\ delta\ binp =$   
 $(wlsOpS\ delta \wedge sameDom\ (barOf\ delta)\ binp \wedge liftAll2\ wlsAbs\ (barOf\ delta)\ binp)$   
 $\langle proof \rangle$

**lemma** *exists-asSort-wls*:  
 $\exists X. wls\ (asSort\ xs)\ X$   
 $\langle proof \rangle$

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

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

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

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-emptiness. "witOpS" stands for "witness operation symbol".

**definition** *witOpS* **where**  
 $witOpS s delta R ==$   
 $wlsOpS delta \wedge stOf delta = s \wedge$   
 $liftAll (\lambda s'. (s',s) : R) (arOf delta) \wedge$   
 $liftAll (\lambda (us,s'). (s',s) : R) (barOf delta)$

**lemma** *wf-exists-wls*:  
**assumes**  $wf: wf R$  **and** \*:  $\bigwedge s. (\exists xs. s = asSort xs) \vee witOpS s (f s) R$   
**shows**  $\exists X. wls s X$   
 $\langle proof \rangle$

**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**  $\exists A. wlsAbs (us,s) A$   
 $\langle proof \rangle$

### 7.3.7 Well-sorted implies Good

**lemma** *wlsInp-empAr-empInp[simp]*:  
**assumes**  $wlsOpS delta$  **and**  $arOf delta = empAr$   
**shows**  $wlsInp delta empInp$   
 $\langle proof \rangle$

**lemma** *wlsBinp-empBar-empBinp[simp]*:  
**assumes**  $wlsOpS delta$  **and**  $barOf delta = empBar$   
**shows**  $wlsBinp delta empBinp$   
 $\langle proof \rangle$

**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*  $\longrightarrow$  *goodBinp binp*)  
 ⟨*proof*⟩

**corollary** *wls-imp-good[simp]*: *wls s X*  $\implies$  *good X*  
 ⟨*proof*⟩

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

**corollary** *wlsInp-imp-goodInp[simp]*: *wlsInp delta inp*  $\implies$  *goodInp inp*  
 ⟨*proof*⟩

**corollary** *wlsBinp-imp-goodBinp[simp]*: *wlsBinp delta binp*  $\implies$  *goodBinp binp*  
 ⟨*proof*⟩

**lemma** *wlsEnv-imp-goodEnv[simp]*: *wlsEnv rho*  $\implies$  *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*  $\longrightarrow$  *wls s (X #[z1  $\wedge$  z2]-zs)*)  $\wedge$   
 (*wlsAbs (xs,s') A*  $\longrightarrow$  *wlsAbs (xs,s') (A \$[z1  $\wedge$  z2]-zs)*)  $\wedge$   
 (*wlsInp delta inp*  $\longrightarrow$  *wlsInp delta (inp %[z1  $\wedge$  z2]-zs)*)  $\wedge$   
 (*wlsBinp delta binp*  $\longrightarrow$  *wlsBinp delta (binp %%[z1  $\wedge$  z2]-zs)*)  
 ⟨*proof*⟩

**lemma** *swap-preserves-wls[simp]*:  
*wls s X*  $\implies$  *wls s (X #[z1  $\wedge$  z2]-zs)*  
 ⟨*proof*⟩

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

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

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

*<proof>*

**lemma** *swapInp-preserves-wls[simp]*:

*wlsInp delta inp  $\implies$  wlsInp delta (inp % $[z1 \wedge z2]$ -zs)*

*<proof>*

**lemma** *swapBinp-preserves-wls[simp]*:

*wlsBinp delta binp  $\implies$  wlsBinp delta (binp % $[z1 \wedge z2]$ -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 & $[z1 \wedge z2]$ -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  $\longrightarrow$*

*( $\forall$  xs x. X = Var xs x  $\longrightarrow$  s = asSort xs)  $\wedge$*

*( $\forall$  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))*

*$\wedge$*

*(wlsAbs xs-s A  $\longrightarrow$*

*isInBar xs-s  $\wedge$*

*( $\forall$  x X. good X  $\wedge$  A = Abs (fst xs-s) x X  $\longrightarrow$*

*wls (snd xs-s) X))*

*$\wedge$*

*(wlsInp delta inp  $\longrightarrow$  True)*

$\wedge$   
 $(wlsBinp\ delta\ binp \longrightarrow True)$   
 $\langle proof \rangle$

**lemma** *conjLeft*:  $\llbracket phi1 \wedge phi2; phi1 \implies chi \rrbracket \implies chi$   
 $\langle proof \rangle$

**lemma** *conjRight*:  $\llbracket phi1 \wedge phi2; phi2 \implies chi \rrbracket \implies chi$   
 $\langle proof \rangle$

**lemma** *wls-inversion*[*rule-format*]:

$wls\ s\ X \longrightarrow$   
 $(\forall\ xs\ x.\ X = Var\ xs\ x \longrightarrow s = asSort\ xs) \wedge$   
 $(\forall\ delta\ inp\ binp.\ goodInp\ inp \wedge goodBinp\ binp \wedge X = Op\ delta\ inp\ binp \longrightarrow$   
 $stOf\ delta = s \wedge wlsInp\ delta\ inp \wedge wlsBinp\ delta\ binp)$   
 $\langle proof \rangle$

**lemma** *wlsAbs-inversion*[*rule-format*]:

$wlsAbs\ (xs,s)\ A \longrightarrow$   
 $isInBar\ (xs,s) \wedge$   
 $(\forall\ x\ X.\ good\ X \wedge A = Abs\ xs\ x\ X \longrightarrow wls\ s\ X)$   
 $\langle proof \rangle$

**lemma** *wls-Var-simp*[*simp*]:

$wls\ s\ (Var\ xs\ x) = (s = asSort\ xs)$   
 $\langle proof \rangle$

**lemma** *wls-Op-simp*[*simp*]:

**assumes** *goodInp inp and goodBinp binp*  
**shows**  
 $wls\ s\ (Op\ delta\ inp\ binp) =$   
 $(stOf\ delta = s \wedge wlsInp\ delta\ inp \wedge wlsBinp\ delta\ binp)$   
 $\langle proof \rangle$

**lemma** *wls-Abs-simp*[*simp*]:

**assumes** *good X*  
**shows**  $wlsAbs\ (xs,s)\ (Abs\ xs\ x\ X) = (isInBar\ (xs,s) \wedge wls\ s\ X)$   
 $\langle proof \rangle$

**lemma** *wlsAll-inversion2*:

$(wls\ s\ X \longrightarrow True)$   
 $\wedge$   
 $(wlsAbs\ xs-s\ A \longrightarrow$   
 $isInBar\ xs-s \wedge$   
 $(\exists\ x\ X.\ wls\ (snd\ xs-s)\ X \wedge A = Abs\ (fst\ xs-s)\ x\ X))$   
 $\wedge$   
 $(wlsInp\ delta\ inp \longrightarrow True)$

$\wedge$   
*(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.  $\llbracket$ wls s X; (X,Y)  $\in$  rel  $s$  $\rrbracket \implies$  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.  
 $\llbracket$ wlsInp delta inp; wlsBinp delta binp;  
 liftAll2 phi (arOf delta) inp; liftAll2 phiAbs (barOf delta) binp $\rrbracket$   
 $\implies$  phi (stOf delta) (Op delta inp binp) **and**  
*Abs:*  
 $\bigwedge$  s xs x X.  
 $\llbracket$ isInBar (xs,s); wls s X;  $\bigwedge$  Y. (X,Y)  $\in$  rel s  $\implies$  phi s Y $\rrbracket$   
 $\implies$  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*:  $\bigwedge xs\ x. \text{phi} (\text{asSort } xs) (\text{Var } xs\ x)$  **and**

*Op*:

$\bigwedge \text{delta inp binp.}$

$\llbracket \text{wlsInp delta inp; wlsBinp delta binp;} \\ \text{liftAll2 phi (arOf delta) inp; liftAll2 phiAbs (barOf delta) binp} \rrbracket \\ \implies \text{phi (stOf delta) (Op delta inp binp)}$  **and**

*Abs*:  $\bigwedge s\ xs\ x\ X. \llbracket \text{isInBar } (xs,s); \text{wls } s\ X; \text{phi } s\ X \rrbracket \implies \text{phiAbs } (xs,s) (\text{Abs } xs\ x\ X)$

**shows**

$(\text{wls } s\ X \longrightarrow \text{phi } s\ X) \wedge \\ (\text{wlsAbs } (xs,s')\ A \longrightarrow \text{phiAbs } (xs,s')\ A) \\ \langle \text{proof} \rangle$

## 7.4.2 Fresh induction

First for an unspecified notion of parameter:

**theorem** *wls-templateInduct-fresh*[*case-names Par Rel Var Op Abs*]:

**fixes**  $s\ X\ xs\ s'\ A\ \text{phi}\ \text{phiAbs}\ \text{rel}$

**and**  $\text{vars} :: 'var\ \text{set} \Rightarrow 'var\ \text{set}$

**and**  $\text{terms} :: 'sort \Rightarrow ('index, 'bindex, 'var\ \text{Sort}, 'var, 'opSym)\ \text{term}\ \text{set}$

**and**  $\text{abs} :: ('var\ \text{Sort} * 'sort) \Rightarrow ('index, 'bindex, 'var\ \text{Sort}, 'var, 'opSym)\ \text{abs}\ \text{set}$

**and**  $\text{envs} :: ('index, 'bindex, 'var\ \text{Sort}, 'var, 'opSym)\ \text{env}\ \text{set}$

**assumes**

*PAR*:

$\bigwedge xs\ us\ s.$

$(|\text{vars } xs| < o\ |\text{UNIV} :: 'var\ \text{set}| \vee \text{finite } (\text{vars } xs)) \wedge \\ (|\text{terms } s| < o\ |\text{UNIV} :: 'var\ \text{set}| \vee \text{finite } (\text{terms } s)) \wedge \\ (|\text{abs } (us,s)| < o\ |\text{UNIV} :: 'var\ \text{set}| \vee \text{finite } (\text{abs } (us,s))) \wedge \\ (\forall X \in \text{terms } s. \text{wls } s\ X) \wedge \\ (\forall A \in \text{abs } (us,s). \text{wlsAbs } (us,s)\ A) \wedge \\ (|\text{envs}| < o\ |\text{UNIV} :: 'var\ \text{set}| \vee \text{finite } (\text{envs})) \wedge \\ (\forall rho \in \text{envs}. \text{wlsEnv } rho)$  **and**

*rel*:  $\bigwedge s\ X\ Y. \llbracket \text{wls } s\ X; (X,Y) \in \text{rel } s \rrbracket \implies \text{wls } s\ Y \wedge \text{skel } Y = \text{skel } X$  **and**

*Var*:  $\bigwedge xs\ x. \text{phi} (\text{asSort } xs) (\text{Var } xs\ x)$  **and**

*Op*:

$\bigwedge \text{delta inp binp.}$

$\llbracket \text{wlsInp delta inp; wlsBinp delta binp;} \\ \text{liftAll2 } (\lambda s\ X. \text{phi } s\ X) (\text{arOf delta) inp;} \\ \text{liftAll2 } (\lambda (us,s)\ A. \text{phiAbs } (us,s)\ A) (\text{barOf delta) binp} \rrbracket \\ \implies \text{phi (stOf delta) (Op delta inp binp)}$  **and**

*Abs*:

$\bigwedge s\ xs\ x\ X.$

$\llbracket \text{isInBar } (xs,s); \text{wls } s\ X; \\ x \notin \text{vars } xs; \\ \bigwedge s'\ Y. Y \in \text{terms } s' \implies \text{fresh } xs\ x\ Y; \\ \bigwedge xs'\ s'\ A. A \in \text{abs } (xs',s') \implies \text{freshAbs } xs\ x\ A; \\ \bigwedge rho. rho \in \text{envs} \implies \text{freshEnv } xs\ x\ rho; \\ \bigwedge Y. (X,Y) \in \text{rel } s \implies \text{phi } s\ Y \rrbracket \\ \implies \text{phiAbs } (xs,s) (\text{Abs } xs\ x\ X)$



**shows**

$(wls\ s\ X \longrightarrow phi\ s\ X) \wedge$   
 $(wlsAbs\ (xs,s')\ A \longrightarrow phiAbs\ (xs,s')\ A)$   
 $\langle proof \rangle$

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**  $vars :: 'varSort \Rightarrow 'var\ set$

**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*:

$\bigwedge xs\ us\ s.$

$(|vars\ xs| < o\ |UNIV :: 'var\ set| \vee finite\ (vars\ xs)) \wedge$

$(|terms\ s| < o\ |UNIV :: 'var\ set| \vee finite\ (terms\ s)) \wedge$

$(\forall X \in terms\ s. wls\ s\ X) \wedge$

$(|abs\ (us,s)| < o\ |UNIV :: 'var\ set| \vee finite\ (abs\ (us,s))) \wedge$

$(\forall A \in abs\ (us,s). wlsAbs\ (us,s)\ A) \wedge$

$(|envs| < o\ |UNIV :: 'var\ set| \vee finite\ (envs)) \wedge$

$(\forall rho \in envs. wlsEnv\ rho)$  **and**

$Var: \bigwedge xs\ x. phi\ (asSort\ xs)\ (Var\ xs\ x)$  **and**

*Op*:

$\bigwedge delta\ inp\ binp.$

$\llbracket wlsInp\ delta\ inp; wlsBinp\ delta\ binp;$

$liftAll2\ (\lambda s\ X. phi\ s\ X)\ (arOf\ delta)\ inp;$

$liftAll2\ (\lambda(us,s)\ A. phiAbs\ (us,s)\ A)\ (barOf\ delta)\ binp \rrbracket$

$\implies phi\ (stOf\ delta)\ (Op\ delta\ inp\ binp)$  **and**

*Abs*:

$\bigwedge s\ xs\ x\ X.$

$\llbracket isInBar\ (xs,s); wls\ s\ X;$

$x \notin vars\ xs;$

$\bigwedge s'\ Y. Y \in terms\ s' \implies fresh\ xs\ x\ Y;$

$\bigwedge us\ s'\ A. A \in abs\ (us,s') \implies freshAbs\ xs\ x\ A;$

$\bigwedge rho. rho \in envs \implies freshEnv\ xs\ x\ rho;$

$phi\ s\ X \rrbracket$

$\implies phiAbs\ (xs,s)\ (Abs\ xs\ x\ X)$

**shows**

$(wls\ s\ X \longrightarrow phi\ s\ X) \wedge$

$(wlsAbs\ (xs,s')\ A \longrightarrow phiAbs\ (xs,s')\ A)$

$\langle proof \rangle$

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*:  $\bigwedge xs\ x.\ \text{phi}\ (\text{asSort}\ xs)\ (\text{Var}\ xs\ x)$  **and**  
*Op*:  
 $\bigwedge\ \text{delta}\ \text{inp}\ \text{binp}.$   
 $\llbracket \text{wlsInp}\ \text{delta}\ \text{inp};\ \text{wlsBinp}\ \text{delta}\ \text{binp};$   
 $\text{liftAll2}\ (\lambda s\ X.\ \text{phi}\ s\ X)\ (\text{arOf}\ \text{delta})\ \text{inp};$   
 $\text{liftAll2}\ (\lambda(us,s)\ A.\ \text{phiAbs}\ (us,s)\ A)\ (\text{barOf}\ \text{delta})\ \text{binp} \rrbracket$   
 $\implies \text{phi}\ (\text{stOf}\ \text{delta})\ (\text{Op}\ \text{delta}\ \text{inp}\ \text{binp})$  **and**  
*Abs*:  
 $\bigwedge\ s\ xs\ x\ X.$   
 $\llbracket \text{isInBar}\ (xs,s);\ \text{wls}\ s\ X;$   
 $x \notin \text{varsOfS}\ P\ xs;$   
 $\bigwedge\ s'\ Y.\ Y \in \text{termsOfS}\ P\ s' \implies \text{fresh}\ xs\ x\ Y;$   
 $\bigwedge\ us\ s'\ A.\ A \in \text{absOfS}\ P\ (us,s') \implies \text{freshAbs}\ xs\ x\ A;$   
 $\bigwedge\ rho.\ rho \in \text{envsOfS}\ P \implies \text{freshEnv}\ xs\ x\ rho;$   
 $\text{phi}\ s\ X \rrbracket$   
 $\implies \text{phiAbs}\ (xs,s)\ (\text{Abs}\ xs\ x\ X)$

**shows**  
 $(\text{wls}\ s\ X \longrightarrow \text{phi}\ s\ X) \wedge$   
 $(\text{wlsAbs}\ (xs,s')\ A \longrightarrow \text{phiAbs}\ (xs,s')\ A)$   
 $\langle \text{proof} \rangle$

### 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**  
 $(\text{Op}\ \text{delta}\ \text{inp}\ \text{binp} = \text{Op}\ \text{delta}'\ \text{inp}'\ \text{binp}') =$   
 $(\text{delta} = \text{delta}' \wedge \text{inp} = \text{inp}' \wedge \text{binp} = \text{binp}')$   
 $\langle \text{proof} \rangle$

**lemma** *wls-Abs-ainj-all*:  
**assumes** *wls s X and wls s' X'*  
**shows**  
 $(\text{Abs}\ xs\ x\ X = \text{Abs}\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\forall y.\ (y = x \vee \text{fresh}\ xs\ y\ X) \wedge (y = x' \vee \text{fresh}\ xs\ y\ X') \longrightarrow$   
 $(X \#[y \wedge x]-xs) = (X' \#[y \wedge x']-xs)))$   
 $\langle \text{proof} \rangle$

**theorem** *wls-Abs-swap-all*:  
**assumes** *wls s X and wls s' X'*  
**shows**  
 $(\text{Abs}\ xs\ x\ X = \text{Abs}\ xs\ x'\ X') =$   
 $(\forall y.\ (y = x \vee \text{fresh}\ xs\ y\ X) \wedge (y = x' \vee \text{fresh}\ xs\ y\ X') \longrightarrow$   
 $(X \#[y \wedge x]-xs) = (X' \#[y \wedge x']-xs))$   
 $\langle \text{proof} \rangle$

**lemma** *wls-Abs-ainj-ex*:

**assumes** *wls s X and wls s X'*

**shows**

$(Abs\ xs\ x\ X = Abs\ xs'\ x'\ X') =$   
 $(xs = xs' \wedge$   
 $(\exists\ y. y \notin \{x, x'\} \wedge fresh\ xs\ y\ X \wedge fresh\ xs\ y\ X' \wedge$   
 $(X \#[y \wedge x]-xs) = (X' \#[y \wedge x']-xs)))$   
*<proof>*

**theorem** *wls-Abs-swap-ex*:

**assumes** *wls s X and wls s X'*

**shows**

$(Abs\ xs\ x\ X = Abs\ xs\ x'\ X') =$   
 $(\exists\ y. y \notin \{x, x'\} \wedge fresh\ xs\ y\ X \wedge fresh\ xs\ y\ X' \wedge$   
 $(X \#[y \wedge x]-xs) = (X' \#[y \wedge x']-xs))$   
*<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*  $(X \#[y \wedge x]-xs) = (X' \#[y \wedge x']-xs)$

**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**  $Abs\ xs\ x'\ (X \#[x' \wedge x]-xs) = Abs\ xs\ x\ X$

*<proof>*

**theorem** *wls-Var-diff-Op[simp]*:

**assumes** *wlsInp delta inp and wlsBinp delta binp*

**shows**  $Var\ xs\ x \neq Op\ delta\ inp\ binp$

*<proof>*

**theorem** *wls-Op-diff-Var[simp]*:

**assumes** *wlsInp delta inp and wlsBinp delta binp*

**shows**  $Op\ delta\ inp\ binp \neq Var\ xs\ x$

*<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**  $\bigwedge x\ X. \llbracket isInBar\ (xs,s); wls\ s\ X \rrbracket \implies phiAbs\ (xs,s)\ (Abs\ xs\ x\ X)$   
**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*  $\leftarrow$  *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**

$(wls s X \longrightarrow (\forall rho \in envsOfS P. wls s (X \#[rho]))) \wedge$   
 $(wlsAbs (xs,s') A \longrightarrow (\forall rho \in envsOfS P. wlsAbs (xs,s') (A \$[rho])))$   
 $\langle proof \rangle$

**lemma**  $psubst-preserves-wls[simp]$ :

$\llbracket wls s X; wlsEnv rho \rrbracket \Longrightarrow wls s (X \#[rho])$   
 $\langle proof \rangle$

**lemma**  $psubstAbs-preserves-wls[simp]$ :

$\llbracket wlsAbs (xs,s) A; wlsEnv rho \rrbracket \Longrightarrow wlsAbs (xs,s) (A \$[rho])$   
 $\langle proof \rangle$

**lemma**  $psubstInp-preserves-wls[simp]$ :

**assumes**  $wlsInp delta inp$  **and**  $wlsEnv rho$

**shows**  $wlsInp delta (inp \%[rho])$   
 $\langle proof \rangle$

**lemma**  $psubstBinp-preserves-wls[simp]$ :

**assumes**  $wlsBinp delta binp$  **and**  $wlsEnv rho$

**shows**  $wlsBinp delta (binp \%[rho])$   
 $\langle proof \rangle$

**lemma**  $psubstEnv-preserves-wls[simp]$ :

**assumes**  $wlsEnv rho$  **and**  $wlsEnv rho'$

**shows**  $wlsEnv (rho \&[rho'])$   
 $\langle proof \rangle$

**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**  $wls s (X \#[Y / y]-ys)$   
 $\langle proof \rangle$

**lemma**  $substAbs-preserves-wls[simp]$ :

**assumes**  $wlsAbs (xs,s) A$  **and**  $wls (asSort ys) Y$

**shows**  $wlsAbs (xs,s) (A \$[Y / y]-ys)$   
 $\langle proof \rangle$

**lemma**  $substInp-preserves-wls[simp]$ :

**assumes**  $wlsInp delta inp$  **and**  $wls (asSort ys) Y$

**shows**  $wlsInp delta (inp \%[Y / y]-ys)$   
 $\langle proof \rangle$

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

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

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

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

**lemma** *vsubstInp-preserves-wls[simp]*:  
**assumes** *wlsInp delta inp*  
**shows** *wlsInp delta (inp %[x1 // x]-xs)*  
 $\langle$ *proof* $\rangle$

**lemma** *vsubstBinp-preserves-wls[simp]*:  
**assumes** *wlsBinp delta binp*  
**shows** *wlsBinp delta (binp %%[x1 // x]-xs)*  
 $\langle$ *proof* $\rangle$

**lemma** *vsubstEnv-preserves-wls[simp]*:  
**assumes** *wlsEnv rho*  
**shows** *wlsEnv (rho &[x1 // x]-xs)*  
 $\langle$ *proof* $\rangle$

**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**

$$\begin{aligned} ((Op\ delta\ inp\ binp) \#[x1 \wedge x2]-xs) = \\ Op\ delta\ (inp\ \%[x1 \wedge x2]-xs)\ (binp\ \%[x1 \wedge x2]-xs) \end{aligned}$$

*<proof>*

**theorem** *wls-swapAbs-simp*[simp]:

**assumes** *wls s X*

**shows**  $((Abs\ xs\ x\ X)\ \$[y1 \wedge y2]-ys) = Abs\ xs\ (x\ @xs[y1 \wedge y2]-ys)\ (X\ \#[y1 \wedge y2]-ys)$

*<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**

$$\begin{aligned} fresh\ xs\ x\ (Op\ delta\ inp\ binp) = \\ (freshInp\ xs\ x\ inp \wedge freshBinp\ xs\ x\ binp) \end{aligned}$$

*<proof>*

**theorem** *wls-freshAbs-simp*[simp]:

**assumes** *wls s X*

**shows**  $freshAbs\ ys\ y\ (Abs\ xs\ x\ X) = (ys = xs \wedge y = x \vee fresh\ ys\ y\ X)$

*<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*  
 $\langle$ *proof* $\rangle$

**theorem** *wls-skelAbs-simp[simp]*:  
**assumes** *wls s X*  
**shows** *skelAbs (Abs xs x X) = Branch ( $\lambda i$ . Some (skel X)) Map.empty*  
 $\langle$ *proof* $\rangle$

**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**  $((Var\ xs\ x)\ \#[rho]) = Var\ xs\ x$   
 $\langle$ *proof* $\rangle$

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

**theorem** *wls-psubst-Op-simp[simp]*:  
**assumes** *wlsInp delta inp and wlsBinp delta binp and wlsEnv rho*  
**shows**  
 $((Op\ delta\ inp\ binp)\ \#[rho]) = Op\ delta\ (inp\ \%[rho])\ (binp\ \%[rho])$   
 $\langle$ *proof* $\rangle$

**theorem** *wls-psubstAbs-simp[simp]*:  
**assumes** *wls s X and wlsEnv rho and freshEnv xs x rho*  
**shows**  $((Abs\ xs\ x\ X)\ \$[rho]) = Abs\ xs\ x\ (X\ \#[rho])$   
 $\langle$ *proof* $\rangle$

**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**  $((Var\ xs\ x)\ \#[Y / y]-ys) = Var\ xs\ x$   
 $\langle$ *proof* $\rangle$



**theorem** *wls-subst-Var-simp2*[simp]:  
**assumes** *wls* (*asSort* *xs*) *Y*  
**shows**  $((\text{Var } xs \ x) \#[Y / x]\text{-}xs) = Y$   
 $\langle \text{proof} \rangle$

**theorem** *wls-subst-Op-simp*[simp]:  
**assumes** *wls* (*asSort* *ys*) *Y*  
**and** *wlsInp* *delta inp* **and** *wlsBinp* *delta binp*  
**shows**  
 $((\text{Op } delta \ inp \ binp) \#[Y / y]\text{-}ys) =$   
 $\text{Op } delta \ (inp \ \%[Y / y]\text{-}ys) \ (binp \ \%[Y / y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**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**  $((\text{Abs } xs \ x \ X) \$[Y / y]\text{-}ys) = \text{Abs } xs \ x \ (X \#[Y / y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**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**  
 $((\text{Op } delta \ inp \ binp) \#[y1 // y]\text{-}ys) =$   
 $\text{Op } delta \ (inp \ \%[y1 // y]\text{-}ys) \ (binp \ \%[y1 // y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**theorem** *wls-vsubstAbs-simp*[simp]:  
**assumes** *wls* *s X* **and**  
 $xs \neq ys \vee x \notin \{y, y1\}$   
**shows**  $((\text{Abs } xs \ x \ X) \$[y1 // y]\text{-}ys) = \text{Abs } xs \ x \ (X \#[y1 // y]\text{-}ys)$   
 $\langle \text{proof} \rangle$

**lemmas** *wls-vsubstAll-simps* =  
*vsubst-Var-simp wls-vsubst-Op-simp wls-vsubstAbs-simp*

**theorem** *wls-swapped-skel*:  
**assumes** *wls* *s X* **and**  $(X, Y) \in \text{swapped}$   
**shows** *skel* *Y* = *skel* *X*  
 $\langle \text{proof} \rangle$

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

**shows**  $\exists X'. \text{skel } X' = \text{skel } X \wedge (X, X') \in \text{swapped} \wedge \text{wls } s X' \wedge \text{Abs } xs \ x X = \text{Abs } xs \ x' X'$

*<proof>*

**lemmas** *wls-allOps-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*:

$\text{wls } s X \implies |\{x. \neg \text{fresh } xs \ x X\}| <_o |UNIV :: 'var \text{ set}|$

*<proof>*

**theorem** *wls-single-non-freshAbs-ordLess-var*:

$\text{wlsAbs } (us, s) A \implies |\{x. \neg \text{freshAbs } xs \ x A\}| <_o |UNIV :: 'var \text{ set}|$

*<proof>*

**theorem** *wls-obtain-fresh*:

**fixes**  $V :: 'varSort \Rightarrow 'var \text{ set}$  **and**

$XS :: 'sort \Rightarrow ('index, 'bindex, 'varSort, 'var, 'opSym) \text{ term set}$  **and**

$AS :: 'varSort \Rightarrow 'sort \Rightarrow ('index, 'bindex, 'varSort, 'var, 'opSym) \text{ abs set}$  **and**

$Rho :: ('index, 'bindex, 'varSort, 'var, 'opSym) \text{ env set}$  **and**  $zs$

**assumes**  $VVar: \forall xs. |V xs| <_o |UNIV :: 'var \text{ set}| \vee \text{finite } (V xs)$

**and**  $XSVar: \forall s. |XS s| <_o |UNIV :: 'var \text{ set}| \vee \text{finite } (XS s)$

**and**  $ASVar: \forall xs \ s. |AS xs \ s| <_o |UNIV :: 'var \text{ set}| \vee \text{finite } (AS xs \ s)$

**and**  $XSwls: \forall s. \forall X \in XS \ s. \text{wls } s X$

**and**  $ASwls: \forall xs \ s. \forall A \in AS \ xs \ s. \text{wlsAbs } (xs, s) A$

**and**  $RhoVar: |Rho| <_o |UNIV :: 'var \text{ set}| \vee \text{finite } Rho$

**and**  $Rhowls: \forall rho \in Rho. \text{wlsEnv } rho$

**shows**

$\exists z. (\forall xs. z \notin V xs) \wedge$   
 $(\forall s. \forall X \in XS \ s. \text{fresh } zs \ z X) \wedge$   
 $(\forall xs \ s. \forall A \in AS \ xs \ s. \text{freshAbs } zs \ z A) \wedge$   
 $(\forall rho \in Rho. \text{freshEnv } zs \ z rho)$

*<proof>*

**theorem** *wls-obtain-fresh-paramS*:

**assumes**  $\text{wlsPar } P$

**shows**

$\exists z. (\forall xs. z \notin \text{varsOfS } P \ xs) \wedge$   
 $(\forall s. \forall X \in \text{termsOfS } P \ s. \text{fresh } zs \ z X) \wedge$   
 $(\forall us \ s. \forall A \in \text{absOfS } P \ (us, s). \text{freshAbs } zs \ z A) \wedge$

$(\forall \rho \in \text{envsOfS } P. \text{freshEnv } zs \ z \ \rho)$   
 $\langle \text{proof} \rangle$

**lemma** *wlsAbs-freshAbs-nchotomy*:

**assumes**  $A: \text{wlsAbs } (xs,s) \ A$  **and**  $\text{fresh}: \text{freshAbs } xs \ x \ A$   
**shows**  $\exists X. \text{wls } s \ X \wedge A = \text{Abs } xs \ x \ X$   
 $\langle \text{proof} \rangle$

**theorem** *wlsAbs-fresh-nchotomy*:

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

$\langle \text{proof} \rangle$

**theorem** *wlsAbs-fresh-cases*:

**assumes**  $\text{wlsAbs } (xs,s) \ A$  **and**  $\text{wlsPar } P$   
**and**  $\bigwedge x \ X.$   
 $\llbracket \text{wls } s \ X;$   
 $\bigwedge ys. x \notin \text{varsOfS } P \ ys;$   
 $\bigwedge s' \ Y. Y \in \text{termsOfS } P \ s' \implies \text{fresh } xs \ x \ Y;$   
 $\bigwedge us \ s' \ B. B \in \text{absOfS } P \ (us,s') \implies \text{freshAbs } xs \ x \ B;$   
 $\bigwedge \rho. \rho \in \text{envsOfS } P \implies \text{freshEnv } xs \ x \ \rho \rrbracket$   
 $\implies \text{phi } (xs,s) \ (\text{Abs } xs \ x \ X) \ P$

**shows**  $\text{phi } (xs,s) \ A \ P$

$\langle \text{proof} \rangle$

## 7.8 Compositionality properties of freshness and swapping

### 7.8.1 W.r.t. terms

**theorem** *wls-swap-ident[simp]*:

**assumes**  $\text{wls } s \ X$   
**shows**  $(X \ \#[x \wedge x] \text{-}xs) = X$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swap-compose*:

**assumes**  $\text{wls } s \ X$   
**shows**  $((X \ \#[x \wedge y] \text{-}zs) \ \#[x' \wedge y'] \text{-}zs') =$   
 $((X \ \#[x' \wedge y'] \text{-}zs') \ \#[(x \ @zs[x' \wedge y'] \text{-}zs') \wedge (y \ @zs[x' \wedge y'] \text{-}zs')] \text{-}zs)$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swap-commute*:

$\llbracket \text{wls } s \ X; zs \neq zs' \vee \{x,y\} \cap \{x',y'\} = \{\} \rrbracket \implies$   
 $((X \ \#[x \wedge y] \text{-}zs) \ \#[x' \wedge y'] \text{-}zs') = ((X \ \#[x' \wedge y'] \text{-}zs') \ \#[x \wedge y] \text{-}zs)$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swap-involutive[simp]*:  
**assumes** *wls s X*  
**shows**  $((X \#[x \wedge y]-zs) \#[x \wedge y]-zs) = X$   
 $\langle proof \rangle$

**theorem** *wls-swap-inj[simp]*:  
**assumes** *wls s X and wls s X'*  
**shows**  
 $((X \#[x \wedge y]-zs) = (X' \#[x \wedge y]-zs)) =$   
 $(X = X')$   
 $\langle proof \rangle$

**theorem** *wls-swap-involutive2[simp]*:  
**assumes** *wls s X*  
**shows**  $((X \#[x \wedge y]-zs) \#[y \wedge x]-zs) = X$   
 $\langle proof \rangle$

**theorem** *wls-swap-preserves-fresh[simp]*:  
**assumes** *wls s X*  
**shows** *fresh xs (x @xs[y1  $\wedge$  y2]-ys) (X #[y1  $\wedge$  y2]-ys) = fresh xs x X*  
 $\langle proof \rangle$

**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]-ys) = fresh xs x X*  
 $\langle proof \rangle$

**theorem** *wls-fresh-swap-exchange1*:  
**assumes** *wls s X*  
**shows** *fresh xs x2 (X #[x1  $\wedge$  x2]-xs) = fresh xs x1 X*  
 $\langle proof \rangle$

**theorem** *wls-fresh-swap-exchange2*:  
**assumes** *wls s X*  
**shows** *fresh xs x2 (X #[x2  $\wedge$  x1]-xs) = fresh xs x1 X*  
 $\langle proof \rangle$

**theorem** *wls-fresh-swap-id[simp]*:  
**assumes** *wls s X and fresh xs x1 X and fresh xs x2 X*  
**shows**  $(X \#[x1 \wedge x2]-xs) = X$   
 $\langle proof \rangle$

**theorem** *wls-fresh-swap-compose*:  
**assumes** *wls s X and fresh xs y X and fresh xs z X*  
**shows**  $((X \#[y \wedge x]-xs) \#[z \wedge y]-xs) = (X \#[z \wedge x]-xs)$   
 $\langle proof \rangle$

**theorem** *wls-skel-swap*:  
**assumes** *wls s X*  
**shows**  $\text{skel } (X \#[x1 \wedge x2]-xs) = \text{skel } X$   
 $\langle \text{proof} \rangle$

## 7.8.2 W.r.t. environments

**theorem** *wls-swapEnv-ident[simp]*:  
**assumes** *wlsEnv rho*  
**shows**  $(\text{rho} \ \&[x \wedge x]-xs) = \text{rho}$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-compose*:  
**assumes** *wlsEnv rho*  
**shows**  $((\text{rho} \ \&[x \wedge y]-zs) \ \&[x' \wedge y']-zs') =$   
 $((\text{rho} \ \&[x' \wedge y']-zs') \ \&[(x \ @zs[x' \wedge y']-zs') \wedge (y \ @zs[x' \wedge y']-zs')]-zs)$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-commute*:  
 $\llbracket \text{wlsEnv } \text{rho}; \text{zs} \neq \text{zs}' \vee \{x, y\} \cap \{x', y'\} = \{\} \rrbracket \implies$   
 $((\text{rho} \ \&[x \wedge y]-zs) \ \&[x' \wedge y']-zs') = ((\text{rho} \ \&[x' \wedge y']-zs') \ \&[x \wedge y]-zs)$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-involutive[simp]*:  
**assumes** *wlsEnv rho*  
**shows**  $((\text{rho} \ \&[x \wedge y]-zs) \ \&[x \wedge y]-zs) = \text{rho}$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-inj[simp]*:  
**assumes** *wlsEnv rho* **and** *wlsEnv rho'*  
**shows**  
 $((\text{rho} \ \&[x \wedge y]-zs) = (\text{rho}' \ \&[x \wedge y]-zs)) =$   
 $(\text{rho} = \text{rho}')$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-involutive2[simp]*:  
**assumes** *wlsEnv rho*  
**shows**  $((\text{rho} \ \&[x \wedge y]-zs) \ \&[y \wedge x]-zs) = \text{rho}$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-preserves-freshEnv[simp]*:  
**assumes** *wlsEnv rho*  
**shows**  $\text{freshEnv } xs \ (x \ @xs[y1 \wedge y2]-ys) \ (\text{rho} \ \&[y1 \wedge y2]-ys) = \text{freshEnv } xs \ x \ \text{rho}$   
 $\langle \text{proof} \rangle$

**theorem** *wls-swapEnv-preserves-freshEnv-distinct*:

**assumes**  $wlsEnv\ rho$   
 $xs \neq ys \vee x \notin \{y1, y2\}$   
**shows**  $freshEnv\ xs\ x\ (rho\ \&[y1\ \wedge\ y2]-ys) = freshEnv\ xs\ x\ rho$   
 $\langle proof \rangle$

**theorem**  $wls-freshEnv-swapEnv-exchange1$ :  
**assumes**  $wlsEnv\ rho$   
**shows**  $freshEnv\ xs\ x2\ (rho\ \&[x1\ \wedge\ x2]-xs) = freshEnv\ xs\ x1\ rho$   
 $\langle proof \rangle$

**theorem**  $wls-freshEnv-swapEnv-exchange2$ :  
**assumes**  $wlsEnv\ rho$   
**shows**  $freshEnv\ xs\ x2\ (rho\ \&[x2\ \wedge\ x1]-xs) = freshEnv\ xs\ x1\ rho$   
 $\langle proof \rangle$

**theorem**  $wls-freshEnv-swapEnv-id[simp]$ :  
**assumes**  $wlsEnv\ rho$  **and**  $freshEnv\ xs\ x1\ rho$  **and**  $freshEnv\ xs\ x2\ rho$   
**shows**  $(rho\ \&[x1\ \wedge\ x2]-xs) = rho$   
 $\langle proof \rangle$

**theorem**  $wls-freshEnv-swapEnv-compose$ :  
**assumes**  $wlsEnv\ rho$  **and**  $freshEnv\ xs\ y\ rho$  **and**  $freshEnv\ xs\ z\ rho$   
**shows**  $((rho\ \&[y\ \wedge\ x]-xs)\ \&[z\ \wedge\ y]-xs) = (rho\ \&[z\ \wedge\ x]-xs)$   
 $\langle proof \rangle$

### 7.8.3 W.r.t. abstractions

**theorem**  $wls-swapAbs-ident[simp]$ :  
 $wlsAbs\ (us, s)\ A \implies (A\ \$[x\ \wedge\ x]-xs) = A$   
 $\langle proof \rangle$

**theorem**  $wls-swapAbs-compose$ :  
 $wlsAbs\ (us, s)\ A \implies$   
 $((A\ \$[x\ \wedge\ y]-zs)\ \$[x'\ \wedge\ y']-zs') =$   
 $((A\ \$[x'\ \wedge\ y']-zs')\ \$[(x\ @zs[x'\ \wedge\ y']-zs')\ \wedge\ (y\ @zs[x'\ \wedge\ y']-zs')]-zs)$   
 $\langle proof \rangle$

**theorem**  $wls-swapAbs-commute$ :  
**assumes**  $zs \neq zs' \vee \{x, y\} \cap \{x', y'\} = \{\}$   
**shows**  
 $wlsAbs\ (us, s)\ A \implies$   
 $((A\ \$[x\ \wedge\ y]-zs)\ \$[x'\ \wedge\ y']-zs') = ((A\ \$[x'\ \wedge\ y']-zs')\ \$[x\ \wedge\ y]-zs)$   
 $\langle proof \rangle$

**theorem**  $wls-swapAbs-involutive[simp]$ :  
 $wlsAbs\ (us, s)\ A \implies ((A\ \$[x\ \wedge\ y]-zs)\ \$[x\ \wedge\ y]-zs) = A$   
 $\langle proof \rangle$

**theorem**  $wls-swapAbs-sym$ :

$wlsAbs (us,s) A \implies (A \$[x \wedge y]-zs) = (A \$[y \wedge x]-zs)$   
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

**theorem** *wls-skelAbs-swapAbs*:  
*wlsAbs* (us,s) A  
 $\implies$  *skelAbs* (A \ \$[x1 \wedge x2]-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  $\implies$  (X #[idEnv]) = X  
 ⟨proof⟩

**theorem** *wls-psubstEnv-idEnv-id[simp]*:  
*wlsEnv* rho  $\implies$  (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**  $((rho \ [y \leftarrow Y]-ys) \ \&[z1 \wedge z2]-zs) = ((rho \ \&[z1 \wedge z2]-zs) \ [y \leftarrow Y]-ys)$   
 ⟨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)*  
 $\langle$ *proof* $\rangle$

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

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

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

**theorem** *wls-fresh-psubst-I*:  
**assumes** *wls s X* **and** *wlsEnv rho*  
**and** *rho zs z = None  $\implies$  fresh zs z X* **and**  
 $\wedge$  *ys y Y. rho ys y = Some Y  $\implies$  fresh ys y X  $\vee$  fresh zs z Y*  
**shows** *fresh zs z (X #[rho])*  
 $\langle$ *proof* $\rangle$

**theorem** *wls-fresh-subst*:  
**assumes** *wls s X* **and** *wls (asSort ys) Y*  
**shows** *fresh zs z (X #[Y / y]-ys) =*  
 $((zs = ys \wedge z = y) \vee \text{fresh } zs \ z \ X) \wedge (\text{fresh } ys \ y \ X \vee \text{fresh } zs \ z \ Y)$   
 $\langle$ *proof* $\rangle$

**theorem** *wls-fresh-usubst*:  
**assumes** *wls s X*  
**shows** *fresh zs z (X #[y1 // y]-ys) =*  
 $((zs = ys \wedge z = y) \vee \text{fresh } zs \ z \ X) \wedge (\text{fresh } ys \ y \ X \vee (zs \neq ys \vee z \neq y1))$   
 $\langle$ *proof* $\rangle$

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

**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*  $\vee$  *fresh zs z Y*  
*<proof>*

**theorem** *wls-fresh-vsubst-E2*:

**assumes**  $wls\ s\ X$   
**and**  $fresh\ zs\ z\ (X\ \#[y1\ /\ / y]-ys)$   
**shows**  $fresh\ ys\ y\ X\ \vee\ zs\ \neq\ ys\ \vee\ z\ \neq\ y1$   
 $\langle proof \rangle$

**theorem**  $wls-psubst-cong[fundef-cong]$ :  
**assumes**  $wls\ s\ X$  **and**  $wlsEnv\ rho$  **and**  $wlsEnv\ rho'$   
**and**  $\bigwedge\ ys\ y.\ fresh\ ys\ y\ X\ \vee\ rho\ ys\ y = rho'\ ys\ y$   
**shows**  $(X\ \#[rho]) = (X\ \#[rho'])$   
 $\langle proof \rangle$

**theorem**  $wls-fresh-psubst-updEnv$ :  
**assumes**  $wls\ (asSort\ ys)\ Y$  **and**  $wls\ s\ X$  **and**  $wlsEnv\ rho$   
**and**  $fresh\ ys\ y\ X$   
**shows**  $(X\ \#[rho\ [y\ \leftarrow\ Y]-ys]) = (X\ \#[rho])$   
 $\langle proof \rangle$

**theorem**  $wls-freshEnv-psubst-ident[simp]$ :  
**assumes**  $wls\ s\ X$  **and**  $wlsEnv\ rho$   
**and**  $\bigwedge\ zs\ z.\ freshEnv\ zs\ z\ rho\ \vee\ fresh\ zs\ z\ X$   
**shows**  $(X\ \#[rho]) = X$   
 $\langle proof \rangle$

**theorem**  $wls-fresh-subst-ident[simp]$ :  
**assumes**  $wls\ (asSort\ ys)\ Y$  **and**  $wls\ s\ X$  **and**  $fresh\ ys\ y\ X$   
**shows**  $(X\ \#[Y\ / y]-ys) = X$   
 $\langle proof \rangle$

**theorem**  $wls-substEnv-updEnv-fresh$ :  
**assumes**  $wls\ (asSort\ xs)\ X$  **and**  $wls\ (asSort\ ys)\ Y$  **and**  $fresh\ ys\ y\ X$   
**shows**  $((rho\ [x\ \leftarrow\ X]-xs)\ \&[Y\ / y]-ys) = ((rho\ \&[Y\ / y]-ys)\ [x\ \leftarrow\ X]-xs)$   
 $\langle proof \rangle$

**theorem**  $wls-fresh-substEnv-updEnv[simp]$ :  
**assumes**  $wlsEnv\ rho$  **and**  $wls\ (asSort\ ys)\ Y$   
**and**  $freshEnv\ ys\ y\ rho$   
**shows**  $(rho\ \&[Y\ / y]-ys) = (rho\ [y\ \leftarrow\ Y]-ys)$   
 $\langle proof \rangle$

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

**theorem**  $wls-vsubstEnv-updEnv-fresh$ :  
**assumes**  $wls\ s\ X$  **and**  $fresh\ ys\ y\ X$   
**shows**  $((rho\ [x\ \leftarrow\ X]-xs)\ \&[y1\ /\ / y]-ys) = ((rho\ \&[y1\ /\ / y]-ys)\ [x\ \leftarrow\ X]-xs)$   
 $\langle proof \rangle$

**theorem** *wls-fresh-vssubstEnv-updEnv[simp]*:  
**assumes** *wlsEnv rho*  
**and** *freshEnv ys y rho*  
**shows**  $(rho \ \&[y1 \ // \ y]-ys) = (rho \ [y \leftarrow \ Var \ ys \ y1]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-swap-psubst*:  
**assumes** *wls s X and wlsEnv rho*  
**shows**  $((X \ \#[rho]) \ \#[z1 \ \wedge \ z2]-zs) = ((X \ \#[z1 \ \wedge \ z2]-zs) \ \#[rho \ \&[z1 \ \wedge \ z2]-zs])$   
 $\langle proof \rangle$

**theorem** *wls-swap-subst*:  
**assumes** *wls s X and wls (asSort ys) Y*  
**shows**  $((X \ \#[Y \ / \ y]-ys) \ \#[z1 \ \wedge \ z2]-zs) = ((X \ \#[z1 \ \wedge \ z2]-zs) \ \#[(Y \ \#[z1 \ \wedge \ z2]-zs) \ / \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-swap-vssubst*:  
**assumes** *wls s X*  
**shows**  $((X \ \#[y1 \ // \ y]-ys) \ \#[z1 \ \wedge \ z2]-zs) = ((X \ \#[z1 \ \wedge \ z2]-zs) \ \#[(y1 \ @ys[z1 \ \wedge \ z2]-zs) \ // \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-swapEnv-psubstEnv*:  
**assumes** *wlsEnv rho and wlsEnv rho'*  
**shows**  $((rho \ \&[rho']) \ \&[z1 \ \wedge \ z2]-zs) = ((rho \ \&[z1 \ \wedge \ z2]-zs) \ \&[rho' \ \&[z1 \ \wedge \ z2]-zs])$   
 $\langle proof \rangle$

**theorem** *wls-swapEnv-substEnv*:  
**assumes** *wls (asSort ys) Y and wlsEnv rho*  
**shows**  $((rho \ \&[Y \ / \ y]-ys) \ \&[z1 \ \wedge \ z2]-zs) =$   
 $((rho \ \&[z1 \ \wedge \ z2]-zs) \ \&[(Y \ \#[z1 \ \wedge \ z2]-zs) \ / \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-swapEnv-vssubstEnv*:  
**assumes** *wlsEnv rho*  
**shows**  $((rho \ \&[y1 \ // \ y]-ys) \ \&[z1 \ \wedge \ z2]-zs) =$   
 $((rho \ \&[z1 \ \wedge \ z2]-zs) \ \&[(y1 \ @ys[z1 \ \wedge \ z2]-zs) \ // \ (y \ @ys[z1 \ \wedge \ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-psubst-compose*:  
**assumes** *wls s X and wlsEnv rho and wlsEnv rho'*  
**shows**  $((X \ \#[rho]) \ \#[rho']) = (X \ \#[(rho \ \&[rho'])])$   
 $\langle proof \rangle$

**theorem** *wls-psubstEnv-compose*:  
**assumes** *wlsEnv rho and wlsEnv rho' and wlsEnv rho''*  
**shows**  $((rho \ \&[rho']) \ \&[rho'']) = (rho \ \&[(rho' \ \&[rho''])])$   
 $\langle proof \rangle$

**theorem** *wls-psubst-subst-compose:*

**assumes** *wls s X and wls (asSort ys) Y and wlsEnv rho*

**shows**  $((X \#[Y / y]\text{-ys}) \#[rho]) = (X \#[(rho [y \leftarrow (Y \#[rho])]\text{-ys})])$

*<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**  $((X \#[Y / y]\text{-ys}) \#[rho]) = ((X \#[rho]) \#[(Y \#[rho]) / y]\text{-ys})$

*<proof>*

**theorem** *wls-psubstEnv-substEnv-compose-freshEnv:*

**assumes** *wlsEnv rho and wlsEnv rho' and wls (asSort ys) Y*

**assumes** *freshEnv ys y rho'*

**shows**  $((rho \&[Y / y]\text{-ys}) \&[rho']) = ((rho \&[rho']) \&[(Y \#[rho']) / y]\text{-ys})$

*<proof>*

**theorem** *wls-psubstEnv-substEnv-compose:*

**assumes** *wlsEnv rho and wls (asSort ys) Y' and wlsEnv rho'*

**shows**  $((rho \&[Y / y]\text{-ys}) \&[rho']) = (rho \&[(rho' [y \leftarrow (Y \#[rho'])]\text{-ys})])$

*<proof>*

**theorem** *wls-psubst-vsubst-compose:*

**assumes** *wls s X and wlsEnv rho*

**shows**  $((X \#[y1 // y]\text{-ys}) \#[rho]) = (X \#[(rho [y \leftarrow ((Var ys y1) \#[rho])]\text{-ys})])$

*<proof>*

**theorem** *wls-psubstEnv-vsubstEnv-compose:*

**assumes** *wlsEnv rho and wlsEnv rho'*

**shows**  $((rho \&[y1 // y]\text{-ys}) \&[rho']) = (rho \&[(rho' [y \leftarrow ((Var ys y1) \#[rho'])]\text{-ys})])$

*<proof>*

**theorem** *wls-subst-psubst-compose:*

**assumes** *wls s X and wls (asSort ys) Y and wlsEnv rho*

**shows**  $((X \#[rho]) \#[Y / y]\text{-ys}) = (X \#[(rho \&[Y / y]\text{-ys})])$

*<proof>*

**theorem** *wls-substEnv-psubstEnv-compose:*

**assumes** *wlsEnv rho and wls (asSort ys) Y' and wlsEnv rho'*

**shows**  $((rho \&[rho']) \&[Y / y]\text{-ys}) = (rho \&[(rho' \&[Y / y]\text{-ys})])$

*<proof>*

**theorem** *wls-vsubst-psubst-compose:*

**assumes** *wls s X and wlsEnv rho*

**shows**  $((X \#[rho]) \#[y1 // y]\text{-ys}) = (X \#[(rho \&[y1 // y]\text{-ys})])$

*<proof>*

**theorem** *wls-vsubstEnv-psubstEnv-compose:*

**assumes**  $wlsEnv\ rho$  **and**  $wlsEnv\ rho'$   
**shows**  $((rho \ \&[rho']) \ \&[y1 \ // \ y]-ys) = (rho \ \&[(rho' \ \&[y1 \ // \ y]-ys)])$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}subst\text{-}compose1$ :  
**assumes**  $wls\ s\ X$  **and**  $wls\ (asSort\ ys)\ Y1$  **and**  $wls\ (asSort\ ys)\ Y2$   
**shows**  $((X \ \#[Y1 \ / \ y]-ys) \ \#[Y2 \ / \ y]-ys) = (X \ \#[(Y1 \ \#[Y2 \ / \ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}substEnv\text{-}compose1$ :  
**assumes**  $wlsEnv\ rho$  **and**  $wls\ (asSort\ ys)\ Y1$  **and**  $wls\ (asSort\ ys)\ Y2$   
**shows**  $((rho \ \&[Y1 \ / \ y]-ys) \ \&[Y2 \ / \ y]-ys) = (rho \ \&[(Y1 \ \#[Y2 \ / \ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}subst\text{-}vsubst\text{-}compose1$ :  
**assumes**  $wls\ s\ X$  **and**  $wls\ (asSort\ ys)\ Y$  **and**  $y \neq y1$   
**shows**  $((X \ \#[y1 \ // \ y]-ys) \ \#[Y \ / \ y]-ys) = (X \ \#[[y1 \ // \ y]-ys])$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}substEnv\text{-}vsubstEnv\text{-}compose1$ :  
**assumes**  $wlsEnv\ rho$  **and**  $wls\ (asSort\ ys)\ Y$  **and**  $y \neq y1$   
**shows**  $((rho \ \&[y1 \ // \ y]-ys) \ \&[Y \ / \ y]-ys) = (rho \ \&[y1 \ // \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}vsubst\text{-}subst\text{-}compose1$ :  
**assumes**  $wls\ s\ X$  **and**  $wls\ (asSort\ ys)\ Y$   
**shows**  $((X \ \#[Y \ / \ y]-ys) \ \#[y1 \ // \ y]-ys) = (X \ \#[(Y \ \#[y1 \ // \ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}vsubstEnv\text{-}substEnv\text{-}compose1$ :  
**assumes**  $wlsEnv\ rho$  **and**  $wls\ (asSort\ ys)\ Y$   
**shows**  $((rho \ \&[Y \ / \ y]-ys) \ \&[y1 \ // \ y]-ys) = (rho \ \&[(Y \ \#[y1 \ // \ y]-ys) \ / \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}vsubst\text{-}compose1$ :  
**assumes**  $wls\ s\ X$   
**shows**  $((X \ \#[y1 \ // \ y]-ys) \ \#[y2 \ // \ y]-ys) = (X \ \#[(y1 \ @ys[y2 \ / \ y]-ys) \ // \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}vsubstEnv\text{-}compose1$ :  
**assumes**  $wlsEnv\ rho$   
**shows**  $((rho \ \&[y1 \ // \ y]-ys) \ \&[y2 \ // \ y]-ys) = (rho \ \&[(y1 \ @ys[y2 \ / \ y]-ys) \ // \ y]-ys)$   
 $\langle proof \rangle$

**theorem**  $wls\text{-}subst\text{-}compose2$ :  
**assumes**  $wls\ s\ X$  **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**  $((X \ \#[Y \ / \ y]-ys) \ \#[Z \ / \ z]-zs) = ((X \ \#[Z \ / \ z]-zs) \ \#[(Y \ \#[Z \ / \ z]-zs) \ / \ y]-ys)$

*<proof>*

**theorem** *wls-substEnv-compose2:*

**assumes** *wlsEnv rho and wls (asSort ys) Y and wls (asSort zs) Z*

**and** *ys ≠ 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>*

**theorem** *wls-subst-vsubst-compose2:*

**assumes** *wls s X and wls (asSort zs) Z*

**and** *ys ≠ zs ∨ y ≠ z and fresh: fresh ys y Z*

**shows**  $((X \ \#[y1 \ /\ y]-ys) \ \#[Z \ / \ z]-zs) = ((X \ \#[Z \ / \ z]-zs) \ \#[((Var \ ys \ y1) \ \#[Z \ / \ z]-zs) \ / \ y]-ys)$

*<proof>*

**theorem** *wls-substEnv-vsubstEnv-compose2:*

**assumes** *wlsEnv rho and wls (asSort zs) Z*

**and** *ys ≠ zs ∨ y ≠ z and fresh: fresh ys y Z*

**shows**  $((rho \ \&[y1 \ /\ y]-ys) \ \&[Z \ / \ z]-zs) = ((rho \ \&[Z \ / \ z]-zs) \ \&[((Var \ ys \ y1) \ \#[Z \ / \ z]-zs) \ / \ y]-ys)$

*<proof>*

**theorem** *wls-vsubst-subst-compose2:*

**assumes** *wls s X and wls (asSort ys) Y*

**and** *ys ≠ zs ∨ y ∉ {z, z1}*

**shows**  $((X \ \#[Y \ / \ y]-ys) \ \#[z1 \ /\ z]-zs) = ((X \ \#[z1 \ /\ z]-zs) \ \#[(Y \ \#[z1 \ /\ z]-zs) \ / \ y]-ys)$

*<proof>*

**theorem** *wls-vsubstEnv-substEnv-compose2:*

**assumes** *wlsEnv rho and wls (asSort ys) Y*

**and** *ys ≠ zs ∨ y ∉ {z, z1}*

**shows**  $((rho \ \&[Y \ / \ y]-ys) \ \&[z1 \ /\ z]-zs) = ((rho \ \&[z1 \ /\ z]-zs) \ \&[(Y \ \#[z1 \ /\ z]-zs) \ / \ y]-ys)$

*<proof>*

**theorem** *wls-vsubst-compose2:*

**assumes** *wls s X*

**and** *ys ≠ zs ∨ y ∉ {z, z1}*

**shows**  $((X \ \#[y1 \ /\ y]-ys) \ \#[z1 \ /\ z]-zs) = ((X \ \#[z1 \ /\ z]-zs) \ \#[(y1 \ @ys[z1 \ / \ z]-zs) \ /\ y]-ys)$

*<proof>*

**theorem** *wls-vsubstEnv-compose2:*

**assumes** *wlsEnv rho*

**and** *ys ≠ zs ∨ y ∉ {z, z1}*

**shows**  $((rho \ \&[y1 \ /\ y]-ys) \ \&[z1 \ /\ z]-zs) = ((rho \ \&[z1 \ /\ z]-zs) \ \&[(y1 \ @ys[z1 \ / \ z]-zs) \ /\ y]-ys)$

*<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**  $y1 = y2 \vee \text{fresh } ys\ y1\ X$

**shows**  $(X \#[y1 // y2]-ys) = (X \#[y1 \wedge y2]-ys)$

*<proof>*

**theorem** *wls-skel-vsubst*:

**assumes** *wls s X*

**shows**  $\text{skel } (X \#[y1 // y2]-ys) = \text{skel } X$

*<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**  $xs \neq xs' \vee \{x, y\} \cap \{x', y'\} = \{\}$  **and** *fresh xs x X* **and** *fresh xs' x' X*

**shows**  $((X \#[x // y]-xs) \#[x' // y']-xs') = ((X \#[x' // y']-xs') \#[x // y]-xs)$

*<proof>*

**theorem** *wls-induct[case-names Var Op Abs]*:

**assumes**

*Var*:  $\bigwedge xs\ x.\ \text{phi } (\text{asSort } xs) (Var\ xs\ x)$  **and**

*Op*:

$\bigwedge \text{delta } \text{inp } \text{binp}.$

$\llbracket \text{wlsInp } \text{delta } \text{inp}; \text{wlsBinp } \text{delta } \text{binp};$

$\text{liftAll2 } \text{phi } (\text{arOf } \text{delta}) \text{ inp}; \text{liftAll2 } \text{phiAbs } (\text{barOf } \text{delta}) \text{ binp} \rrbracket$



$\implies \text{phi } (\text{stOf } \text{delta}) (\text{Op } \text{delta } \text{inp } \text{binp})$  **and**  
**Abs:**  
 $\bigwedge s \text{ xs } x \text{ X}.$   
 $\llbracket \text{isInBar } (\text{xs},s); \text{wls } s \text{ X};$   
 $\bigwedge Y. (X, Y) \in \text{swapped} \implies \text{phi } s \text{ Y};$   
 $\bigwedge \text{ys } y1 \text{ y2}. \text{phi } s (X \#[y1 // y2]-\text{ys});$   
 $\bigwedge Y. \llbracket \text{wls } s \text{ Y}; \text{skel } Y = \text{skel } X \rrbracket \implies \text{phi } s \text{ Y} \rrbracket$   
 $\implies \text{phiAbs } (\text{xs},s) (\text{Abs } \text{xs } x \text{ X})$

**shows**

$(\text{wls } s \text{ X} \longrightarrow \text{phi } s \text{ X}) \wedge$   
 $(\text{wlsAbs } (\text{xs},s') \text{ A} \longrightarrow \text{phiAbs } (\text{xs},s') \text{ A})$   
 $\langle \text{proof} \rangle$

**theorem** *wls-Abs-vsubst-all-aux:*

**assumes** *wls s X and wls s X'*

**shows**

$(\text{Abs } \text{xs } x \text{ X} = \text{Abs } \text{xs } x' \text{ X}') =$   
 $(\forall y. (y = x \vee \text{fresh } \text{xs } y \text{ X}) \wedge (y = x' \vee \text{fresh } \text{xs } y \text{ X}') \longrightarrow$   
 $(X \#[y // x]-\text{xs}) = (X' \#[y // x']-\text{xs}))$   
 $\langle \text{proof} \rangle$

**theorem** *wls-Abs-vsubst-ex:*

**assumes** *wls s X and wls s X'*

**shows**

$(\text{Abs } \text{xs } x \text{ X} = \text{Abs } \text{xs } x' \text{ X}') =$   
 $(\exists y. y \notin \{x, x'\} \wedge \text{fresh } \text{xs } y \text{ X} \wedge \text{fresh } \text{xs } y \text{ X}' \wedge$   
 $(X \#[y // x]-\text{xs}) = (X' \#[y // x']-\text{xs}))$   
 $\langle \text{proof} \rangle$

**theorem** *wls-Abs-vsubst-all:*

**assumes** *wls s X and wls s X'*

**shows**

$(\text{Abs } \text{xs } x \text{ X} = \text{Abs } \text{xs } x' \text{ X}') =$   
 $(\forall y. (X \#[y // x]-\text{xs}) = (X' \#[y // x']-\text{xs}))$   
 $\langle \text{proof} \rangle$

**theorem** *wls-Abs-subst-all:*

**assumes** *wls s X and wls s X'*

**shows**

$(\text{Abs } \text{xs } x \text{ X} = \text{Abs } \text{xs } x' \text{ X}') =$   
 $(\forall Y. \text{wls } (\text{asSort } \text{xs}) \text{ Y} \longrightarrow (X \#[Y / x]-\text{xs}) = (X' \#[Y / x']-\text{xs}))$   
 $\langle \text{proof} \rangle$

**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'*

$\langle \text{proof} \rangle$

**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**  $Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$   
 $\langle proof \rangle$

**theorem** *wls-Abs-vsubst-fresh[simp]*:  
**assumes**  $wls\ s\ X$  **and**  $fresh\ xs\ x'\ X$   
**shows**  $Abs\ xs\ x'\ (X\ \#[x'\ /\ x]-xs) = Abs\ xs\ x\ X$   
 $\langle proof \rangle$

**theorem** *wls-Abs-subst-Var-fresh[simp]*:  
**assumes**  $wls\ s\ X$  **and**  $fresh\ xs\ x'\ X$   
**shows**  $Abs\ xs\ x'\ (subst\ xs\ (Var\ xs\ x')\ x\ X) = Abs\ xs\ x\ X$   
 $\langle proof \rangle$

**theorem** *wls-Abs-vsubst-congSTR*:  
**assumes**  $wls\ s\ X$  **and**  $wls\ s\ X'$   
**and**  $y = x \vee fresh\ xs\ y\ X$   $y = x' \vee fresh\ xs\ y\ X'$   
**and**  $(X\ \#[y\ /\ x]-xs) = (X'\ \#[y\ /\ x']-xs)$   
**shows**  $Abs\ xs\ x\ X = Abs\ xs\ x'\ X'$   
 $\langle proof \rangle$

#### 7.9.4 Abstraction versions of the properties

**theorem** *wls-psubstAbs-idEnv[simp]*:  
 $wlsAbs\ (us,s)\ A \implies (A\ \$[idEnv]) = A$   
 $\langle proof \rangle$

**theorem** *wls-freshAbs-psubstAbs*:  
**assumes**  $wlsAbs\ (us,s)\ A$  **and**  $wlsEnv\ rho$   
**shows**  
 $freshAbs\ zs\ z\ (A\ \$[rho]) =$   
 $(\forall\ ys\ y.\ freshAbs\ ys\ y\ A \vee freshImEnvAt\ zs\ z\ rho\ ys\ y)$   
 $\langle proof \rangle$

**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**  $freshAbs\ ys\ y\ A \vee (ys \neq zs \vee y \neq z)$   
 $\langle proof \rangle$

**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**  $freshAbs\ ys\ y\ A \vee fresh\ zs\ z\ Y$

*<proof>*

**theorem** *wls-freshAbs-psubstAbs-II*:  
**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*  $\implies$  *freshAbs* *zs z A* **and**  
 $\wedge$  *ys y Y*. *rho* *ys y = Some Y*  $\implies$  *freshAbs* *ys y A*  $\vee$  *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)* =  
 $((zs = ys \wedge z = y) \vee \text{freshAbs } zs \ z \ A) \wedge (\text{freshAbs } ys \ y \ A \vee \text{fresh } zs \ z \ Y)$   
*<proof>*

**theorem** *wls-freshAbs-gsubstAbs*:  
**assumes** *wlsAbs* (*us,s*) *A*  
**shows** *freshAbs* *zs z (A \$[y1 // y]-ys)* =  
 $((zs = ys \wedge z = y) \vee \text{freshAbs } zs \ z \ A) \wedge$   
 $(\text{freshAbs } ys \ y \ A \vee (zs \neq ys \vee z \neq y1))$   
*<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-gsubstAbs-preserves-freshAbs*:  
**assumes** *wlsAbs* (*us,s*) *A*  
**and** *freshAbs* *zs z A* **and**  $zs \neq ys \vee z \neq y1$   
**shows** *freshAbs* *zs z (A \$[y1 // y]-ys)*  
*<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-gsubstAbs[simp]*:  
**assumes** *wlsAbs* (*us,s*) *A*  
**and**  $y \neq y1$

**shows**  $\text{freshAbs } ys \ y \ (A \ \$[y1 \ // \ y]-ys)$   
*<proof>*

**theorem**  $\text{wls-freshAbs-substAbs-E1}$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{wls } (asSort \ ys) \ Y$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[Y \ / \ y]-ys)$  **and**  $z \neq y \ \vee \ zs \neq ys$   
**shows**  $\text{freshAbs } zs \ z \ A$   
*<proof>*

**theorem**  $\text{wls-freshAbs-vsubstAbs-E1}$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[y1 \ // \ y]-ys)$  **and**  $z \neq y \ \vee \ zs \neq ys$   
**shows**  $\text{freshAbs } zs \ z \ A$   
*<proof>*

**theorem**  $\text{wls-freshAbs-substAbs-E2}$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{wls } (asSort \ ys) \ Y$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[Y \ / \ y]-ys)$   
**shows**  $\text{freshAbs } ys \ y \ A \ \vee \ \text{fresh } zs \ z \ Y$   
*<proof>*

**theorem**  $\text{wls-freshAbs-vsubstAbs-E2}$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$   
**and**  $\text{freshAbs } zs \ z \ (A \ \$[y1 \ // \ y]-ys)$   
**shows**  $\text{freshAbs } ys \ y \ A \ \vee \ zs \neq ys \ \vee \ z \neq y1$   
*<proof>*

**theorem**  $\text{wls-psubstAbs-cong}[fundef-cong]$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{wlsEnv } rho$  **and**  $\text{wlsEnv } rho'$   
**and**  $\bigwedge \ ys \ y. \ \text{freshAbs } ys \ y \ A \ \vee \ rho \ ys \ y = rho' \ ys \ y$   
**shows**  $(A \ \$[rho]) = (A \ \$[rho'])$   
*<proof>*

**theorem**  $\text{wls-freshAbs-psubstAbs-updEnv}$ :  
**assumes**  $\text{wls } (asSort \ xs) \ X$  **and**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{wlsEnv } rho$   
**and**  $\text{freshAbs } xs \ x \ A$   
**shows**  $(A \ \$[rho \ [x \leftarrow X]-xs]) = (A \ \$[rho])$   
*<proof>*

**lemma**  $\text{wls-freshEnv-psubstAbs-ident}[simp]$ :  
**assumes**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{wlsEnv } rho$   
**and**  $\bigwedge \ zs \ z. \ \text{freshEnv } zs \ z \ rho \ \vee \ \text{freshAbs } zs \ z \ A$   
**shows**  $(A \ \$[rho]) = A$   
*<proof>*

**theorem**  $\text{wls-freshAbs-substAbs-ident}[simp]$ :  
**assumes**  $\text{wls } (asSort \ xs) \ X$  **and**  $\text{wlsAbs } (us,s) \ A$  **and**  $\text{freshAbs } xs \ x \ A$   
**shows**  $(A \ \$[X \ / \ x]-xs) = A$   
*<proof>*

**theorem** *wls-substAbs-Abs[simp]*:  
**assumes** *wls s X* **and** *wls (asSort xs) Y*  
**shows**  $((Abs\ xs\ x\ X)\ \$[Y / x]-xs) = Abs\ xs\ x\ X$   
 $\langle proof \rangle$

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

**theorem** *wls-swapAbs-psubstAbs*:  
**assumes** *wlsAbs (us,s) A* **and** *wlsEnv rho*  
**shows**  $((A\ \$[rho])\ \$[z1\ \wedge\ z2]-zs) = ((A\ \$[z1\ \wedge\ z2]-zs)\ \$[rho\ \&[z1\ \wedge\ z2]-zs])$   
 $\langle proof \rangle$

**theorem** *wls-swapAbs-substAbs*:  
**assumes** *wlsAbs (us,s) A* **and** *wls (asSort ys) Y*  
**shows**  $((A\ \$[Y / y]-ys)\ \$[z1\ \wedge\ z2]-zs) =$   
 $((A\ \$[z1\ \wedge\ z2]-zs)\ \$[(Y\ \#[z1\ \wedge\ z2]-zs) / (y\ @ys[z1\ \wedge\ z2]-zs)]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-swapAbs-vsubstAbs*:  
**assumes** *wlsAbs (us,s) A*  
**shows**  $((A\ \$[y1\ //\ y]-ys)\ \$[z1\ \wedge\ z2]-zs) =$   
 $((A\ \$[z1\ \wedge\ z2]-zs)\ \$[(y1\ @ys[z1\ \wedge\ z2]-zs) // (y\ @ys[z1\ \wedge\ z2]-zs)]-ys)$   
 $\langle proof \rangle$

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

**theorem** *wls-psubstAbs-substAbs-compose*:  
**assumes** *wlsAbs (us,s) A* **and** *wls (asSort ys) Y* **and** *wlsEnv rho*  
**shows**  $((A\ \$[Y / y]-ys)\ \$[rho]) = (A\ \$[(rho\ [y\ \leftarrow\ (Y\ \#[rho])]-ys)])$   
 $\langle proof \rangle$

**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**  $((A\ \$[Y / y]-ys)\ \$[rho]) = ((A\ \$[rho])\ \$[(Y\ \#[rho]) / y]-ys)$   
 $\langle proof \rangle$

**theorem** *wls-psubstAbs-vsubstAbs-compose*:  
**assumes** *wlsAbs (us,s) A* **and** *wlsEnv rho*  
**shows**  $((A\ \$[y1\ //\ y]-ys)\ \$[rho]) = (A\ \$[(rho\ [y\ \leftarrow\ ((Var\ ys\ y1)\ \#[rho])]-ys)])$   
 $\langle proof \rangle$

**theorem** *wls-substAbs-psubstAbs-compose*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wlsEnv rho*  
**shows**  $((A \ \$[\rho]) \ \$[Y / y]\text{-ys}) = (A \ \$[(\rho \ \&[Y / y]\text{-ys})])$   
*<proof>*

**theorem** *wls-vsubstAbs-psubstAbs-compose*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wlsEnv rho*  
**shows**  $((A \ \$[\rho]) \ \$[y1 // y]\text{-ys}) = (A \ \$[(\rho \ \&[y1 // y]\text{-ys})])$   
*<proof>*

**theorem** *wls-substAbs-compose1*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y1* **and** *wls* (*asSort ys*) *Y2*  
**shows**  $((A \ \$[Y1 / y]\text{-ys}) \ \$[Y2 / y]\text{-ys}) = (A \ \$[(Y1 \ \#[Y2 / y]\text{-ys}) / y]\text{-ys})$   
*<proof>*

**theorem** *wls-substAbs-vsubstAbs-compose1*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y* **and**  $y \neq y1$   
**shows**  $((A \ \$[y1 // y]\text{-ys}) \ \$[Y / y]\text{-ys}) = (A \ \$[y1 // y]\text{-ys})$   
*<proof>*

**theorem** *wls-vsubstAbs-substAbs-compose1*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y*  
**shows**  $((A \ \$[Y / y]\text{-ys}) \ \$[y1 // y]\text{-ys}) = (A \ \$[(Y \ \#[y1 // y]\text{-ys}) / y]\text{-ys})$   
*<proof>*

**theorem** *wls-vsubstAbs-compose1*:  
**assumes** *wlsAbs* (*us,s*) *A*  
**shows**  $((A \ \$[y1 // y]\text{-ys}) \ \$[y2 // y]\text{-ys}) = (A \ \$[(y1 \ @ys[y2 / y]\text{-ys}) // y]\text{-ys})$   
*<proof>*

**theorem** *wls-substAbs-compose2*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y* **and** *wls* (*asSort zs*) *Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh*: *fresh ys y Z*  
**shows**  $((A \ \$[Y / y]\text{-ys}) \ \$[Z / z]\text{-zs}) = ((A \ \$[Z / z]\text{-zs}) \ \$[(Y \ \#[Z / z]\text{-zs}) / y]\text{-ys})$   
*<proof>*

**theorem** *wls-substAbs-vsubstAbs-compose2*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort zs*) *Z*  
**and**  $ys \neq zs \vee y \neq z$  **and** *fresh*: *fresh ys y Z*  
**shows**  $((A \ \$[y1 // y]\text{-ys}) \ \$[Z / z]\text{-zs}) = ((A \ \$[Z / z]\text{-zs}) \ \$[((Var \ ys \ y1) \ \#[Z / z]\text{-zs}) / y]\text{-ys})$   
*<proof>*

**theorem** *wls-vsubstAbs-substAbs-compose2*:  
**assumes** *wlsAbs* (*us,s*) *A* **and** *wls* (*asSort ys*) *Y*  
**and**  $ys \neq zs \vee y \notin \{z, z1\}$   
**shows**  $((A \ \$[Y / y]\text{-ys}) \ \$[z1 // z]\text{-zs}) = ((A \ \$[z1 // z]\text{-zs}) \ \$[(Y \ \#[z1 // z]\text{-zs}) / y]\text{-ys})$   
*<proof>*

**theorem** *wls-vsubstAbs-compose2*:

**assumes** *wlsAbs* (*us,s*) *A*

**and**  $ys \neq zs \vee y \notin \{z, z1\}$

**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**  $y1 = y2 \vee freshAbs \ ys \ y1 \ A$

**shows**  $(A \ \$[y1 \ // \ y2]-ys) = (A \ \$[y1 \ \wedge \ y2]-ys)$

*<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**  $xs \neq xs' \vee \{x,y\} \cap \{x',y'\} = \{\}$  **and** *freshAbs* *xs* *x* *A* **and** *freshAbs* *xs'* *x'* *A*

**shows**  $((A \ \$[x \ // \ y]-xs) \ \$[x' \ // \ y']-xs') = ((A \ \$[x' \ // \ y']-xs') \ \$[x \ // \ y]-xs)$

*<proof>*

**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-allOps-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) ==*  
 $\exists xs\ x. s = asSort\ xs \wedge X = Var\ xs\ x$

**definition** *castVar where*

*castVar s (X :: ('index,'bindex,'varSort,'var,'opSym)term) ==*  
 $SOME\ xs.x. s = asSort\ (fst\ xs-x) \wedge X = Var\ (fst\ xs-x)\ (snd\ xs-x)$

**definition** *isOp where*

*isOp s X ≡*  
 $\exists\ delta\ inp\ binp.$   
 $wlsInp\ delta\ inp \wedge wlsBinp\ delta\ binp \wedge s = stOf\ delta \wedge X = Op\ delta\ inp\ binp$

**definition** *castOp where*

*castOp s X ≡*  
 $SOME\ delta-inp-binp.$   
 $wlsInp\ (fst3\ delta-inp-binp)\ (snd3\ delta-inp-binp) \wedge$   
 $wlsBinp\ (fst3\ delta-inp-binp)\ (trd3\ delta-inp-binp) \wedge$   
 $s = stOf\ (fst3\ delta-inp-binp) \wedge$   
 $X = Op\ (fst3\ delta-inp-binp)\ (snd3\ delta-inp-binp)\ (trd3\ delta-inp-binp)$



**definition** *sortTermCase* **where**

*sortTermCase* *fVar* *fOp* *s* *X*  $\equiv$

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*]:

$\neg$  *isVar* *s* (*Op* *delta* *inp* *binp*)

*<proof>*

**lemma** *isVar-imp-wls*:

*isVar* *s* *X*  $\implies$  *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*  $\wedge$

*Var* (*fst* (*castVar* *s* *X*)) (*snd* (*castVar* *s* *X*)) = *X*

*<proof>*

**lemma** *asSort-castVar*[*simp*]:

*isVar* *s* *X*  $\implies$  *asSort* (*fst* (*castVar* *s* *X*)) = *s*

*<proof>*

**lemma** *Var-castVar*[*simp*]:

*isVar* *s* *X*  $\implies$  *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'*  $\wedge$  *X* = *X'*)

*<proof>*

**lemmas** *castVar-simps* =

*castVar-asSort-Var*  
*asSort-castVar Var-castVar castVar-inj*

**lemma** *isOp-stOf-Op[simp]*:  
 $\llbracket \text{wlsInp } \delta \text{ inp}; \text{wlsBinp } \delta \text{ binp} \rrbracket$   
 $\implies \text{isOp } (\text{stOf } \delta) (Op \ \delta \ \text{inp} \ \text{binp})$   
 $\langle \text{proof} \rangle$

**lemma** *not-isOp-Var[simp]*:  
 $\neg \text{isOp } s (Var \ x \ X)$   
 $\langle \text{proof} \rangle$

**lemma** *isOp-imp-wls*:  
 $\text{isOp } s \ X \implies \text{wls } s \ X$   
 $\langle \text{proof} \rangle$

**lemmas** *isOp-simps =*  
*isOp-stOf-Op not-isOp-Var*

**lemma** *castOp-stOf-Op[simp]*:  
**assumes** *wlsInp delta inp and wlsBinp delta binp*  
**shows**  $\text{castOp } (\text{stOf } \delta) (Op \ \delta \ \text{inp} \ \text{binp}) = (\delta, \text{inp}, \text{binp})$   
 $\langle \text{proof} \rangle$

**lemma** *isOp-castOp*:  
**assumes** *isOp s X*  
**shows**  $\text{wlsInp } (\text{fst3 } (\text{castOp } s \ X)) (\text{snd3 } (\text{castOp } s \ X)) \wedge$   
 $\text{wlsBinp } (\text{fst3 } (\text{castOp } s \ X)) (\text{trd3 } (\text{castOp } s \ X)) \wedge$   
 $\text{stOf } (\text{fst3 } (\text{castOp } s \ X)) = s \wedge$   
 $Op (\text{fst3 } (\text{castOp } s \ X)) (\text{snd3 } (\text{castOp } s \ X)) (\text{trd3 } (\text{castOp } s \ X)) = X$   
 $\langle \text{proof} \rangle$

**lemma** *wlsInp-castOp[simp]*:  
 $\text{isOp } s \ X \implies \text{wlsInp } (\text{fst3 } (\text{castOp } s \ X)) (\text{snd3 } (\text{castOp } s \ X))$   
 $\langle \text{proof} \rangle$

**lemma** *wlsBinp-castOp[simp]*:  
 $\text{isOp } s \ X \implies \text{wlsBinp } (\text{fst3 } (\text{castOp } s \ X)) (\text{trd3 } (\text{castOp } s \ X))$   
 $\langle \text{proof} \rangle$

**lemma** *stOf-castOp[simp]*:  
 $\text{isOp } s \ X \implies \text{stOf } (\text{fst3 } (\text{castOp } s \ X)) = s$   
 $\langle \text{proof} \rangle$

**lemma** *Op-castOp[simp]*:  
 $\text{isOp } s \ X \implies$   
 $Op (\text{fst3 } (\text{castOp } s \ X)) (\text{snd3 } (\text{castOp } s \ X)) (\text{trd3 } (\text{castOp } s \ X)) = X$

$\langle proof \rangle$

**lemma** *castOp-inj*[simp]:

**assumes** *isOp* *s* *X* **and** *isOp* *s'* *X'*

**shows**  $(castOp\ s\ X = castOp\ s'\ X') = (s = s' \wedge X = X')$

$\langle proof \rangle$

**lemmas** *castOp-simps* =

*castOp-stOf-Op* *wlsInp-castOp* *wlsBinp-castOp*

*stOf-castOp* *Op-castOp* *castOp-inj*

**lemma** *not-isVar-isOp*:

$\neg (isVar\ s\ X \wedge isOp\ s\ X)$

$\langle proof \rangle$

**lemma** *isVar-or-isOp*:

$wls\ s\ X \implies isVar\ s\ X \vee isOp\ s\ X$

$\langle proof \rangle$

**lemma** *sortTermCase-asSort-Var-simp*[simp]:

$sortTermCase\ fVar\ fOp\ (asSort\ xs)\ (Var\ xs\ x) = fVar\ xs\ x$

$\langle proof \rangle$

**lemma** *sortTermCase-stOf-Op-simp*[simp]:

$\llbracket wlsInp\ delta\ inp; wlsBinp\ delta\ binp \rrbracket \implies$

$sortTermCase\ fVar\ fOp\ (stOf\ delta)\ (Op\ delta\ inp\ binp) = fOp\ delta\ inp\ binp$

$\langle proof \rangle$

**lemma** *sortTermCase-cong*[fundef-cong]:

**assumes**  $\bigwedge xs\ x. fVar\ xs\ x = gVar\ xs\ x$

**and**  $\bigwedge delta\ inp\ binp. \llbracket wlsInp\ delta\ inp; wlsInp\ delta\ inp \rrbracket$

$\implies fOp\ delta\ inp\ binp = gOp\ delta\ inp\ binp$

**shows**  $wls\ s\ X \implies$

$sortTermCase\ fVar\ fOp\ s\ X = sortTermCase\ gVar\ gOp\ s\ X$

$\langle proof \rangle$

**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**

$$\text{castAbs } xs \ s \ A \equiv \text{SOME } x\text{-}X. \text{ wls } s \ (\text{snd } x\text{-}X) \wedge A = \text{Abs } xs \ (\text{fst } x\text{-}X) \ (\text{snd } x\text{-}X)$$

**definition** *absCase* **where**

*absCase*  $xs \ s \ f \ A \equiv$  if *wlsAbs* (xs,s) A then  $f \ (\text{fst } (\text{castAbs } xs \ s \ A)) \ (\text{snd } (\text{castAbs } xs \ s \ A))$  else *undefined*

**definition** *compatAbsSwap* **where**

$$\begin{aligned} \text{compatAbsSwap } xs \ s \ f &\equiv \\ \forall x \ X \ x' \ X'. (\forall y. (y = x \vee \text{fresh } xs \ y \ X) \wedge (y = x' \vee \text{fresh } xs \ y \ X)) & \\ \quad \longrightarrow (X \ \#[y \wedge x]\text{-}xs) = (X' \ \#[y \wedge x']\text{-}xs)) & \\ \quad \longrightarrow f \ x \ X = f \ x' \ X' & \end{aligned}$$

**definition** *compatAbsSubst* **where**

$$\begin{aligned} \text{compatAbsSubst } xs \ s \ f &\equiv \\ \forall x \ X \ x' \ X'. (\forall Y. \text{wls } (\text{asSort } xs) \ Y \longrightarrow (X \ \#[Y / x]\text{-}xs) = (X' \ \#[Y / x']\text{-}xs)) & \\ \quad \longrightarrow f \ x \ X = f \ x' \ X' & \end{aligned}$$

**definition** *compatAbsVsubst* **where**

$$\begin{aligned} \text{compatAbsVsubst } xs \ s \ f &\equiv \\ \forall x \ X \ x' \ X'. (\forall y. (X \ \#[y // x]\text{-}xs) = (X' \ \#[y // x']\text{-}xs)) & \\ \quad \longrightarrow f \ x \ X = f \ x' \ X' & \end{aligned}$$

**lemma** *wlsAbs-castAbs*:

**assumes** *wlsAbs* (xs,s) A

**shows**  $\text{wls } s \ (\text{snd } (\text{castAbs } xs \ s \ A)) \wedge$

$$\text{Abs } xs \ (\text{fst } (\text{castAbs } xs \ s \ A)) \ (\text{snd } (\text{castAbs } xs \ s \ A)) = A$$

*<proof>*

**lemma** *wls-castAbs[simp]*:

$$\text{wlsAbs } (xs,s) \ A \implies \text{wls } s \ (\text{snd } (\text{castAbs } xs \ s \ A))$$

*<proof>*

**lemma** *Abs-castAbs[simp]*:  
 $wlsAbs\ (xs,s)\ A \implies Abs\ xs\ (fst\ (castAbs\ xs\ s\ A))\ (snd\ (castAbs\ xs\ s\ A)) = A$   
*<proof>*

**lemma** *castAbs-Abs-swap*:  
**assumes** *isInBar*  $(xs,s)$  **and**  $X: wls\ s\ X$   
**and**  $yxX: y = x \vee fresh\ xs\ y\ X$  **and**  $yx'X': y = x' \vee fresh\ xs\ y\ X'$   
**and**  $*$ :  $castAbs\ xs\ s\ (Abs\ xs\ x\ X) = (x',X')$   
**shows**  $(X\ \#[y \wedge x]-xs) = (X'\ \#[y \wedge x']-xs)$   
*<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]*:  
 $compatAbsVsubst\ xs\ s\ f \implies compatAbsSubst\ xs\ s\ f$   
*<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**  $\bigwedge x X. wls s X \implies f x X = f' x X$   
**shows** *wlsAbs (xs,s) A  $\implies$*   
*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 iter-

ation principles.

As a matter of notation: the prefix “g” will stand for “generalized” – elements of models are referred to as “generalized terms”. The actual full prefix will be “ig” (where “i” stands for “iteration”), symbolizing the fact that the models from this section support iteration, and not general recursion. The latter is dealt with by the models introduced in the next section, for which we use the simple prefix “g”.

## 8.1 Models

We have two basic kinds of models:

- fresh-swap (FSw) models, featuring operations corresponding to the concrete syntactic constructs (“Var”, “Op”, “Abs”), henceforth referred to simply as *the constructs*, and to fresh and swap;

- fresh-swap-subst (FSb) models, featuring substitution instead of swapping.

We also consider two combinations of the above, FSwSb-models and FSbSw-models.

To keep things structurally simple, we use one single Isabelle for all the 4 kinds models, allowing the most generous signature. Since terms are the main actors of our theory, models being considered only for the sake of recursive definitions, we call the items inhabiting these models “generalized” terms, abstractions and inputs, and correspondingly the operations; hence the prefix “g” from the names of the type parameters and operators. (However, we refer to the generalized items using the same notations as for “concrete items”: X, A, etc.) Indeed, a model can be regarded as implementing a generalization/axiomatization of the term structure, where now the objects are not terms, but do have term-like properties.

### 8.1.1 Raw models

```

record ('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm,'gAbs)model =
  igWls :: 'sort ⇒ 'gTerm ⇒ bool
  igWlsAbs :: 'varSort × 'sort ⇒ 'gAbs ⇒ bool

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

  igFresh :: 'varSort ⇒ 'var ⇒ 'gTerm ⇒ bool
  igFreshAbs :: 'varSort ⇒ 'var ⇒ 'gAbs ⇒ bool

  igSwap :: 'varSort ⇒ 'var ⇒ 'var ⇒ 'gTerm ⇒ 'gTerm
  igSwapAbs :: 'varSort ⇒ 'var ⇒ 'var ⇒ 'gAbs ⇒ 'gAbs

  igSubst :: 'varSort ⇒ 'gTerm ⇒ 'var ⇒ 'gTerm ⇒ 'gTerm

```

$igSubstAbs :: 'varSort \Rightarrow 'gTerm \Rightarrow 'var \Rightarrow 'gAbs \Rightarrow 'gAbs$

- “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**

$igFreshInp \text{ MOD } ys \ y \ inp == liftAll \ (igFresh \ \text{MOD } ys \ y) \ inp$

**definition** *igFreshBinp* **where**

$igFreshBinp \text{ MOD } ys \ y \ binp == liftAll \ (igFreshAbs \ \text{MOD } ys \ y) \ binp$

**definition** *igSwapInp* **where**

$igSwapInp \text{ MOD } zs \ z1 \ z2 \ inp == lift \ (igSwap \ \text{MOD } zs \ z1 \ z2) \ inp$

**definition** *igSwapBinp* **where**

$igSwapBinp \text{ MOD } zs \ z1 \ z2 \ binp == lift \ (igSwapAbs \ \text{MOD } zs \ z1 \ z2) \ binp$

**definition** *igSubstInp* **where**

$igSubstInp \text{ MOD } ys \ Y \ y \ inp == lift \ (igSubst \ \text{MOD } ys \ Y \ y) \ inp$

**definition** *igSubstBinp* **where**

$igSubstBinp \text{ MOD } ys \ Y \ y \ binp == lift \ (igSubstAbs \ \text{MOD } ys \ Y \ y) \ binp$

**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).<sup>2</sup>

- For “iwlsFSb”: the same as for “iwlsFSw”, except that:

---

<sup>2</sup>Here, by “congruence for abstraction” we do not mean the standard notion of congruence (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 arguments be equal, but that they be in a “permutative” relationship based either on swapping or on substitution.



- “swap” is replaced by “subst”; <sup>3</sup>
- The [fresh and swap]-based congruence clause is replaced by an “abstraction-renaming” clause, which is stronger than the corresponding [fresh and subst]-based congruence clause. <sup>4</sup>
- For “iwlsFSwSb”: the clauses for “iwlsFSw”, plus some of the definition-like clauses for “subst”. <sup>5</sup>
- 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**

$$\begin{aligned} &igWlsInp \text{ MOD } \delta \text{ inp} == \\ &wlsOpS \delta \wedge sameDom (arOf \delta) \text{ inp} \wedge liftAll2 (igWls \text{ MOD}) (arOf \delta) \\ &\text{inp} \end{aligned}$$

**lemmas** *igWlsInp-defs* = *igWlsInp-def sameDom-def liftAll2-def*

**definition** *igWlsBinp* **where**

$$\begin{aligned} &igWlsBinp \text{ MOD } \delta \text{ binp} == \\ &wlsOpS \delta \wedge sameDom (barOf \delta) \text{ binp} \wedge liftAll2 (igWlsAbs \text{ MOD}) (barOf \\ &\delta) \text{ binp} \end{aligned}$$

**lemmas** *igWlsBinp-defs* = *igWlsBinp-def sameDom-def liftAll2-def*

Domain disjointness:

**definition** *igWlsDisj* **where**

$$igWlsDisj \text{ MOD} == \forall s s' X. igWls \text{ MOD } s X \wedge igWls \text{ MOD } s' X \longrightarrow s = s'$$

**definition** *igWlsAbsDisj* **where**

$$igWlsAbsDisj \text{ MOD} ==$$


---

<sup>3</sup>Note that traditionally alpha-equivalence is defined using “subst”, not “swap”.

<sup>4</sup>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.

<sup>5</sup>Not all the “subst” definition-like clauses from “iwlsFSb” are required for “iwlsFSwSb” – namely, the clause that we call “igSubstIGAbsCls2” is not required here.

$$\begin{aligned}
& \forall xs\ s\ xs'\ s'\ A. \\
& \quad isInBar\ (xs,s) \wedge isInBar\ (xs',s') \wedge \\
& \quad igWlsAbs\ MOD\ (xs,s)\ A \wedge igWlsAbs\ MOD\ (xs',s')\ A \\
& \quad \longrightarrow xs = xs' \wedge s = s'
\end{aligned}$$

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

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

Abstraction 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  $\implies$  *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 ==

$\forall$   $\text{delta inp binp}$ .  
 $\text{igWls MOD (stOf delta) (igOp MOD delta inp binp)} =$   
 $(\text{igWlsInp MOD delta inp} \wedge \text{igWlsBinp MOD delta binp})$

**lemma**  $\text{igOpIPresIGWlsSTR-imp-igOpIPresIGWls}$ :  
 $\text{igOpIPresIGWlsSTR MOD} \implies \text{igOpIPresIGWls MOD}$   
 ⟨proof⟩

**definition**  $\text{igConsIPresIGWls}$  **where**  
 $\text{igConsIPresIGWls MOD} ==$   
 $\text{igVarIPresIGWls MOD} \wedge$   
 $\text{igAbsIPresIGWls MOD} \wedge$   
 $\text{igOpIPresIGWls MOD}$

**lemmas**  $\text{igConsIPresIGWls-defs} = \text{igConsIPresIGWls-def}$   
 $\text{igVarIPresIGWls-def}$   
 $\text{igAbsIPresIGWls-def}$   
 $\text{igOpIPresIGWls-def}$

**definition**  $\text{igConsIPresIGWlsSTR}$  **where**  
 $\text{igConsIPresIGWlsSTR MOD} ==$   
 $\text{igVarIPresIGWls MOD} \wedge$   
 $\text{igAbsIPresIGWlsSTR MOD} \wedge$   
 $\text{igOpIPresIGWlsSTR MOD}$

**lemmas**  $\text{igConsIPresIGWlsSTR-defs} = \text{igConsIPresIGWlsSTR-def}$   
 $\text{igVarIPresIGWls-def}$   
 $\text{igAbsIPresIGWlsSTR-def}$   
 $\text{igOpIPresIGWlsSTR-def}$

**lemma**  $\text{igConsIPresIGWlsSTR-imp-igConsIPresIGWls}$ :  
 $\text{igConsIPresIGWlsSTR MOD} \implies \text{igConsIPresIGWls MOD}$   
 ⟨proof⟩

“swap” preserves the domains:

**definition**  $\text{igSwapIPresIGWls}$  **where**  
 $\text{igSwapIPresIGWls MOD} ==$   
 $\forall$   $zs z1 z2 s X$ .  $\text{igWls MOD } s X \longrightarrow$   
 $\text{igWls MOD } s (\text{igSwap MOD } zs z1 z2 X)$

**definition**  $\text{igSwapIPresIGWlsSTR}$  **where**  
 $\text{igSwapIPresIGWlsSTR MOD} ==$   
 $\forall$   $zs z1 z2 s X$ .  $\text{igWls MOD } s (\text{igSwap MOD } zs z1 z2 X) =$   
 $\text{igWls MOD } s X$

**lemma**  $\text{igSwapIPresIGWlsSTR-imp-igSwapIPresIGWls}$ :  
 $\text{igSwapIPresIGWlsSTR MOD} \implies \text{igSwapIPresIGWls MOD}$   
 ⟨proof⟩

**definition**  $igSwapAbsIPresIGWlsAbs$  **where**

$igSwapAbsIPresIGWlsAbs \text{ MOD} ==$

$\forall zs \ z1 \ z2 \ us \ s \ A.$

$isInBar \ (us,s) \wedge igWlsAbs \text{ MOD} \ (us,s) \ A \longrightarrow$

$igWlsAbs \text{ MOD} \ (us,s) \ (igSwapAbs \text{ MOD} \ zs \ z1 \ z2 \ A)$

**definition**  $igSwapAbsIPresIGWlsAbsSTR$  **where**

$igSwapAbsIPresIGWlsAbsSTR \text{ MOD} ==$

$\forall zs \ z1 \ z2 \ us \ s \ A.$

$igWlsAbs \text{ MOD} \ (us,s) \ (igSwapAbs \text{ MOD} \ zs \ z1 \ z2 \ A) =$

$igWlsAbs \text{ MOD} \ (us,s) \ A$

**lemma**  $igSwapAbsIPresIGWlsAbsSTR$ - $imp$ - $igSwapAbsIPresIGWlsAbs$ :

$igSwapAbsIPresIGWlsAbsSTR \text{ MOD} \Longrightarrow igSwapAbsIPresIGWlsAbs \text{ MOD}$

$\langle proof \rangle$

**definition**  $igSwapAllIPresIGWlsAll$  **where**

$igSwapAllIPresIGWlsAll \text{ MOD} ==$

$igSwapIPresIGWls \text{ MOD} \wedge igSwapAbsIPresIGWlsAbs \text{ MOD}$

**lemmas**  $igSwapAllIPresIGWlsAll$ - $defs = igSwapAllIPresIGWlsAll$ - $def$

$igSwapIPresIGWls$ - $def \ igSwapAbsIPresIGWlsAbs$ - $def$

**definition**  $igSwapAllIPresIGWlsAllSTR$  **where**

$igSwapAllIPresIGWlsAllSTR \text{ MOD} ==$

$igSwapIPresIGWlsSTR \text{ MOD} \wedge igSwapAbsIPresIGWlsAbsSTR \text{ MOD}$

**lemmas**  $igSwapAllIPresIGWlsAllSTR$ - $defs = igSwapAllIPresIGWlsAllSTR$ - $def$

$igSwapIPresIGWlsSTR$ - $def \ igSwapAbsIPresIGWlsAbsSTR$ - $def$

**lemma**  $igSwapAllIPresIGWlsAllSTR$ - $imp$ - $igSwapAllIPresIGWlsAll$ :

$igSwapAllIPresIGWlsAllSTR \text{ MOD} \Longrightarrow igSwapAllIPresIGWlsAll \text{ MOD}$

$\langle proof \rangle$

“subst” preserves the domains:

**definition**  $igSubstIPresIGWls$  **where**

$igSubstIPresIGWls \text{ MOD} ==$

$\forall ys \ Y \ y \ s \ X. \ igWls \text{ MOD} \ (asSort \ ys) \ Y \wedge \ igWls \text{ MOD} \ s \ X \longrightarrow$

$igWls \text{ MOD} \ s \ (igSubst \text{ MOD} \ ys \ Y \ y \ X)$

**definition**  $igSubstIPresIGWlsSTR$  **where**

$igSubstIPresIGWlsSTR \text{ MOD} ==$

$\forall ys \ Y \ y \ s \ X.$

$igWls \text{ MOD} \ s \ (igSubst \text{ MOD} \ ys \ Y \ y \ X) =$

$(igWls \text{ MOD} \ (asSort \ ys) \ Y \wedge \ igWls \text{ MOD} \ s \ X)$

**lemma**  $igSubstIPresIGWlsSTR$ - $imp$ - $igSubstIPresIGWls$ :

$igSubstIPresIGWlsSTR \text{ MOD} \Longrightarrow igSubstIPresIGWls \text{ MOD}$

$\langle proof \rangle$

**definition** *igSubstAbsIPresIGWlsAbs* **where**

*igSubstAbsIPresIGWlsAbs* MOD ==

$\forall ys Y y us s A.$

$isInBar (us,s) \wedge igWls MOD (asSort ys) Y \wedge igWlsAbs MOD (us,s) A \longrightarrow$   
 $igWlsAbs MOD (us,s) (igSubstAbs MOD ys Y y A)$

**definition** *igSubstAbsIPresIGWlsAbsSTR* **where**

*igSubstAbsIPresIGWlsAbsSTR* MOD ==

$\forall ys Y y us s A.$

$igWlsAbs MOD (us,s) (igSubstAbs MOD ys Y y A) =$   
 $(igWls MOD (asSort ys) Y \wedge igWlsAbs MOD (us,s) A)$

**lemma** *igSubstAbsIPresIGWlsAbsSTR-imp-igSubstAbsIPresIGWlsAbs*:

*igSubstAbsIPresIGWlsAbsSTR* MOD  $\implies$  *igSubstAbsIPresIGWlsAbs* MOD

*<proof>*

**definition** *igSubstAllIPresIGWlsAll* **where**

*igSubstAllIPresIGWlsAll* MOD ==

*igSubstIPresIGWls* MOD  $\wedge$  *igSubstAbsIPresIGWlsAbs* MOD

**lemmas** *igSubstAllIPresIGWlsAll-defs* = *igSubstAllIPresIGWlsAll-def*

*igSubstIPresIGWls-def* *igSubstAbsIPresIGWlsAbs-def*

**definition** *igSubstAllIPresIGWlsAllSTR* **where**

*igSubstAllIPresIGWlsAllSTR* MOD ==

*igSubstIPresIGWlsSTR* MOD  $\wedge$  *igSubstAbsIPresIGWlsAbsSTR* MOD

**lemmas** *igSubstAllIPresIGWlsAllSTR-defs* = *igSubstAllIPresIGWlsAllSTR-def*

*igSubstIPresIGWlsSTR-def* *igSubstAbsIPresIGWlsAbsSTR-def*

**lemma** *igSubstAllIPresIGWlsAllSTR-imp-igSubstAllIPresIGWlsAll*:

*igSubstAllIPresIGWlsAllSTR* MOD  $\implies$  *igSubstAllIPresIGWlsAll* MOD

*<proof>*

Clauses for fresh: fully conditional versions and less conditional, stronger versions (the latter having suffix "STR").

**definition** *igFreshIGVar* **where**

*igFreshIGVar* MOD ==

$\forall ys y xs x.$

$ys \neq xs \vee y \neq x \longrightarrow$   
 $igFresh MOD ys y (igVar MOD xs x)$

**definition** *igFreshIGAbs1* **where**

*igFreshIGAbs1* MOD ==

$\forall ys y s X.$

$isInBar (ys,s) \wedge igWls MOD s X \longrightarrow$   
 $igFreshAbs MOD ys y (igAbs MOD ys y X)$

**definition** *igFreshIGAbs1STR* where

*igFreshIGAbs1STR* MOD ==

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

**lemma** *igFreshIGAbs1STR-imp-igFreshIGAbs1*:

*igFreshIGAbs1STR* MOD  $\implies$  *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  $\implies$  *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  $\implies$  *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  $\implies$  *igFreshCls* MOD  
{proof}

**definition** *igSwapIGVar* **where**

*igSwapIGVar* MOD ==  
 $\forall$  *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 ==  
 $\forall$  *zs z1 z2 xs x s X*.  
*isInBar* (*xs,s*) ∧ *igWls* MOD *s X*  $\longrightarrow$   
*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 ==  
 $\forall$  *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  $\implies$  *igSwapIGAbs* MOD  
{proof}

**definition** *igSwapIGOp* **where**

*igSwapIGOp* MOD ==  
 $\forall$  *zs z1 z2 delta inp binp*.  
*igWlsInp* MOD *delta inp* ∧ *igWlsBinp* MOD *delta binp*  $\longrightarrow$   
*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*  $\implies$  *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*  $\implies$  *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*.



$$\begin{aligned}
& (ys \neq xs \vee x \neq y) \longrightarrow \\
& \text{igSubst MOD } ys \text{ (igVar MOD } ys \ y1) \ y \text{ (igVar MOD } xs \ x) = \text{igVar MOD } xs \ x) \\
& \wedge \\
& (\forall \ y \ y \ Y \ xs \ x. \\
& \quad \text{igWls MOD (asSort } ys) \ Y \longrightarrow \\
& \quad (ys \neq xs \vee y \neq x) \longrightarrow \\
& \quad \text{igSubst MOD } ys \ Y \ y \text{ (igVar MOD } xs \ x) = \text{igVar MOD } xs \ x)
\end{aligned}$$

**lemma** *igSubstIGVar1STR-imp-igSubstIGVar1*:  
*igSubstIGVar1STR MOD*  $\implies$  *igSubstIGVar1 MOD*  
 {proof}

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

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

**lemma** *igSubstIGVar2STR-imp-igSubstIGVar2*:  
*igSubstIGVar2STR MOD*  $\implies$  *igSubstIGVar2 MOD*  
 {proof}

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

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

**lemma** *igSubstIGAbsSTR-imp-igSubstIGAbs*:  
*igSubstIGAbsSTR MOD*  $\implies$  *igSubstIGAbs MOD*  
 {proof}

**definition** *igSubstIGOp* where

$$\begin{aligned}
& \text{igSubstIGOp MOD} == \\
& \forall \text{ ys y Y delta inp binp.} \\
& \quad \text{igWls MOD (asSort ys) Y} \wedge \\
& \quad \text{igWlsInp MOD delta inp} \wedge \text{igWlsBinp MOD delta binp} \longrightarrow \\
& \quad \text{igSubst MOD ys Y y (igOp MOD delta inp binp)} = \\
& \quad \text{igOp MOD delta (igSubstInp MOD ys Y y inp) (igSubstBinp MOD ys Y y binp)}
\end{aligned}$$

**definition** *igSubstIGOpSTR* where

$$\begin{aligned}
& \text{igSubstIGOpSTR MOD} == \\
& (\forall \text{ ys y y1 delta inp binp.} \\
& \quad \text{igSubst MOD ys (igVar MOD ys y1) y (igOp MOD delta inp binp)} = \\
& \quad \text{igOp MOD delta (igSubstInp MOD ys (igVar MOD ys y1) y inp)} \\
& \quad \quad (\text{igSubstBinp MOD ys (igVar MOD ys y1) y binp})) \\
& \wedge \\
& (\forall \text{ ys y Y delta inp binp.} \\
& \quad \text{igWls MOD (asSort ys) Y} \longrightarrow \\
& \quad \text{igSubst MOD ys Y y (igOp MOD delta inp binp)} = \\
& \quad \text{igOp MOD delta (igSubstInp MOD ys Y y inp) (igSubstBinp MOD ys Y y binp)})
\end{aligned}$$

**lemma** *igSubstIGOpSTR-imp-igSubstIGOp*:

$$\text{igSubstIGOpSTR MOD} \Longrightarrow \text{igSubstIGOp MOD}$$

*<proof>*

**definition** *igSubstCls* where

$$\begin{aligned}
& \text{igSubstCls MOD} == \\
& \text{igSubstIGVar1 MOD} \wedge \text{igSubstIGVar2 MOD} \wedge \\
& \text{igSubstIGAbs MOD} \wedge \\
& \text{igSubstIGOp MOD}
\end{aligned}$$

**lemmas** *igSubstCls-defs = igSubstCls-def*

*igSubstIGVar1-def igSubstIGVar2-def*

*igSubstIGAbs-def*

*igSubstIGOp-def*

**definition** *igSubstClsSTR* where

$$\begin{aligned}
& \text{igSubstClsSTR MOD} == \\
& \text{igSubstIGVar1STR MOD} \wedge \text{igSubstIGVar2STR MOD} \wedge \\
& \text{igSubstIGAbsSTR MOD} \wedge \\
& \text{igSubstIGOpSTR MOD}
\end{aligned}$$

**lemmas** *igSubstClsSTR-defs = igSubstClsSTR-def*

*igSubstIGVar1STR-def igSubstIGVar2STR-def*

*igSubstIGAbsSTR-def*

*igSubstIGOpSTR-def*

**lemma** *igSubstClsSTR-imp-igSubstCls*:

$$\text{igSubstClsSTR MOD} \Longrightarrow \text{igSubstCls MOD}$$

*<proof>*

**definition** *igAbsCongS* **where**

*igAbsCongS* *MOD* ==

$\forall xs\ x\ x'\ y\ s\ X\ X'.$

$isInBar\ (xs,s) \wedge igWls\ MOD\ s\ X \wedge igWls\ MOD\ s\ X' \longrightarrow$

$igFresh\ MOD\ xs\ y\ X \wedge igFresh\ MOD\ xs\ y\ X' \wedge igSwap\ MOD\ xs\ y\ x\ X = igSwap$   
 $MOD\ xs\ y\ x'\ X' \longrightarrow$

$igAbs\ MOD\ xs\ x\ X = igAbs\ MOD\ xs\ x'\ X'$

**definition** *igAbsCongSSTR* **where**

*igAbsCongSSTR* *MOD* ==

$\forall xs\ x\ x'\ y\ X\ X'.$

$igFresh\ MOD\ xs\ y\ X \wedge igFresh\ MOD\ xs\ y\ X' \wedge igSwap\ MOD\ xs\ y\ x\ X = igSwap$   
 $MOD\ xs\ y\ x'\ X' \longrightarrow$

$igAbs\ MOD\ xs\ x\ X = igAbs\ MOD\ xs\ x'\ X'$

**lemma** *igAbsCongSSTR-imp-igAbsCongS*:

$igAbsCongSSTR\ MOD \Longrightarrow igAbsCongS\ MOD$

*<proof>*

**definition** *igAbsCongU* **where**

*igAbsCongU* *MOD* ==

$\forall xs\ x\ x'\ y\ s\ X\ X'.$

$isInBar\ (xs,s) \wedge igWls\ MOD\ s\ X \wedge igWls\ MOD\ s\ X' \longrightarrow$

$igFresh\ MOD\ xs\ y\ X \wedge igFresh\ MOD\ xs\ y\ X' \wedge$

$igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)\ x\ X = igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)$   
 $x'\ X' \longrightarrow$

$igAbs\ MOD\ xs\ x\ X = igAbs\ MOD\ xs\ x'\ X'$

**definition** *igAbsCongUSTR* **where**

*igAbsCongUSTR* *MOD* ==

$\forall xs\ x\ x'\ y\ X\ X'.$

$igFresh\ MOD\ xs\ y\ X \wedge igFresh\ MOD\ xs\ y\ X' \wedge$

$igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)\ x\ X = igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)$   
 $x'\ X' \longrightarrow$

$igAbs\ MOD\ xs\ x\ X = igAbs\ MOD\ xs\ x'\ X'$

**lemma** *igAbsCongUSTR-imp-igAbsCongU*:

$igAbsCongUSTR\ MOD \Longrightarrow igAbsCongU\ MOD$

*<proof>*

**definition** *igAbsRen* **where**

*igAbsRen* *MOD* ==

$\forall xs\ y\ x\ s\ X.$

$isInBar\ (xs,s) \wedge igWls\ MOD\ s\ X \longrightarrow$

$igFresh\ MOD\ xs\ y\ X \longrightarrow$

$igAbs\ MOD\ xs\ y\ (igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)\ x\ X) = igAbs\ MOD\ xs\ x$

*X*

**definition** *igAbsRenSTR* **where**

*igAbsRenSTR* *MOD* ==

$\forall xs\ y\ x\ X.$

$igFresh\ MOD\ xs\ y\ X \longrightarrow$

$igAbs\ MOD\ xs\ y\ (igSubst\ MOD\ xs\ (igVar\ MOD\ xs\ y)\ x\ X) = igAbs\ MOD\ xs\ x\ X$

**lemma** *igAbsRenSTR-imp-igAbsRen*:

*igAbsRenSTR* *MOD*  $\implies$  *igAbsRen* *MOD*

*<proof>*

**lemma** *igAbsRenSTR-imp-igAbsCongUSTR*:

*igAbsRenSTR* *MOD*  $\implies$  *igAbsCongUSTR* *MOD*

*<proof>*

Well-sorted fresh-swap models:

**definition** *iwlsFSw* **where**

*iwlsFSw* *MOD* ==

$igWlsAllDisj\ MOD \wedge igWlsAbsIsInBar\ MOD \wedge$

$igConsIPresIGWls\ MOD \wedge igSwapAllIPresIGWlsAll\ MOD \wedge$

$igFreshCls\ MOD \wedge igSwapCls\ MOD \wedge igAbsCongS\ MOD$

**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* ==

$igWlsAllDisj\ MOD \wedge igWlsAbsIsInBar\ MOD \wedge$

$igConsIPresIGWlsSTR\ MOD \wedge igSwapAllIPresIGWlsAllSTR\ MOD \wedge$

$igFreshClsSTR\ MOD \wedge igSwapClsSTR\ MOD \wedge igAbsCongSSTR\ MOD$

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

Well-sorted fresh-subst models:

**definition** *iwlsFSb* where  
*iwlsFSb MOD ==*  
*igWlsAllDisj MOD  $\wedge$  igWlsAbsIsInBar MOD  $\wedge$*   
*igConsIPresIGWls MOD  $\wedge$  igSubstAllIPresIGWlsAll MOD  $\wedge$*   
*igFreshCls MOD  $\wedge$  igSubstCls MOD  $\wedge$  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  $\wedge$  igWlsAbsIsInBar MOD  $\wedge$*   
*igConsIPresIGWlsSTR MOD  $\wedge$  igSubstAllIPresIGWlsAllSTR MOD  $\wedge$*   
*igFreshClsSTR MOD  $\wedge$  igSubstClsSTR MOD  $\wedge$  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  $\implies$  iwlsFSb MOD*  
 ⟨proof⟩

Well-sorted fresh-swap-subst-models

**definition** *iwlsFSwSb* **where**

*iwlsFSwSb* *MOD* ==

*iwlsFSw* *MOD*  $\wedge$  *igSubstAllIPresIGWlsAll* *MOD*  $\wedge$  *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*  $\wedge$  *igSwapAllIPresIGWlsAll* *MOD*  $\wedge$  *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* ==

$\forall$  *zs z1 z2 delta inp*.

*igWlsInp* *MOD* *delta inp*  $\longrightarrow$

*igWlsInp* *MOD* *delta* (*igSwapInp* *MOD* *zs z1 z2 inp*)

**definition** *igSwapInpIPresIGWlsInpSTR* **where**

*igSwapInpIPresIGWlsInpSTR* *MOD* ==

$\forall$  *zs z1 z2 delta inp*.

*igWlsInp* *MOD* *delta* (*igSwapInp* *MOD* *zs z1 z2 inp*) =

*igWlsInp* *MOD* *delta inp*

**definition** *igSubstInpIPresIGWlsInp* **where**

*igSubstInpIPresIGWlsInp* *MOD* ==

$\forall$  *ys y Y delta inp*.

*igWls* *MOD* (*asSort* *ys*) *Y*  $\wedge$  *igWlsInp* *MOD* *delta inp*  $\longrightarrow$

*igWlsInp* *MOD* *delta* (*igSubstInp* *MOD* *ys Y y inp*)

**definition** *igSubstInpIPresIGWlsInpSTR* **where**

*igSubstInpIPresIGWlsInpSTR* *MOD* ==

$\forall$  *ys y Y delta inp*.

*igWls* *MOD* (*asSort* *ys*) *Y*  $\longrightarrow$

*igWlsInp* *MOD* *delta* (*igSubstInp* *MOD* *ys Y y inp*) =

*igWlsInp* *MOD* *delta inp*

**lemma** *imp-igSwapInpIPresIGWlsInp*:  
 $igSwapIPresIGWls\ MOD \implies igSwapInpIPresIGWlsInp\ MOD$   
 ⟨proof⟩

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

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

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

Then for bound inputs:

**definition** *igSwapBinpIPresIGWlsBinp* **where**  
 $igSwapBinpIPresIGWlsBinp\ MOD ==$   
 $\forall\ zs\ z1\ z2\ delta\ binp.$   
 $igWlsBinp\ MOD\ delta\ binp \longrightarrow$   
 $igWlsBinp\ MOD\ delta\ (igSwapBinp\ MOD\ zs\ z1\ z2\ binp)$

**definition** *igSwapBinpIPresIGWlsBinpSTR* **where**  
 $igSwapBinpIPresIGWlsBinpSTR\ MOD ==$   
 $\forall\ zs\ z1\ z2\ delta\ binp.$   
 $igWlsBinp\ MOD\ delta\ (igSwapBinp\ MOD\ zs\ z1\ z2\ binp) =$   
 $igWlsBinp\ MOD\ delta\ binp$

**definition** *igSubstBinpIPresIGWlsBinp* **where**  
 $igSubstBinpIPresIGWlsBinp\ MOD ==$   
 $\forall\ ys\ y\ Y\ delta\ binp.$   
 $igWls\ MOD\ (asSort\ ys)\ Y \wedge igWlsBinp\ MOD\ delta\ binp \longrightarrow$   
 $igWlsBinp\ MOD\ delta\ (igSubstBinp\ MOD\ ys\ Y\ y\ binp)$

**definition** *igSubstBinpIPresIGWlsBinpSTR* **where**  
 $igSubstBinpIPresIGWlsBinpSTR\ MOD ==$   
 $\forall\ ys\ y\ Y\ delta\ binp.$   
 $igWls\ MOD\ (asSort\ ys)\ Y \longrightarrow$   
 $igWlsBinp\ MOD\ delta\ (igSubstBinp\ MOD\ ys\ Y\ y\ binp) =$   
 $igWlsBinp\ MOD\ delta\ binp$

**lemma** *imp-igSwapBinpIPresIGWlsBinp*:  
 $igSwapAbsIPresIGWlsAbs\ MOD \implies igSwapBinpIPresIGWlsBinp\ MOD$   
 ⟨proof⟩

**lemma** *imp-igSwapBinpIPresIGWlsBinpSTR*:  
 $igSwapAbsIPresIGWlsAbsSTR\ MOD \implies igSwapBinpIPresIGWlsBinpSTR\ MOD$

*<proof>*

**lemma** *imp-igSubstBinpIPresIGWlsBinp*:

*igSubstAbsIPresIGWlsAbs MOD*  $\implies$  *igSubstBinpIPresIGWlsBinp MOD*

*<proof>*

**lemma** *imp-igSubstBinpIPresIGWlsBinpSTR*:

*igSubstAbsIPresIGWlsAbsSTR MOD*  $\implies$  *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'* ==

$\forall s X. \text{igWls } MOD \ s \ X \longrightarrow \text{igWls } MOD' \ s \ (h \ X)$

**definition** *ipresIGWlsAbs where*

*ipresIGWlsAbs hA MOD MOD'* ==

$\forall us \ s \ A. \text{igWlsAbs } MOD \ (us, s) \ A \longrightarrow \text{igWlsAbs } MOD' \ (us, s) \ (hA \ A)$

**definition** *ipresIGWlsAll where*

*ipresIGWlsAll h hA MOD MOD'* ==

$\text{ipresIGWls } h \ MOD \ MOD' \wedge \text{ipresIGWlsAbs } hA \ MOD \ MOD'$

**lemmas** *ipresIGWlsAll-defs = ipresIGWlsAll-def*

*ipresIGWls-def ipresIGWlsAbs-def*

### 8.2.2 Preservation of the constructs

**definition** *ipresIGVar where*

*ipresIGVar h MOD MOD'* ==

$\forall xs \ x. h \ (\text{igVar } MOD \ xs \ x) = \text{igVar } MOD' \ xs \ x$

**definition** *ipresIGAbs where*

*ipresIGAbs h hA MOD MOD'* ==

$\forall xs \ x \ s \ X. \text{isInBar } (xs, s) \wedge \text{igWls } MOD \ s \ X \longrightarrow$   
 $hA \ (\text{igAbs } MOD \ xs \ x \ X) = \text{igAbs } MOD' \ xs \ x \ (h \ X)$

**definition** *ipresIGOp*

**where**

*ipresIGOp h hA MOD MOD'* ==



$\forall \text{ delta inp binp.}$   
 $\text{igWlsInp MOD delta inp} \wedge \text{igWlsBinp MOD delta binp} \longrightarrow$   
 $h (\text{igOp MOD delta inp binp}) = \text{igOp MOD}' \text{ delta (lift h inp) (lift hA binp)}$

**definition** *ipresIGCons* **where**  
 $\text{ipresIGCons } h \text{ hA MOD MOD}' ==$   
 $\text{ipresIGVar } h \text{ MOD MOD}' \wedge$   
 $\text{ipresIGAbs } h \text{ hA MOD MOD}' \wedge$   
 $\text{ipresIGOp } h \text{ hA MOD MOD}'$

**lemmas** *ipresIGCons-defs* = *ipresIGCons-def*  
*ipresIGVar-def*  
*ipresIGAbs-def*  
*ipresIGOp-def*

### 8.2.3 Preservation of freshness

**definition** *ipresIGFresh* **where**  
 $\text{ipresIGFresh } h \text{ MOD MOD}' ==$   
 $\forall \text{ ys y s X.}$   
 $\text{igWls MOD s X} \longrightarrow$   
 $\text{igFresh MOD ys y X} \longrightarrow \text{igFresh MOD}' \text{ ys y (h X)}$

**definition** *ipresIGFreshAbs* **where**  
 $\text{ipresIGFreshAbs } hA \text{ MOD MOD}' ==$   
 $\forall \text{ ys y us s A.}$   
 $\text{igWlsAbs MOD (us,s) A} \longrightarrow$   
 $\text{igFreshAbs MOD ys y A} \longrightarrow \text{igFreshAbs MOD}' \text{ ys y (hA A)}$

**definition** *ipresIGFreshAll* **where**  
 $\text{ipresIGFreshAll } h \text{ hA MOD MOD}' ==$   
 $\text{ipresIGFresh } h \text{ MOD MOD}' \wedge \text{ipresIGFreshAbs } hA \text{ MOD MOD}'$

**lemmas** *ipresIGFreshAll-defs* = *ipresIGFreshAll-def*  
*ipresIGFresh-def* *ipresIGFreshAbs-def*

### 8.2.4 Preservation of swapping

**definition** *ipresIGSwap* **where**  
 $\text{ipresIGSwap } h \text{ MOD MOD}' ==$   
 $\forall \text{ zs z1 z2 s X.}$   
 $\text{igWls MOD s X} \longrightarrow$   
 $h (\text{igSwap MOD zs z1 z2 X}) = \text{igSwap MOD}' \text{ zs z1 z2 (h X)}$

**definition** *ipresIGSwapAbs* **where**  
 $\text{ipresIGSwapAbs } hA \text{ MOD MOD}' ==$   
 $\forall \text{ zs z1 z2 us s A.}$   
 $\text{igWlsAbs MOD (us,s) A} \longrightarrow$   
 $hA (\text{igSwapAbs MOD zs z1 z2 A}) = \text{igSwapAbs MOD}' \text{ zs z1 z2 (hA A)}$

**definition** *ipresIGSwapAll* **where**  
*ipresIGSwapAll*  $h$   $hA$   $MOD$   $MOD'$  ==  
*ipresIGSwap*  $h$   $MOD$   $MOD'$   $\wedge$  *ipresIGSwapAbs*  $hA$   $MOD$   $MOD'$

**lemmas** *ipresIGSwapAll-defs* = *ipresIGSwapAll-def*  
*ipresIGSwap-def* *ipresIGSwapAbs-def*

### 8.2.5 Preservation of subst

**definition** *ipresIGSubst* **where**  
*ipresIGSubst*  $h$   $MOD$   $MOD'$  ==  
 $\forall$   $ys$   $Y$   $y$   $s$   $X$ .  
*igWls*  $MOD$  (*asSort*  $ys$ )  $Y$   $\wedge$  *igWls*  $MOD$   $s$   $X$   $\longrightarrow$   
 $h$  (*igSubst*  $MOD$   $ys$   $Y$   $y$   $X$ ) = *igSubst*  $MOD'$   $ys$  ( $h$   $Y$ )  $y$  ( $h$   $X$ )

**definition** *ipresIGSubstAbs* **where**  
*ipresIGSubstAbs*  $h$   $hA$   $MOD$   $MOD'$  ==  
 $\forall$   $ys$   $Y$   $y$   $us$   $s$   $A$ .  
*igWls*  $MOD$  (*asSort*  $ys$ )  $Y$   $\wedge$  *igWlsAbs*  $MOD$  ( $us, s$ )  $A$   $\longrightarrow$   
 $hA$  (*igSubstAbs*  $MOD$   $ys$   $Y$   $y$   $A$ ) = *igSubstAbs*  $MOD'$   $ys$  ( $h$   $Y$ )  $y$  ( $hA$   $A$ )

**definition** *ipresIGSubstAll* **where**  
*ipresIGSubstAll*  $h$   $hA$   $MOD$   $MOD'$  ==  
*ipresIGSubst*  $h$   $MOD$   $MOD'$   $\wedge$   
*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*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$  *ipresIGCons*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$   
*ipresIGFreshAll*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$  *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*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$  *ipresIGCons*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$   
*ipresIGFreshAll*  $h$   $hA$   $MOD$   $MOD'$   $\wedge$  *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'  $\wedge$  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'  $\wedge$  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  $\implies$  ipresIGWls h MOD MOD'*  
 *$\implies$  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**  $igWlsInp\ MOD'\ \delta\ (lift\ h\ inp)$   
 $\langle proof \rangle$

**lemma**  $FSwSbImorph-igWlsInp$ :  
**assumes**  $igWlsInp\ MOD\ \delta\ inp$  **and**  $FSwSbImorph\ h\ hA\ MOD\ MOD'$   
**shows**  $igWlsInp\ MOD'\ \delta\ (lift\ h\ inp)$   
 $\langle proof \rangle$

Similar facts for bound inpus:

**lemma**  $igSwapBinp-None[simp]$ :  
 $(igSwapBinp\ MOD\ zs\ z1\ z2\ binp\ i = None) = (binp\ i = None)$   
 $\langle proof \rangle$

**lemma**  $igSubstBinp-None[simp]$ :  
 $(igSubstBinp\ MOD\ ys\ Y\ y\ binp\ i = None) = (binp\ i = None)$   
 $\langle proof \rangle$

**lemma**  $imp-igWlsBinp$ :  
**assumes**  $*$ :  $igWlsBinp\ MOD\ \delta\ binp$   
**and**  $**$ :  $ipresIGWlsAbs\ hA\ MOD\ MOD'$   
**shows**  $igWlsBinp\ MOD'\ \delta\ (lift\ hA\ binp)$   
 $\langle proof \rangle$

**corollary**  $FSwImorph-igWlsBinp$ :  
**assumes**  $igWlsBinp\ MOD\ \delta\ binp$  **and**  $FSwImorph\ h\ hA\ MOD\ MOD'$   
**shows**  $igWlsBinp\ MOD'\ \delta\ (lift\ hA\ binp)$   
 $\langle proof \rangle$

**corollary**  $FSbImorph-igWlsBinp$ :  
**assumes**  $igWlsBinp\ MOD\ \delta\ binp$  **and**  $FSbImorph\ h\ hA\ MOD\ MOD'$   
**shows**  $igWlsBinp\ MOD'\ \delta\ (lift\ hA\ binp)$   
 $\langle proof \rangle$

**lemma**  $FSwSbImorph-igWlsBinp$ :  
**assumes**  $igWlsBinp\ MOD\ \delta\ binp$  **and**  $FSwSbImorph\ h\ hA\ MOD\ MOD'$   
**shows**  $igWlsBinp\ MOD'\ \delta\ (lift\ hA\ binp)$   
 $\langle proof \rangle$

**lemmas**  $input-igSwap-igSubst-None =$   
 $igSwapInp-None\ igSubstInp-None$   
 $igSwapBinp-None\ igSubstBinp-None$

### 8.2.10 Identity and composition

**lemma**  $id-FSwImorph$ :  $FSwImorph\ id\ id\ MOD\ MOD$   
 $\langle proof \rangle$

**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**  $ipresIGSubstAll (h' \circ h) (hA' \circ hA) MOD MOD''$   
 $\langle proof \rangle$

**lemma** *comp-FSwImorph*:

**assumes** \*:  $FSwImorph h hA MOD MOD'$  **and** \*\*:  $FSwImorph h' hA' MOD' MOD''$

**shows**  $FSwImorph (h' \circ h) (hA' \circ hA) MOD MOD''$   
 $\langle proof \rangle$

**lemma** *comp-FSbImorph*:

**assumes**  $igWlsAbsIsInBar MOD$

**and**  $FSbImorph h hA MOD MOD'$  **and**  $FSbImorph h' hA' MOD' MOD''$

**shows**  $FSbImorph (h' \circ h) (hA' \circ hA) MOD MOD''$   
 $\langle proof \rangle$

**lemma** *comp-FSwSbImorph*:

**assumes**  $igWlsAbsIsInBar MOD$

**and**  $FSwSbImorph h hA MOD MOD'$  **and**  $FSwSbImorph h' hA' MOD' MOD''$

**shows**  $FSwSbImorph (h' \circ h) (hA' \circ hA) MOD MOD''$   
 $\langle proof \rangle$

### 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]*:  $igWls termMOD = wls$   
 $\langle proof \rangle$

**lemma** *igWlsAbs-termMOD[simp]*:  $igWlsAbs termMOD = wlsAbs$   
 $\langle proof \rangle$

**lemma** *igWlsInp-termMOD-wlsInp[simp]*:  
 $igWlsInp termMOD delta inp = wlsInp delta inp$   
 $\langle proof \rangle$

**lemma** *igWlsBinp-termMOD-wlsBinp[simp]*:  
 $igWlsBinp termMOD delta binp = wlsBinp delta binp$   
 $\langle proof \rangle$

**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* ==

$\forall us s A. wlsAbs (us,s) A \longrightarrow igWlsAbs MOD (us,s) (hA A)$

**lemma** *ipresIGWlsAbs-termMOD[simp]*:

*ipresIGWlsAbs* *hA termMOD MOD* = *ipresWlsAbs* *hA MOD*

*<proof>*

**definition** *ipresWlsAll* **where**

*ipresWlsAll* *h hA MOD* ==

*ipresWls* *h MOD*  $\wedge$  *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* ==

$\forall xs x. h (Var xs x) = igVar MOD xs x$

**lemma** *ipresIGVar-termMOD[simp]*:

*ipresIGVar* *h termMOD MOD* = *ipresVar* *h MOD*

*<proof>*

**definition** *ipresAbs* **where**

*ipresAbs* *h hA MOD* ==

$\forall xs x s X. isInBar (xs,s) \wedge wls s X \longrightarrow hA (Abs xs x X) = igAbs MOD xs x (h X)$

**lemma** *ipresIGAbs-termMOD[simp]*:

*ipresIGAbs* *h hA termMOD MOD* = *ipresAbs* *h hA MOD*

*<proof>*

**definition** *ipresOp* **where**

*ipresOp* *h hA MOD* ==

$\forall delta inp binp.$

$wlsInp delta inp \wedge wlsBinp delta binp \longrightarrow$

$h (Op delta inp binp) =$

$igOp MOD delta (lift h inp) (lift hA binp)$

**lemma** *ipresIGOpe-termMOD[simp]*:

*ipresIGOpe* *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* ==  
 $\forall$  *zs z1 z2 s X*.  
*wls* *s X*  $\longrightarrow$   
 $h (X \#[z1 \wedge z2]-zs) = igSwap \text{ MOD } zs \ z1 \ z2 (h \ X)$

**lemma** *ipresIGSwap-termMOD[simp]*:  
*ipresIGSwap* *h termMOD MOD* = *ipresSwap* *h MOD*  
*<proof>*

**definition** *ipresSwapAbs* **where**

*ipresSwapAbs* *hA MOD* ==  
 $\forall$  *zs z1 z2 us s A*.  
*wlsAbs* (*us,s*) *A*  $\longrightarrow$   
 $hA (A \#[z1 \wedge z2]-zs) = igSwapAbs \text{ MOD } zs \ z1 \ z2 (hA \ A)$

**lemma** *ipresIGSwapAbs-termMOD[simp]*:  
*ipresIGSwapAbs* *hA termMOD MOD* = *ipresSwapAbs* *hA MOD*  
*<proof>*

**definition** *ipresSwapAll* **where**

*ipresSwapAll* *h hA MOD* ==  
*ipresSwap* *h MOD*  $\wedge$  *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* ==  
 $\forall$  *ys Y y s X*.  
*wls* (*asSort ys*) *Y*  $\wedge$  *wls* *s X*  $\longrightarrow$   
 $h (\text{subst } ys \ Y \ y \ X) = igSubst \text{ MOD } ys (h \ Y) \ y (h \ X)$

**lemma** *ipresIGSubst-termMOD[simp]*:  
*ipresIGSubst* *h termMOD MOD* = *ipresSubst* *h MOD*  
*<proof>*

**definition** *ipresSubstAbs* **where**

*ipresSubstAbs* *h hA MOD* ==

$\forall$  *ys Y y us s A*.

*wls* (*asSort ys*) *Y*  $\wedge$  *wlsAbs* (*us,s*) *A*  $\longrightarrow$

*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*  $\wedge$  *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*  $\wedge$  *ipresCons h hA MOD*  $\wedge$

*ipresFreshAll h hA MOD*  $\wedge$  *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*  $\wedge$  *ipresCons h hA MOD*  $\wedge$

*ipresFreshAll h hA MOD*  $\wedge$  *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**  $FSbImorph-termMOD[simp]$ :  
 $FSbImorph h hA termMOD MOD = termFSbImorph h hA MOD$   
 $\langle proof \rangle$

**definition**  $termFSwSbImorph$  **where**  
 $termFSwSbImorph h hA MOD ==$   
 $termFSwImorph h hA MOD \wedge ipresSubstAll h hA MOD$

**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**  $termFSwSbImorph-iff$ :  
 $termFSwSbImorph h hA MOD =$   
 $(termFSbImorph h hA MOD \wedge ipresSwapAll h hA MOD)$   
 $\langle proof \rangle$

**lemma**  $FSwSbImorph-termMOD[simp]$ :  
 $FSwSbImorph h hA termMOD MOD = termFSwSbImorph h hA MOD$   
 $\langle proof \rangle$

**lemma**  $ipresWls-wlsInp$ :  
**assumes**  $wlsInp delta inp$  **and**  $ipresWls h MOD$   
**shows**  $igWlsInp MOD delta (lift h inp)$   
 $\langle proof \rangle$

**lemma**  $termFSwImorph-wlsInp$ :  
**assumes**  $wlsInp delta inp$  **and**  $termFSwImorph h hA MOD$   
**shows**  $igWlsInp MOD delta (lift h inp)$   
 $\langle proof \rangle$

**lemma**  $termFSwSbImorph-wlsInp$ :  
**assumes**  $wlsInp delta inp$  **and**  $termFSwSbImorph h hA MOD$   
**shows**  $igWlsInp MOD delta (lift h inp)$   
 $\langle proof \rangle$

**lemma**  $ipresWls-wlsBinp$ :  
**assumes**  $wlsBinp delta binp$  **and**  $ipresWlsAbs hA MOD$   
**shows**  $igWlsBinp MOD delta (lift hA binp)$   
 $\langle proof \rangle$



**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**  $ipresSubstAll\ h\ hA\ MOD$   
 $\langle proof \rangle$

**lemma**  $iwlsFSw-termFSwImorph-iff$ :  
 $iwlsFSw\ MOD \implies termFSwImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**corollary**  $iwlsFSwSTR-termFSwImorph-iff$ :  
 $iwlsFSwSTR\ MOD \implies termFSwImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**lemma**  $iwlsFSb-termFSbImorph-iff$ :  
 $iwlsFSb\ MOD \implies termFSbImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**corollary**  $iwlsFSbSwTR-termFSbImorph-iff$ :  
 $iwlsFSbSwTR\ MOD \implies termFSbImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**lemma**  $iwlsFSwSb-termFSwSbImorph-iff$ :  
 $iwlsFSwSb\ MOD \implies termFSwSbImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**lemma**  $iwlsFSbSw-termFSwSbImorph-iff$ :  
 $iwlsFSbSw\ MOD \implies termFSwSbImorph\ h\ hA\ MOD = ipresCons\ h\ hA\ MOD$   
 $\langle proof \rangle$

**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 target, 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*:  
 $eX \neq ERR \implies (EX!\ X.\ eX = OK\ X)$   
{*proof*}

**lemma** *check-OK[simp]*:  
 $check\ (OK\ X) = X$   
{*proof*}

**lemma** *OK-check[simp]*:  
 $eX \neq ERR \implies OK\ (check\ eX) = eX$   
{*proof*}

**lemma** *checkI-OKI[simp]*:  
 $checkI\ (OKI\ inp) = inp$   
{*proof*}

**lemma** *OKI-checkI[simp]*:  
**assumes** *liftAll* ( $\lambda\ X.\ X \neq ERR$ ) *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* ::  
 $('index, 'bindex, 'varSort, 'sort, 'opSym, 'var, 'gTerm, 'gAbs)model \implies$

('index,'bindex,'varSort,'sort,'opSym,'var,'gTerm withERR,'gAbs withERR)model  
**where**  
 errMOD MOD ==  
 (igWls =  $\lambda s eX$ . case eX of ERR  $\Rightarrow$  False | OK X  $\Rightarrow$  igWls MOD s X,  
 igWlsAbs =  $\lambda (us,s) eA$ . case eA of ERR  $\Rightarrow$  False | OK A  $\Rightarrow$  igWlsAbs MOD  
 (us,s) A,  
  
 igVar =  $\lambda xs x$ . OK (igVar MOD xs x),  
 igAbs =  $\lambda xs x eX$ .  
   if (eX  $\neq$  ERR  $\wedge$  ( $\exists s$ . isInBar (xs,s)  $\wedge$  igWls MOD s (check eX)))  
   then OK (igAbs MOD xs x (check eX))  
   else ERR,  
 igOp =  $\lambda delta einp ebinp$ .  
   if liftAll ( $\lambda X$ . X  $\neq$  ERR) einp  $\wedge$  liftAll ( $\lambda A$ . A  $\neq$  ERR) ebinp  
    $\wedge$  igWlsInp MOD delta (checkI einp)  $\wedge$  igWlsBinp MOD delta (checkI  
 ebinp)  
   then OK (igOp MOD delta (checkI einp) (checkI ebinp))  
   else ERR,  
 igFresh =  $\lambda ys y eX$ .  
   if eX  $\neq$  ERR  $\wedge$  ( $\exists s$ . igWls MOD s (check eX))  
   then igFresh MOD ys y (check eX)  
   else True,  
 igFreshAbs =  $\lambda ys y eA$ .  
   if eA  $\neq$  ERR  $\wedge$  ( $\exists us s$ . igWlsAbs MOD (us,s) (check eA))  
   then igFreshAbs MOD ys y (check eA)  
   else True,  
 igSwap =  $\lambda zs z1 z2 eX$ .  
   if eX  $\neq$  ERR  $\wedge$  ( $\exists s$ . igWls MOD s (check eX))  
   then OK (igSwap MOD zs z1 z2 (check eX))  
   else ERR,  
 igSwapAbs =  $\lambda zs z1 z2 eA$ .  
   if eA  $\neq$  ERR  $\wedge$  ( $\exists us s$ . igWlsAbs MOD (us,s) (check eA))  
   then OK (igSwapAbs MOD zs z1 z2 (check eA))  
   else ERR,  
 igSubst =  $\lambda ys eY y eX$ .  
   if eY  $\neq$  ERR  $\wedge$  igWls MOD (asSort ys) (check eY)  
    $\wedge$  eX  $\neq$  ERR  $\wedge$  ( $\exists s$ . igWls MOD s (check eX))  
   then OK (igSubst MOD ys (check eY) y (check eX))  
   else ERR,  
 igSubstAbs =  $\lambda ys eY y eA$ .  
   if eY  $\neq$  ERR  $\wedge$  igWls MOD (asSort ys) (check eY)  
    $\wedge$  eA  $\neq$  ERR  $\wedge$  ( $\exists us s$ . 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**  $eWlsBinp \text{ MOD} == igWlsBinp \text{ (errMOD MOD)}$   
**abbreviation**  $eVar$  **where**  $eVar \text{ MOD} == igVar \text{ (errMOD MOD)}$   
**abbreviation**  $eAbs$  **where**  $eAbs \text{ MOD} == igAbs \text{ (errMOD MOD)}$   
**abbreviation**  $eOp$  **where**  $eOp \text{ MOD} == igOp \text{ (errMOD MOD)}$   
**abbreviation**  $eFresh$  **where**  $eFresh \text{ MOD} == igFresh \text{ (errMOD MOD)}$   
**abbreviation**  $eFreshAbs$  **where**  $eFreshAbs \text{ MOD} == igFreshAbs \text{ (errMOD MOD)}$   
**abbreviation**  $eFreshInp$  **where**  $eFreshInp \text{ MOD} == igFreshInp \text{ (errMOD MOD)}$   
**abbreviation**  $eFreshBinp$  **where**  $eFreshBinp \text{ MOD} == igFreshBinp \text{ (errMOD MOD)}$   
**abbreviation**  $eSwap$  **where**  $eSwap \text{ MOD} == igSwap \text{ (errMOD MOD)}$   
**abbreviation**  $eSwapAbs$  **where**  $eSwapAbs \text{ MOD} == igSwapAbs \text{ (errMOD MOD)}$   
**abbreviation**  $eSwapInp$  **where**  $eSwapInp \text{ MOD} == igSwapInp \text{ (errMOD MOD)}$   
**abbreviation**  $eSwapBinp$  **where**  $eSwapBinp \text{ MOD} == igSwapBinp \text{ (errMOD MOD)}$   
**abbreviation**  $eSubst$  **where**  $eSubst \text{ MOD} == igSubst \text{ (errMOD MOD)}$   
**abbreviation**  $eSubstAbs$  **where**  $eSubstAbs \text{ MOD} == igSubstAbs \text{ (errMOD MOD)}$   
**abbreviation**  $eSubstInp$  **where**  $eSubstInp \text{ MOD} == igSubstInp \text{ (errMOD MOD)}$   
**abbreviation**  $eSubstBinp$  **where**  $eSubstBinp \text{ MOD} == igSubstBinp \text{ (errMOD MOD)}$

### 8.4.3 Simplification rules

**lemma**  $eWls-simp1[simp]$ :  
 $eWls \text{ MOD } s \text{ (OK } X) = igWls \text{ MOD } s \text{ } X$   
*<proof>*

**lemma**  $eWls-simp2[simp]$ :  
 $eWls \text{ MOD } s \text{ ERR} = False$   
*<proof>*

**lemma**  $eWlsAbs-simp1[simp]$ :  
 $eWlsAbs \text{ MOD } (us,s) \text{ (OK } A) = igWlsAbs \text{ MOD } (us,s) \text{ } A$   
*<proof>*

**lemma**  $eWlsAbs-simp2[simp]$ :  
 $eWlsAbs \text{ MOD } (us,s) \text{ ERR} = False$   
*<proof>*

**lemma**  $eWlsInp-simp1[simp]$ :  
 $eWlsInp \text{ MOD } delta \text{ (OKI } inp) = igWlsInp \text{ MOD } delta \text{ } inp$   
*<proof>*

**lemma**  $eWlsInp-simp2[simp]$ :  
 $\neg \text{liftAll } (\lambda eX. eX \neq ERR) \text{ } einp \implies \neg eWlsInp \text{ MOD } delta \text{ } einp$   
*<proof>*

**corollary**  $eWlsInp-simp3[simp]$ :  
 $\neg eWlsInp \text{ MOD } delta \text{ } (\lambda i. \text{Some } ERR)$   
*<proof>*

**lemma**  $eWlsBinp\text{-simp1}[simp]$ :  
 $eWlsBinp \text{ MOD } \delta (OKI \text{ binp}) = igWlsBinp \text{ MOD } \delta \text{ binp}$   
 $\langle proof \rangle$

**lemma**  $eWlsBinp\text{-simp2}[simp]$ :  
 $\neg liftAll (\lambda eA. eA \neq ERR) \text{ ebinp} \implies \neg eWlsBinp \text{ MOD } \delta \text{ ebinp}$   
 $\langle proof \rangle$

**corollary**  $eWlsBinp\text{-simp3}[simp]$ :  
 $\neg eWlsBinp \text{ MOD } \delta (\lambda i. Some \text{ ERR})$   
 $\langle proof \rangle$

**lemmas**  $eWlsAll\text{-simps} =$   
 $eWls\text{-simp1} \ eWls\text{-simp2}$   
 $eWlsAbs\text{-simp1} \ eWlsAbs\text{-simp2}$   
 $eWlsInp\text{-simp1} \ eWlsInp\text{-simp2} \ eWlsInp\text{-simp3}$   
 $eWlsBinp\text{-simp1} \ eWlsBinp\text{-simp2} \ eWlsBinp\text{-simp3}$

**lemma**  $eVar\text{-simp}[simp]$ :  
 $eVar \text{ MOD } xs \ x = OK (igVar \text{ MOD } xs \ x)$   
 $\langle proof \rangle$

**lemma**  $eAbs\text{-simp1}[simp]$ :  
 $\llbracket isInBar (xs, s); igWls \text{ MOD } s \ X \rrbracket \implies eAbs \text{ MOD } xs \ x (OK \ X) = OK (igAbs \text{ MOD } xs \ x \ X)$   
 $\langle proof \rangle$

**lemma**  $eAbs\text{-simp2}[simp]$ :  
 $\forall s. \neg (isInBar (xs, s) \wedge igWls \text{ MOD } s \ X) \implies eAbs \text{ MOD } xs \ x (OK \ X) = ERR$   
 $\langle proof \rangle$

**lemma**  $eAbs\text{-simp3}[simp]$ :  
 $eAbs \text{ MOD } xs \ x \text{ ERR} = ERR$   
 $\langle proof \rangle$

**lemma**  $eOp\text{-simp1}[simp]$ :  
**assumes**  $igWlsInp \text{ MOD } \delta \text{ inp}$  **and**  $igWlsBinp \text{ MOD } \delta \text{ binp}$   
**shows**  $eOp \text{ MOD } \delta (OKI \text{ inp}) (OKI \text{ binp}) = OK (igOp \text{ MOD } \delta \text{ inp} \text{ binp})$   
 $\langle proof \rangle$

**lemma**  $eOp\text{-simp2}[simp]$ :  
**assumes**  $\neg igWlsInp \text{ MOD } \delta \text{ inp}$   
**shows**  $eOp \text{ MOD } \delta (OKI \text{ inp}) \text{ ebinp} = ERR$   
 $\langle proof \rangle$

**lemma**  $eOp\text{-simp3}[simp]$ :  
**assumes**  $\neg igWlsBinp \text{ MOD } \delta \text{ binp}$   
**shows**  $eOp \text{ MOD } \delta \text{ einp} (OKI \text{ binp}) = ERR$

*<proof>*

**lemma**  $eOp\text{-simp4}[simp]$ :  
**assumes**  $\neg \text{liftAll } (\lambda eX. eX \neq ERR) \text{ einp}$   
**shows**  $eOp \text{ MOD } \text{delta einp ebinp} = ERR$   
*<proof>*

**corollary**  $eOp\text{-simp5}[simp]$ :  
 $eOp \text{ MOD } \text{delta } (\lambda i. \text{Some } ERR) \text{ ebinp} = ERR$   
*<proof>*

**lemma**  $eOp\text{-simp6}[simp]$ :  
**assumes**  $\neg \text{liftAll } (\lambda eA. eA \neq ERR) \text{ ebinp}$   
**shows**  $eOp \text{ MOD } \text{delta einp ebinp} = ERR$   
*<proof>*

**corollary**  $eOp\text{-simp7}[simp]$ :  
 $eOp \text{ MOD } \text{delta einp } (\lambda i. \text{Some } ERR) = ERR$   
*<proof>*

**lemmas**  $eCons\text{-sims} =$   
 $eVar\text{-simp}$   
 $eAbs\text{-simp1 } eAbs\text{-simp2 } eAbs\text{-simp3}$   
 $eOp\text{-simp1 } eOp\text{-simp2 } eOp\text{-simp3 } eOp\text{-simp4 } eOp\text{-simp5 } eOp\text{-simp6 } eOp\text{-simp7}$

**lemma**  $eFresh\text{-simp1}[simp]$ :  
 $igWls \text{ MOD } s X \implies eFresh \text{ MOD } ys y (OK X) = igFresh \text{ MOD } ys y X$   
*<proof>*

**lemma**  $eFresh\text{-simp2}[simp]$ :  
 $\forall s. \neg igWls \text{ MOD } s X \implies eFresh \text{ MOD } ys y (OK X)$   
*<proof>*

**lemma**  $eFresh\text{-simp3}[simp]$ :  
 $eFresh \text{ MOD } ys y ERR$   
*<proof>*

**lemma**  $eFreshAbs\text{-simp1}[simp]$ :  
 $igWlsAbs \text{ MOD } (us,s) A \implies eFreshAbs \text{ MOD } ys y (OK A) = igFreshAbs \text{ MOD } ys y A$   
*<proof>*

**lemma**  $eFreshAbs\text{-simp2}[simp]$ :  
 $\forall us s. \neg igWlsAbs \text{ MOD } (us,s) A \implies eFreshAbs \text{ MOD } ys y (OK A)$   
*<proof>*

**lemma**  $eFreshAbs\text{-simp3}[simp]$ :  
 $eFreshAbs \text{ MOD } ys y ERR$   
*<proof>*



**lemma** *eFreshInp-simp*[simp]:  
*igWlsInp MOD delta inp*  
 $\implies eFreshInp \text{ MOD } ys \ y \ (OKI \ inp) = igFreshInp \text{ MOD } ys \ y \ inp$   
 ⟨proof⟩

**lemma** *eFreshBinp-simp*[simp]:  
*igWlsBinp MOD delta binp*  
 $\implies eFreshBinp \text{ MOD } ys \ y \ (OKI \ binp) = igFreshBinp \text{ MOD } ys \ y \ binp$   
 ⟨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*  
 $\implies eSwap \text{ MOD } zs \ z1 \ z2 \ (OK \ X) = OK \ (igSwap \text{ MOD } zs \ z1 \ z2 \ X)$   
 ⟨proof⟩

**lemma** *eSwap-simp2*[simp]:  
 $\forall \ s. \neg \ igWls \text{ MOD } s \ X \implies eSwap \text{ MOD } zs \ z1 \ z2 \ (OK \ X) = ERR$   
 ⟨proof⟩

**lemma** *eSwap-simp3*[simp]:  
*eSwap MOD zs z1 z2 ERR = ERR*  
 ⟨proof⟩

**lemma** *eSwapAbs-simp1*[simp]:  
*igWlsAbs MOD (us,s) A*  
 $\implies eSwapAbs \text{ MOD } zs \ z1 \ z2 \ (OK \ A) = OK \ (igSwapAbs \text{ MOD } zs \ z1 \ z2 \ A)$   
 ⟨proof⟩

**lemma** *eSwapAbs-simp2*[simp]:  
 $\forall \ us \ s. \neg \ igWlsAbs \text{ MOD } (us,s) \ A \implies eSwapAbs \text{ MOD } zs \ z1 \ z2 \ (OK \ A) = ERR$   
 ⟨proof⟩

**lemma** *eSwapAbs-simp3*[simp]:  
*eSwapAbs MOD zs z1 z2 ERR = ERR*  
 ⟨proof⟩

**lemma** *eSwapInp-simp1*[simp]:  
*igWlsInp MOD delta inp*  
 $\implies eSwapInp \text{ MOD } zs \ z1 \ z2 \ (OKI \ inp) = OKI \ (igSwapInp \text{ MOD } zs \ z1 \ z2 \ inp)$   
 ⟨proof⟩

**lemma** *eSwapInp-simp2*[simp]:

**assumes**  $\neg \text{liftAll } (\lambda eX. eX \neq \text{ERR}) \text{ einp}$   
**shows**  $\neg \text{liftAll } (\lambda eX. eX \neq \text{ERR}) (e\text{SwapInp MOD } zs \ z1 \ z2 \ \text{einp})$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{SwapBinp-simp1}[\text{simp}]$ :  
 $igWlsBinp \text{ MOD } \text{delta } \text{binp}$   
 $\implies e\text{SwapBinp MOD } zs \ z1 \ z2 \ (\text{OKI } \text{binp}) = \text{OKI } (ig\text{SwapBinp MOD } zs \ z1 \ z2 \ \text{binp})$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{SwapBinp-simp2}[\text{simp}]$ :  
**assumes**  $\neg \text{liftAll } (\lambda eA. eA \neq \text{ERR}) \text{ ebinp}$   
**shows**  $\neg \text{liftAll } (\lambda eA. eA \neq \text{ERR}) (e\text{SwapBinp MOD } zs \ z1 \ z2 \ \text{ebinp})$   
 $\langle \text{proof} \rangle$

**lemmas**  $e\text{SwapAll-simps} =$   
 $e\text{Swap-simp1 } e\text{Swap-simp2 } e\text{Swap-simp3}$   
 $e\text{SwapAbs-simp1 } e\text{SwapAbs-simp2 } e\text{SwapAbs-simp3}$   
 $e\text{SwapInp-simp1 } e\text{SwapInp-simp2}$   
 $e\text{SwapBinp-simp1 } e\text{SwapBinp-simp2}$

**lemma**  $e\text{Subst-simp1}[\text{simp}]$ :  
 $\llbracket igWls \text{ MOD } (\text{asSort } ys) \ Y; \ igWls \text{ MOD } s \ X \rrbracket$   
 $\implies e\text{Subst MOD } ys \ (\text{OK } Y) \ y \ (\text{OK } X) = \text{OK } (ig\text{Subst MOD } ys \ Y \ y \ X)$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{Subst-simp2}[\text{simp}]$ :  
 $\neg \text{igWls MOD } (\text{asSort } ys) \ Y \implies e\text{Subst MOD } ys \ (\text{OK } Y) \ y \ eX = \text{ERR}$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{Subst-simp3}[\text{simp}]$ :  
 $\forall s. \neg \text{igWls MOD } s \ X \implies e\text{Subst MOD } ys \ eY \ y \ (\text{OK } X) = \text{ERR}$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{Subst-simp4}[\text{simp}]$ :  
 $e\text{Subst MOD } ys \ eY \ y \ \text{ERR} = \text{ERR}$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{Subst-simp5}[\text{simp}]$ :  
 $e\text{Subst MOD } ys \ \text{ERR} \ y \ eX = \text{ERR}$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{SubstAbs-simp1}[\text{simp}]$ :  
 $\llbracket igWls \text{ MOD } (\text{asSort } ys) \ Y; \ igWlsAbs \text{ MOD } (us, s) \ A \rrbracket$   
 $\implies e\text{SubstAbs MOD } ys \ (\text{OK } Y) \ y \ (\text{OK } A) = \text{OK } (ig\text{SubstAbs MOD } ys \ Y \ y \ A)$   
 $\langle \text{proof} \rangle$

**lemma**  $e\text{SubstAbs-simp2}[\text{simp}]$ :  
 $\neg \text{igWls MOD } (\text{asSort } ys) \ Y \implies e\text{SubstAbs MOD } ys \ (\text{OK } Y) \ y \ eA = \text{ERR}$

*<proof>*

**lemma** *eSubstAbs-simp3[simp]*:

$\forall us s. \neg \text{igWlsAbs MOD } (us, s) A \implies \text{eSubstAbs MOD } ys eY y (OK A) = ERR$

*<proof>*

**lemma** *eSubstAbs-simp4[simp]*:

$\text{eSubstAbs MOD } ys eY y ERR = ERR$

*<proof>*

**lemma** *eSubstAbs-simp5[simp]*:

$\text{eSubstAbs MOD } ys ERR y eA = ERR$

*<proof>*

**lemma** *eSubstInp-simp1[simp]*:

$\llbracket \text{igWls MOD } (asSort ys) Y; \text{igWlsInp MOD } delta \text{ inp} \rrbracket$

$\implies \text{eSubstInp MOD } ys (OK Y) y (OKI \text{ inp}) = OKI (\text{igSubstInp MOD } ys Y y \text{ inp})$

*<proof>*

**lemma** *eSubstInp-simp2[simp]*:

**assumes**  $\neg \text{liftAll } (\lambda eX. eX \neq ERR) \text{ einp}$

**shows**  $\neg \text{liftAll } (\lambda eX. eX \neq ERR) (\text{eSubstInp MOD } ys eY y \text{ einp})$

*<proof>*

**lemma** *eSubstInp-simp3[simp]*:

**assumes**  $*$ :  $\neg \text{igWls MOD } (asSort ys) Y$  **and**  $**$ :  $\neg \text{einp} = (\lambda i. None)$

**shows**  $\neg \text{liftAll } (\lambda eX. eX \neq ERR) (\text{eSubstInp MOD } ys (OK Y) y \text{ einp})$

*<proof>*

**lemma** *eSubstInp-simp4[simp]*:

**assumes**  $\neg \text{einp} = (\lambda i. None)$

**shows**  $\neg \text{liftAll } (\lambda eX. eX \neq ERR) (\text{eSubstInp MOD } ys ERR y \text{ einp})$

*<proof>*

**lemma** *eSubstBinp-simp1[simp]*:

$\llbracket \text{igWls MOD } (asSort ys) Y; \text{igWlsBinp MOD } delta \text{ binp} \rrbracket$

$\implies \text{eSubstBinp MOD } ys (OK Y) y (OKI \text{ binp}) = OKI (\text{igSubstBinp MOD } ys Y y \text{ binp})$

*<proof>*

**lemma** *eSubstBinp-simp2[simp]*:

**assumes**  $\neg \text{liftAll } (\lambda eA. eA \neq ERR) \text{ ebinp}$

**shows**  $\neg \text{liftAll } (\lambda eA. eA \neq ERR) (\text{eSubstBinp MOD } ys eY y \text{ ebinp})$

*<proof>*

**lemma** *eSubstBinp-simp3[simp]*:

**assumes**  $*$ :  $\neg \text{igWls MOD } (asSort ys) Y$  **and**  $**$ :  $\neg \text{ebinp} = (\lambda i. None)$

**shows**  $\neg \text{liftAll } (\lambda eA. eA \neq ERR) (\text{eSubstBinp MOD } ys (OK Y) y \text{ ebinp})$

$\langle proof \rangle$

**lemma**  $eSubstBinp-simp4[simp]$ :

**assumes**  $\neg ebinp = (\lambda i. None)$

**shows**  $\neg liftAll (\lambda eA. eA \neq ERR) (eSubstBinp \text{ MOD } ys \text{ ERR } y \text{ ebinp})$

$\langle proof \rangle$

**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**  $eWls-nchotomy$ :

$(\exists X. eX = OK \ X \wedge igWls \text{ MOD } s \ X) \vee \neg eWls \text{ MOD } s \ eX$

$\langle proof \rangle$

**lemma**  $eWlsAbs-nchotomy$ :

$(\exists A. eA = OK \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A) \vee \neg eWlsAbs \text{ MOD } (us,s) \ eA$

$\langle proof \rangle$

**lemma**  $eAbs-nchotomy$ :

$((\exists s \ X. eX = OK \ X \wedge isInBar \ (xs,s) \ \wedge \ igWls \text{ MOD } s \ X)) \vee (eAbs \text{ MOD } xs \ x$   
 $eX = ERR)$

$\langle proof \rangle$

**lemma**  $eOp-nchotomy$ :

$(\exists \text{ inp } \text{ binp}. \text{ einp} = OKI \ \text{inp} \wedge \text{ igWlsInp } \text{ MOD } \text{ delta } \ \text{inp} \wedge$   
 $\text{ ebinp} = OKI \ \text{binp} \wedge \text{ igWlsBinp } \text{ MOD } \text{ delta } \ \text{binp})$

$\vee$

$(eOp \text{ MOD } \text{ delta } \ \text{einp} \ \text{ebinp} = ERR)$

$\langle proof \rangle$

**lemma**  $eFresh-nchotomy$ :

$(\exists s \ X. eX = OK \ X \wedge \text{ igWls } \text{ MOD } s \ X) \vee eFresh \text{ MOD } ys \ y \ eX$

$\langle proof \rangle$

**lemma**  $eFreshAbs-nchotomy$ :

$(\exists us \ s \ A. eA = OK \ A \wedge \text{ igWlsAbs } \text{ MOD } (us,s) \ A)$

$\vee eFreshAbs \text{ MOD } ys \ y \ eA$   
 $\langle proof \rangle$

**lemma** *eSwap-nchotomy*:  
 $(\exists s \ X. \ eX = OK \ X \wedge igWls \text{ MOD } s \ X) \vee$   
 $(eSwap \text{ MOD } zs \ z1 \ z2 \ eX = ERR)$   
 $\langle proof \rangle$

**lemma** *eSwapAbs-nchotomy*:  
 $(\exists us \ s \ A. \ eA = OK \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A) \vee$   
 $(eSwapAbs \text{ MOD } zs \ z1 \ z2 \ eA = ERR)$   
 $\langle proof \rangle$

**lemma** *eSubst-nchotomy*:  
 $(\exists Y. \ eY = OK \ Y \wedge$   
 $igWls \text{ MOD } (asSort \ ys) \ Y) \wedge (\exists s \ X. \ eX = OK \ X \wedge igWls \text{ MOD } s \ X)$   
 $\vee$   
 $(eSubst \text{ MOD } ys \ eY \ y \ eX = ERR)$   
 $\langle proof \rangle$

**lemma** *eSubstAbs-nchotomy*:  
 $(\exists Y. \ eY = OK \ Y \wedge igWls \text{ MOD } (asSort \ ys) \ Y) \wedge$   
 $(\exists us \ s \ A. \ eA = OK \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A)$   
 $\vee$   
 $(eSubstAbs \text{ MOD } ys \ eY \ y \ eA = ERR)$   
 $\langle proof \rangle$

### 8.4.5 Inversion rules

**lemma** *eWls-invert*:  
**assumes**  $eWls \text{ MOD } s \ eX$   
**shows**  $\exists X. \ eX = OK \ X \wedge igWls \text{ MOD } s \ X$   
 $\langle proof \rangle$

**lemma** *eWlsAbs-invert*:  
**assumes**  $eWlsAbs \text{ MOD } (us,s) \ eA$   
**shows**  $\exists A. \ eA = OK \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A$   
 $\langle proof \rangle$

**lemma** *eWlsInp-invert*:  
**assumes**  $eWlsInp \text{ MOD } delta \ einp$   
**shows**  $\exists inp. \ igWlsInp \text{ MOD } delta \ inp \wedge einp = OKI \ inp$   
 $\langle proof \rangle$

**lemma** *eWlsBinp-invert*:  
**assumes**  $eWlsBinp \text{ MOD } delta \ ebinp$   
**shows**  $\exists binp. \ igWlsBinp \text{ MOD } delta \ binp \wedge ebinp = OKI \ binp$   
 $\langle proof \rangle$

**lemma** *eAbs-invert*:

**assumes**  $eAbs \text{ MOD } xs \ x \ eX = OK \ A$

**shows**  $\exists \ s \ X. \ eX = OK \ X \wedge isInBar \ (xs,s) \wedge A = igAbs \text{ MOD } xs \ x \ X \wedge igWls \text{ MOD } s \ X$

*<proof>*

**lemma** *eOp-invert*:

**assumes**  $eOp \text{ MOD } delta \ einp \ ebinp = OK \ X$

**shows**

$\exists \ inp \ binp. \ einp = OKI \ inp \wedge ebinp = OKI \ binp \wedge$

$X = igOp \text{ MOD } delta \ inp \ binp \wedge$

$igWlsInp \text{ MOD } delta \ inp \wedge igWlsBinp \text{ MOD } delta \ binp$

*<proof>*

**lemma** *eFresh-invert*:

**assumes**  $\neg \ eFresh \text{ MOD } ys \ y \ eX$

**shows**  $\exists \ s \ X. \ eX = OK \ X \wedge \neg \ igFresh \text{ MOD } ys \ y \ X \wedge igWls \text{ MOD } s \ X$

*<proof>*

**lemma** *eFreshAbs-invert*:

**assumes**  $\neg \ eFreshAbs \text{ MOD } ys \ y \ eA$

**shows**  $\exists \ us \ s \ A. \ eA = OK \ A \wedge \neg \ igFreshAbs \text{ MOD } ys \ y \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A$

*<proof>*

**lemma** *eSwap-invert*:

**assumes**  $eSwap \text{ MOD } zs \ z1 \ z2 \ eX = OK \ Y$

**shows**  $\exists \ s \ X. \ eX = OK \ X \wedge Y = igSwap \text{ MOD } zs \ z1 \ z2 \ X \wedge igWls \text{ MOD } s \ X$

*<proof>*

**lemma** *eSwapAbs-invert*:

**assumes**  $eSwapAbs \text{ MOD } zs \ z1 \ z2 \ eA = OK \ B$

**shows**  $\exists \ us \ s \ A. \ eA = OK \ A \wedge B = igSwapAbs \text{ MOD } zs \ z1 \ z2 \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A$

*<proof>*

**lemma** *eSubst-invert*:

**assumes**  $eSubst \text{ MOD } ys \ eY \ y \ eX = OK \ Z$

**shows**

$\exists \ s \ X \ Y. \ eY = OK \ Y \wedge eX = OK \ X \wedge igWls \text{ MOD } s \ X \wedge igWls \text{ MOD } (asSort \ ys) \ Y \wedge$

$Z = igSubst \text{ MOD } ys \ Y \ y \ X$

*<proof>*

**lemma** *eSubstAbs-invert*:

**assumes**  $eSubstAbs \text{ MOD } ys \ eY \ y \ eA = OK \ Z$

**shows**

$\exists \ us \ s \ A \ Y. \ eY = OK \ Y \wedge eA = OK \ A \wedge igWlsAbs \text{ MOD } (us,s) \ A \wedge igWls \text{ MOD } (asSort \ ys) \ Y \wedge$

$Z = igSubstAbs\ MOD\ ys\ Y\ y\ A$   
(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 morphism from an error model to its original model

This morphism is given 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**

$(qFresh\ ys\ y\ qX \longrightarrow igFresh\ MOD\ ys\ y\ (qInit\ MOD\ qX)) \wedge$   
 $(qFreshAbs\ ys\ y\ qA \longrightarrow igFreshAbs\ MOD\ ys\ y\ (qInitAbs\ MOD\ qA))$   
 $\langle proof \rangle$

**corollary** *iwlsFSwSTR-qFreshAll-igFreshAll-qInitAll:*

**assumes** *iwlsFSwSTR MOD*

**shows**

$(qFresh\ ys\ y\ qX \longrightarrow igFresh\ MOD\ ys\ y\ (qInit\ MOD\ qX)) \wedge$   
 $(qFreshAbs\ ys\ y\ qA \longrightarrow igFreshAbs\ MOD\ ys\ y\ (qInitAbs\ MOD\ qA))$   
 $\langle proof \rangle$

**corollary** *iwlsFSbSwTR-qFreshAll-igFreshAll-qInitAll:*

**assumes** *iwlsFSbSwTR MOD*

**shows**

$(qFresh\ ys\ y\ qX \longrightarrow igFresh\ MOD\ ys\ y\ (qInit\ MOD\ qX)) \wedge$   
 $(qFreshAbs\ ys\ y\ qA \longrightarrow igFreshAbs\ MOD\ ys\ y\ (qInitAbs\ MOD\ qA))$   
 $\langle proof \rangle$

**lemma** *qSwapAll-igSwapAll-qInitAll:*

**assumes** *igSwapClsSTR MOD*

**shows**

$qInit\ MOD\ (qX\ \#[[z1\ \wedge\ z2]]-zs) = igSwap\ MOD\ zs\ z1\ z2\ (qInit\ MOD\ qX) \wedge$   
 $qInitAbs\ MOD\ (qA\ \$[[z1\ \wedge\ z2]]-zs) = igSwapAbs\ MOD\ zs\ z1\ z2\ (qInitAbs\ MOD\ qA)$   
 $\langle proof \rangle$

**corollary** *iwlsFSwSTR-qSwapAll-igSwapAll-qInitAll:*

**assumes** *wls: iwlsFSwSTR MOD*

**shows**

$qInit\ MOD\ (qX\ \#[[z1\ \wedge\ z2]]-zs) = igSwap\ MOD\ zs\ z1\ z2\ (qInit\ MOD\ qX) \wedge$   
 $qInitAbs\ MOD\ (qA\ \$[[z1\ \wedge\ z2]]-zs) = igSwapAbs\ MOD\ zs\ z1\ z2\ (qInitAbs\ MOD\ qA)$   
 $\langle proof \rangle$

**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**

$(qGood\ qX \longrightarrow$   
 $(\forall\ ys\ y1\ y.$   
 $qAFresh\ ys\ y1\ qX \longrightarrow$   
 $qInit\ MOD\ (qX\ \#[[y1\ \wedge\ y]]-ys) = igSubst\ MOD\ ys\ (igVar\ MOD\ ys\ y1)\ y\ (qInit$   
 $MOD\ qX)))$   
 $\wedge$

$(qGoodAbs\ qA \longrightarrow$   
 $(\forall\ ys\ y1\ y.$   
 $\quad qAFreshAbs\ ys\ y1\ qA \longrightarrow$   
 $\quad qInitAbs\ MOD\ (qA\ \$[[y1\ \wedge\ y]]-ys) = igSubstAbs\ MOD\ ys\ (igVar\ MOD\ ys\ y1)$   
 $\quad y\ (qInitAbs\ MOD\ qA))$   
 $\langle proof \rangle$

**lemma** *iwlsFSbSwTR-qSwapAll-igSubstAll-qInitAll:*

**assumes** *wls: iwlsFSbSwTR MOD*

**shows**

$(qGood\ qX \longrightarrow$   
 $\quad qAFresh\ ys\ y1\ qX \longrightarrow$   
 $\quad qInit\ MOD\ (qX\ \#[[y1\ \wedge\ y]]-ys) = igSubst\ MOD\ ys\ (igVar\ MOD\ ys\ y1)\ y\ (qInit$   
 $\quad MOD\ qX))$   
 $\wedge$   
 $(qGoodAbs\ qA \longrightarrow$   
 $\quad qAFreshAbs\ ys\ y1\ qA \longrightarrow$   
 $\quad qInitAbs\ MOD\ (qA\ \$[[y1\ \wedge\ y]]-ys) = igSubstAbs\ MOD\ ys\ (igVar\ MOD\ ys\ y1)\ y$   
 $\quad (qInitAbs\ MOD\ qA))$   
 $\langle proof \rangle$

**lemma** *iwlsFSwSTR-alphaAll-qInitAll:*

**assumes** *iwlsFSwSTR MOD*

**shows**

$(\forall\ qX'.\ qX\ \#\ =\ qX' \longrightarrow qInit\ MOD\ qX = qInit\ MOD\ qX') \wedge$   
 $(\forall\ qA'.\ qA\ \$\ =\ qA' \longrightarrow qInitAbs\ MOD\ qA = qInitAbs\ MOD\ qA')$   
 $\langle proof \rangle$

**corollary** *iwlsFSwSTR-qInit-respectsP-alpha:*

**assumes** *iwlsFSwSTR MOD* **shows**  $(qInit\ MOD)$  *respectsP alpha*  
 $\langle proof \rangle$

**corollary** *iwlsFSwSTR-qInitAbs-respectsP-alphaAbs:*

**assumes** *iwlsFSwSTR MOD* **shows**  $(qInitAbs\ MOD)$  *respectsP alphaAbs*  
 $\langle proof \rangle$

**lemma** *iwlsFSbSwTR-alphaAll-qInitAll:*

**fixes**  $qX::('index,'bindex,'varSort,'var,'opSym)qTerm$  **and**

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

**assumes** *iwlsFSbSwTR MOD*

**shows**

$(qGood\ qX \longrightarrow (\forall\ qX'.\ qX\ \#\ =\ qX' \longrightarrow qInit\ MOD\ qX = qInit\ MOD\ qX')) \wedge$   
 $(qGoodAbs\ qA \longrightarrow (\forall\ qA'.\ qA\ \$\ =\ qA' \longrightarrow qInitAbs\ MOD\ qA = qInitAbs\ MOD$   
 $\quad qA'))$   
 $\langle proof \rangle$

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

**lemma** *iwlsFSbSwTR-iterSTR-ipresCons:*  
**assumes** *iwlsFSbSwTR MOD*  
**shows** *ipresCons (iterSTR MOD) (iterAbsSTR MOD) MOD*  
 $\langle$ *proof* $\rangle$

**lemma** *iwlsFSwSTR-iterSTR-termFSwImorph:*  
**assumes** *iwlsFSwSTR MOD*  
**shows** *termFSwImorph (iterSTR MOD) (iterAbsSTR MOD) MOD*  
 $\langle$ *proof* $\rangle$

**corollary** *iterSTR-termFSwImorph-errMOD:*  
**assumes** *iwlsFSw MOD*  
**shows**  
*termFSwImorph (iterSTR (errMOD MOD))*  
                  *(iterAbsSTR (errMOD MOD))*  
                  *(errMOD MOD)*  
 $\langle$ *proof* $\rangle$

**lemma** *iwlsFSbSwTR-iterSTR-termFSbImorph:*  
**assumes** *iwlsFSbSwTR MOD*  
**shows** *termFSbImorph (iterSTR MOD) (iterAbsSTR MOD) MOD*  
 $\langle$ *proof* $\rangle$

**corollary** *iterSTR-termFSbImorph-errMOD:*  
**assumes** *iwlsFSb MOD*  
**shows**  
*termFSbImorph (iterSTR (errMOD MOD))*  
                  *(iterAbsSTR (errMOD MOD))*  
                  *(errMOD MOD)*  
 $\langle$ *proof* $\rangle$

**declare** *qItem-simps[simp del]*  
**declare** *qItem-versus-item-simps[simp del]*  
**declare** *good-item-simps[simp del]*

### 8.5.3 The initial morphism (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” morphism – 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:*

$iwlsFSw\ MOD \implies termFSwImorph\ (iter\ MOD)\ (iterAbs\ MOD)\ MOD$

$\langle proof \rangle$

**theorem** *iwlsFSb-iterAll-termFSbImorph:*

$iwlsFSb\ MOD \implies termFSbImorph\ (iter\ MOD)\ (iterAbs\ MOD)\ MOD$

$\langle proof \rangle$

**theorem** *iwlsFSwSb-iterAll-termFSwSbImorph:*

$iwlsFSwSb\ MOD \implies termFSwSbImorph\ (iter\ MOD)\ (iterAbs\ MOD)\ MOD$

$\langle proof \rangle$

**theorem** *iwlsFSbSw-iterAll-termFSwSbImorph:*

$iwlsFSbSw\ MOD \implies termFSwSbImorph\ (iter\ MOD)\ (iterAbs\ MOD)\ MOD$

$\langle proof \rangle$

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**

$(wls\ s\ X \longrightarrow f\ X = ig\ X) \wedge$

$(wlsAbs\ (us, s')\ A \longrightarrow fA\ A = igA\ A)$

$\langle proof \rangle$

**theorem** *iwlsFSw-iterAll-unique-ipresCons:*

**assumes**  $iwlsFSw\ MOD$  **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** *iwlsFSb-iterAll-unique-ipresCons:*

**assumes**  $iwlsFSb\ MOD$  **and**  $ipresCons\ h\ hA\ MOD$

**shows**

$(wls\ s\ X \longrightarrow h\ X = iter\ MOD\ X) \wedge$

$(wlsAbs\ (us, s')\ A \longrightarrow hA\ A = iterAbs\ MOD\ A)$

$\langle proof \rangle$

**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<sup>6</sup> of corresponding arity, where:

---

<sup>6</sup>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.<sup>7</sup>

In particular, for the untyped and simply-typed  $\lambda$ -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**  $(\text{'varSort}, \text{'sTerm})sAbs = sAbs \text{'varSort} \text{'sTerm} \Rightarrow \text{'sTerm}$

**record**  $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'sTerm})semDom =$   
 $sWls :: \text{'sort} \Rightarrow \text{'sTerm} \Rightarrow bool$   
 $sDummy :: \text{'sort} \Rightarrow \text{'sTerm}$   
 $sOp :: \text{'opSym} \Rightarrow (\text{'index}, \text{'sTerm})input \Rightarrow (\text{'bindex}, (\text{'varSort}, \text{'sTerm})sAbs)input$   
 $\Rightarrow \text{'sTerm}$

The type of valuations:

**type-synonym**  $(\text{'varSort}, \text{'var}, \text{'sTerm})val = \text{'varSort} \Rightarrow \text{'var} \Rightarrow \text{'sTerm}$

**context**  $FixSyn$

**begin**

**fun**  $sWlsAbs$  **where**

$sWlsAbs SEM (xs, s) (sAbs xs' sF) =$   
 $(isInBar (xs, s) \wedge xs = xs' \wedge$   
 $(\forall sX. \text{if } sWls SEM (asSort xs) sX$   
 $\text{then } sWls SEM s (sF sX)$   
 $\text{else } sF sX = sDummy SEM s))$

**definition**  $sWlsInp$  **where**

$sWlsInp SEM delta sinp \equiv$   
 $wlsOpS delta \wedge sameDom (arOf delta) sinp \wedge liftAll2 (sWls SEM) (arOf delta)$   
 $sinp$

---

<sup>7</sup>In other words, syntactic bindings are captured semantically as functional bindings.

**definition** *sWlsBinp* **where**

*sWlsBinp SEM delta sbinp*  $\equiv$   
 $wlsOpS\ delta \wedge sameDom\ (barOf\ delta)\ sbinp \wedge liftAll2\ (sWlsAbs\ SEM)\ (barOf\ delta)\ sbinp$

**definition** *sWlsNE* **where**

*sWlsNE SEM*  $\equiv$   
 $\forall s. \exists sX. sWls\ SEM\ s\ sX$

**definition** *sWlsDisj* **where**

*sWlsDisj SEM*  $\equiv$   
 $\forall s\ s'\ sX. sWls\ SEM\ s\ sX \wedge sWls\ SEM\ s'\ sX \longrightarrow s = s'$

**definition** *sOpPrSWls* **where**

*sOpPrSWls SEM*  $\equiv$   
 $\forall\ delta\ sinp\ sbinp.$   
 $sWlsInp\ SEM\ delta\ sinp \wedge sWlsBinp\ SEM\ delta\ sbinp$   
 $\longrightarrow sWls\ SEM\ (stOf\ delta)\ (sOp\ SEM\ delta\ sinp\ sbinp)$

The notion of a “well-sorted” (better read as “well-structured”) semantic domain: <sup>8</sup>

**definition** *wlsSEM* **where**

*wlsSEM SEM*  $\equiv$   
 $sWlsNE\ SEM \wedge sWlsDisj\ SEM \wedge sOpPrSWls\ SEM$

The properties described in the next 4 definitions turn out to be consequences of the well-structuredness of the semantic domain:

**definition** *sWlsAbsNE* **where**

*sWlsAbsNE SEM*  $\equiv$   
 $\forall\ us\ s. isInBar\ (us,s) \longrightarrow (\exists\ sA. sWlsAbs\ SEM\ (us,s)\ sA)$

**definition** *sWlsAbsDisj* **where**

*sWlsAbsDisj SEM*  $\equiv$   
 $\forall\ us\ s\ us'\ s'\ sA.$   
 $isInBar\ (us,s) \wedge isInBar\ (us',s') \wedge sWlsAbs\ SEM\ (us,s)\ sA \wedge sWlsAbs\ SEM\ (us',s')\ sA$   
 $\longrightarrow us = us' \wedge s = s'$

The notion of two valuations being equal everywhere but on a given variable:

**definition** *eqBut* **where**

*eqBut val val' xs x*  $\equiv$   
 $\forall\ ys\ y. (ys = xs \wedge y = x) \vee val\ ys\ y = val'\ ys\ y$

**definition** *updVal* **::**

$(\prime varSort, \prime var, \prime sTerm)val \Rightarrow$   
 $\prime var \Rightarrow \prime sTerm \Rightarrow \prime varSort \Rightarrow$

---

<sup>8</sup>As usual in Isabelle, we first define the “raw” version, and then “fix” it with a well-structuredness predicate.

$(\text{'varSort}, \text{'var}, \text{'sTerm})\text{val} (- \text{'(- := -)'} \text{'-- } 200)$

**where**

$(\text{val } (x := sX)\text{-xs}) \equiv$

$\lambda \text{ys } y. (\text{if } \text{ys} = \text{xs} \wedge y = x \text{ then } sX \text{ else val ys } y)$

**definition**  $\text{swapVal} ::$

$\text{'varSort} \Rightarrow \text{'var} \Rightarrow \text{'var} \Rightarrow (\text{'varSort}, \text{'var}, \text{'sTerm})\text{val} \Rightarrow$

$(\text{'varSort}, \text{'var}, \text{'sTerm})\text{val}$

**where**

$\text{swapVal } \text{zs } z1 \ z2 \ \text{val} \equiv \lambda \text{xs } x. \ \text{val } \text{xs} \ (x \ @\text{xs}[z1 \ \wedge \ z2]\text{-zs})$

**abbreviation**  $\text{swapVal}\text{-abbrev} (- \ \text{'[- \ \wedge \ -]'} \text{'-- } 200)$  **where**

$\text{val } \text{'[- \ \wedge \ -]}\text{-zs} \equiv \text{swapVal } \text{zs } z1 \ z2 \ \text{val}$

**definition**  $sWlsVal$  **where**

$sWlsVal \ \text{SEM } \text{val} \equiv$

$\forall \ \text{ys } y. \ sWls \ \text{SEM} \ (\text{asSort } \text{ys}) \ (\text{val } \text{ys } y)$

**definition**  $sWlsValNE ::$

$(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'sTerm})\text{semDom} \Rightarrow \text{'var} \Rightarrow \text{bool}$

**where**

$sWlsValNE \ \text{SEM } x \equiv \exists \ (\text{val} :: (\text{'varSort}, \text{'var}, \text{'sTerm})\text{val}). \ sWlsVal \ \text{SEM } \text{val}$

### 9.1.2 Basic facts

**lemma**  $sWlsNE\text{-imp-}sWlsAbsNE:$

**assumes**  $sWlsNE \ \text{SEM}$

**shows**  $sWlsAbsNE \ \text{SEM}$

$\langle \text{proof} \rangle$

**lemma**  $sWlsDisj\text{-imp-}sWlsAbsDisj:$

$sWlsDisj \ \text{SEM} \Longrightarrow sWlsNE \ \text{SEM} \Longrightarrow sWlsAbsDisj \ \text{SEM}$

$\langle \text{proof} \rangle$

**lemma**  $sWlsNE\text{-imp-}sWlsValNE:$

$sWlsNE \ \text{SEM} \Longrightarrow sWlsValNE \ \text{SEM } x$

$\langle \text{proof} \rangle$

**theorem**  $\text{updVal}\text{-simp}[\text{simp}]:$

$(\text{val } (x := sX)\text{-xs}) \ \text{ys } y = (\text{if } \text{ys} = \text{xs} \wedge y = x \text{ then } sX \text{ else val ys } y)$

$\langle \text{proof} \rangle$

**theorem**  $\text{updVal}\text{-over}[\text{simp}]:$

$((\text{val } (x := sX)\text{-xs}) \ (x := sX')\text{-xs}) = (\text{val } (x := sX')\text{-xs})$

$\langle \text{proof} \rangle$

**theorem**  $\text{updVal}\text{-commute}:$

**assumes**  $xs \neq ys \vee x \neq y$   
**shows**  $((val\ (x := sX)-xs)\ (y := sY)-ys) = ((val\ (y := sY)-ys)\ (x := sX)-xs)$   
 $\langle proof \rangle$

**theorem** *updVal-preserves-sWls[simp]*:  
**assumes**  $sWls\ SEM\ (asSort\ xs)\ sX$  **and**  $sWlsVal\ SEM\ val$   
**shows**  $sWlsVal\ SEM\ (val\ (x := sX)-xs)$   
 $\langle proof \rangle$

**lemmas**  $updVal-simps = updVal-simp\ updVal-over\ updVal-preserves-sWls$

**theorem** *swapVal-ident[simp]*:  $(val\ \lceil x \wedge x \rceil -xs) = val$   
 $\langle proof \rangle$

**theorem** *swapVal-compose*:  
 $((val\ \lceil x \wedge y \rceil -zs)\ \lceil x' \wedge y' \rceil -zs') =$   
 $((val\ \lceil x' @zs'[x \wedge y] -zs \wedge y' @zs'[x \wedge y] -zs' \rceil -zs)\ \lceil x \wedge y \rceil -zs)$   
 $\langle proof \rangle$

**theorem** *swapVal-commute*:  
 $zs \neq zs' \vee \{x, y\} \cap \{x', y'\} = \{\} \implies$   
 $((val\ \lceil x \wedge y \rceil -zs)\ \lceil x' \wedge y' \rceil -zs') = ((val\ \lceil x' \wedge y' \rceil -zs')\ \lceil x \wedge y \rceil -zs)$   
 $\langle proof \rangle$

**lemma** *swapVal-involutive[simp]*:  $((val\ \lceil x \wedge y \rceil -zs)\ \lceil x \wedge y \rceil -zs) = val$   
 $\langle proof \rangle$

**lemma** *swapVal-sym*:  $(val\ \lceil x \wedge y \rceil -zs) = (val\ \lceil y \wedge x \rceil -zs)$   
 $\langle proof \rangle$

**lemma** *swapVal-preserves-sWls1*:  
**assumes**  $sWlsVal\ SEM\ val$   
**shows**  $sWlsVal\ SEM\ (val\ \lceil z1 \wedge z2 \rceil -zs)$   
 $\langle proof \rangle$

**theorem** *swapVal-preserves-sWls[simp]*:  
 $sWlsVal\ SEM\ (val\ \lceil z1 \wedge z2 \rceil -zs) = sWlsVal\ SEM\ val$   
 $\langle proof \rangle$

**lemmas**  $swapVal-simps = swapVal-ident\ swapVal-involutive\ swapVal-preserves-sWls$

**lemma** *updVal-swapVal*:  
 $((val\ (x := sX)-xs)\ \lceil y1 \wedge y2 \rceil -ys) =$   
 $((val\ \lceil y1 \wedge y2 \rceil -ys)\ ((x @xs[y1 \wedge y2] -ys) := sX)-xs)$   
 $\langle proof \rangle$

**lemma** *updVal-preserves-eqBut*:  
**assumes**  $eqBut\ val\ val'\ ys\ y$   
**shows**  $eqBut\ (val\ (x := sX)-xs)\ (val'\ (x := sX)-xs)\ ys\ y$

*<proof>*

**lemma** *updVal-eqBut-eq*:

**assumes** *eqBut val val' ys y*

**shows**  $(val (y := sY)-ys) = (val' (y := sY)-ys)$

*<proof>*

**lemma** *swapVal-preserves-eqBut*:

**assumes** *eqBut val val' xs x*

**shows**  $eqBut (val \hat{\sim}[z1 \wedge z2]-zs) (val' \hat{\sim}[z1 \wedge z2]-zs) xs (x @xs[z1 \wedge z2]-zs)$

*<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 compositionality 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*

$prWls g SEM \equiv \forall s X val.$

$wls s X \wedge sWlsVal SEM val$

$\longrightarrow sWls SEM s (g X val)$

**definition** *prWlsAbs where*

$prWlsAbs gA SEM \equiv \forall us s A val.$

$wlsAbs (us,s) A \wedge sWlsVal SEM val$

$\longrightarrow sWlsAbs SEM (us,s) (gA A val)$

**definition** *prWlsAll where*

$prWlsAll g gA SEM \equiv prWls g SEM \wedge prWlsAbs gA SEM$

**definition** *prVar where*

$prVar g SEM \equiv \forall xs x val.$

$sWlsVal SEM val \longrightarrow g (Var xs x) val = val xs x$

**definition** *prAbs where*

$prAbs g gA SEM \equiv \forall xs s x X val.$

$isInBar (xs,s) \wedge wls s X \wedge sWlsVal SEM val$

$\longrightarrow$

$gA (Abs xs x X) val =$

$sAbs xs (\lambda sX. \text{if } sWls SEM (asSort xs) sX \text{ then } g X (val (x := sX)-xs) \\ \text{else } sDummy SEM s)$

**definition** *prOp where*



$prOp\ g\ gA\ SEM \equiv \forall\ delta\ inp\ binp\ val.$   
 $wlsInp\ delta\ inp \wedge wlsBinp\ delta\ binp \wedge sWlsVal\ SEM\ val$   
 $\longrightarrow$   
 $g\ (Op\ delta\ inp\ binp)\ val =$   
 $sOp\ SEM\ delta\ (lift\ (\lambda X. g\ X\ val)\ inp)$   
 $(lift\ (\lambda A. gA\ A\ val)\ binp)$

**definition** *prCons* **where**

$prCons\ g\ gA\ SEM \equiv prVar\ g\ SEM \wedge prAbs\ g\ gA\ SEM \wedge prOp\ g\ gA\ SEM$

**definition** *prFresh* **where**

$prFresh\ g\ SEM \equiv \forall\ ys\ y\ s\ X\ val\ val'.$   
 $wls\ s\ X \wedge fresh\ ys\ y\ X \wedge$   
 $sWlsVal\ SEM\ val \wedge sWlsVal\ SEM\ val' \wedge eqBut\ val\ val'\ ys\ y$   
 $\longrightarrow g\ X\ val = g\ X\ val'$

**definition** *prFreshAbs* **where**

$prFreshAbs\ gA\ SEM \equiv \forall\ ys\ y\ us\ s\ A\ val\ val'.$   
 $wlsAbs\ (us,s)\ A \wedge freshAbs\ ys\ y\ A \wedge$   
 $sWlsVal\ SEM\ val \wedge sWlsVal\ SEM\ val' \wedge eqBut\ val\ val'\ ys\ y$   
 $\longrightarrow gA\ A\ val = gA\ A\ val'$

**definition** *prFreshAll* **where**

$prFreshAll\ g\ gA\ SEM \equiv prFresh\ g\ SEM \wedge prFreshAbs\ gA\ SEM$

**definition** *prSwap* **where**

$prSwap\ g\ SEM \equiv \forall\ zs\ z1\ z2\ s\ X\ val.$   
 $wls\ s\ X \wedge sWlsVal\ SEM\ val$   
 $\longrightarrow$   
 $g\ (X\ \#[z1 \wedge z2]-zs)\ val =$   
 $g\ X\ (val\ \hat{\wedge}[z1 \wedge z2]-zs)$

**definition** *prSwapAbs* **where**

$prSwapAbs\ gA\ SEM \equiv \forall\ zs\ z1\ z2\ us\ s\ A\ val.$   
 $wlsAbs\ (us,s)\ A \wedge sWlsVal\ SEM\ val$   
 $\longrightarrow$   
 $gA\ (A\ \$[z1 \wedge z2]-zs)\ val =$   
 $gA\ A\ (val\ \hat{\wedge}[z1 \wedge z2]-zs)$

**definition** *prSwapAll* **where**

$prSwapAll\ g\ gA\ SEM \equiv prSwap\ g\ SEM \wedge prSwapAbs\ gA\ SEM$

**definition** *prSubst* **where**

$prSubst\ g\ SEM \equiv \forall\ ys\ Y\ y\ s\ X\ val.$   
 $wls\ (asSort\ ys)\ Y \wedge wls\ s\ X$   
 $\wedge sWlsVal\ SEM\ val$   
 $\longrightarrow$   
 $g\ (X\ \#[Y / y]-ys)\ val =$   
 $g\ X\ (val\ (y := g\ Y\ val)-ys)$

**definition** *prSubstAbs* **where**

*prSubstAbs* *g gA SEM*  $\equiv \forall$  *ys Y y us s A val*.  
*wls* (*asSort ys*) *Y*  $\wedge$  *wlsAbs* (*us,s*) *A*  
 $\wedge$  *sWlsVal SEM val*  
 $\longrightarrow$   
*gA* (*A*  $\$[Y / y]$ -*ys*) *val* =  
*gA* *A* (*val* (*y* := *g Y val*)-*ys*)

**definition** *prSubstAll* **where**

*prSubstAll* *g gA SEM*  $\equiv$  *prSubst* *g SEM*  $\wedge$  *prSubstAbs* *g gA SEM*

**definition** *compInt* **where**

*compInt* *g gA SEM*  $\equiv$  *prWlsAll* *g gA SEM*  $\wedge$  *prCons* *g gA SEM*  $\wedge$   
*prFreshAll* *g gA SEM*  $\wedge$  *prSwapAll* *g gA SEM*  $\wedge$  *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* ( $\lambda$  *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* ( $\lambda$  *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”.<sup>9</sup> 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 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

---

<sup>9</sup>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: <sup>10</sup>

**type-synonym**  $(\text{'varSort}, \text{'var}, \text{'sTerm}) \text{gTerm} = (\text{'varSort}, \text{'var}, \text{'sTerm}) \text{val} \Rightarrow \text{'sTerm}$

**type-synonym**  $(\text{'varSort}, \text{'var}, \text{'sTerm}) \text{gAbs} = (\text{'varSort}, \text{'var}, \text{'sTerm}) \text{val} \Rightarrow (\text{'varSort}, \text{'sTerm}) \text{sAbs}$

**context** *FixSyn*  
**begin**

**definition** *asIMOD* ::

$(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'sTerm}) \text{semDom} \Rightarrow$   
 $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'var},$   
 $(\text{'varSort}, \text{'var}, \text{'sTerm}) \text{gTerm},$   
 $(\text{'varSort}, \text{'var}, \text{'sTerm}) \text{gAbs}) \text{model}$

**where**

*asIMOD SEM*  $\equiv$

$(\text{igWls} = \lambda s X. \forall \text{val}. (\text{sWlsVal SEM val} \vee X \text{val} = \text{undefined}) \wedge$   
 $(\text{sWlsVal SEM val} \longrightarrow \text{sWls SEM s (X val)}),$   
 $\text{igWlsAbs} = \lambda (xs, s) A. \forall \text{val}. (\text{sWlsVal SEM val} \vee A \text{val} = \text{undefined}) \wedge$   
 $(\text{sWlsVal SEM val} \longrightarrow \text{sWlsAbs SEM (xs, s) (A val)}),$   
 $\text{igVar} = \lambda \text{ys } y. \lambda \text{val}. \text{if } \text{sWlsVal SEM val} \text{ then } \text{val ys } y \text{ else } \text{undefined},$   
 $\text{igAbs} =$   
 $\lambda xs \ x \ X. \lambda \text{val}. \text{if } \text{sWlsVal SEM val}$   
 $\text{then } \text{sAbs xs } (\lambda s X. \text{if } \text{sWls SEM (asSort xs) sX}$   
 $\text{then } X (\text{val (x := sX)-xs})$   
 $\text{else } \text{sDummy SEM (SOME s. sWls SEM s (X$   
 $\text{val}))}$   
 $\text{else } \text{undefined},$   
 $\text{igOp} = \lambda \text{delta inp binp}. \lambda \text{val}.$   
 $\text{if } \text{sWlsVal SEM val} \text{ then } \text{sOp SEM delta (lift } (\lambda X. X \text{val}) \text{ inp)}$   
 $(\text{lift } (\lambda A. A \text{val}) \text{ binp})$   
 $\text{else } \text{undefined},$

---

<sup>10</sup>Recall that "generalized items" inhabit models.

$$\begin{aligned}
igFresh &= \\
\lambda ys y X. \forall val val'. sWlsVal SEM val \wedge sWlsVal SEM val' \wedge eqBut val val' ys y \\
&\quad \longrightarrow X val = X val', \\
igFreshAbs &= \\
\lambda ys y A. \forall val val'. sWlsVal SEM val \wedge sWlsVal SEM val' \wedge eqBut val val' ys y \\
&\quad \longrightarrow A val = A val', \\
igSwap &= \lambda zs z1 z2 X. \lambda val. \text{if } sWlsVal SEM val \text{ then } X (val \overline{[z1 \wedge z2]}-zs) \\
&\quad \text{else undefined}, \\
igSwapAbs &= \lambda zs z1 z2 A. \lambda val. \text{if } sWlsVal SEM val \text{ then } A (val \overline{[z1 \wedge z2]}-zs) \\
&\quad \text{else undefined}, \\
igSubst &= \lambda ys Y y X. \lambda val. \text{if } sWlsVal SEM val \text{ then } X (val (y := Y val)-ys) \\
&\quad \text{else undefined}, \\
igSubstAbs &= \lambda ys Y y A. \lambda val. \text{if } sWlsVal SEM val \text{ then } A (val (y := Y val)-ys) \\
&\quad \text{else undefined}
\end{aligned}$$

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*:

$$\begin{aligned}
igWls (asIMOD SEM) s X &\longleftrightarrow \\
(\forall val. (sWlsVal SEM val \vee X val = \text{undefined}) \wedge \\
&\quad (sWlsVal SEM val \longrightarrow sWls SEM s (X val))) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *asIMOD-igWlsAbs*:

$$\begin{aligned}
igWlsAbs (asIMOD SEM) (us,s) A &\longleftrightarrow \\
(\forall val. (sWlsVal SEM val \vee A val = \text{undefined}) \wedge \\
&\quad (sWlsVal SEM val \longrightarrow sWlsAbs SEM (us,s) (A val))) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *asIMOD-igOp*:

$$\begin{aligned}
igOp (asIMOD SEM) delta inp binp &= \\
(\lambda val. \text{if } sWlsVal SEM val \text{ then } sOp SEM delta (\text{lift } (\lambda X. X val) inp) \\
&\quad (\text{lift } (\lambda A. A val) binp) \\
&\quad \text{else undefined}) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *asIMOD-igVar*:

$$\begin{aligned}
igVar (asIMOD SEM) ys y &= (\lambda val. \text{if } sWlsVal SEM val \text{ then } val ys y \text{ else unde-} \\
&\quad \text{fined}) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *asIMOD-igAbs*:

$$\begin{aligned}
igAbs (asIMOD SEM) xs x X &= \\
(\lambda val. \text{if } sWlsVal SEM val \text{ then } sAbs xs (\lambda sX. \text{if } sWls SEM (asSort xs) sX \\
&\quad \text{then } X (val (x := sX)-xs) \\
&\quad \text{else } sDummy SEM (SOME s. sWls SEM s \\
&\quad (X val))) \\
&\quad \text{else undefined})
\end{aligned}$$

*<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 =*

*(λval. if sWlsVal SEM val then sAbs xs (λsX. 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 =*

*(λval. if sWlsVal SEM val then X (val  $\hat{\sim}$ [z1 ∧ z2]-zs) else undefined)*

*<proof>*

**lemma** *asIMOD-igSwapAbs*:

*igSwapAbs (asIMOD SEM) zs z1 z2 A =*

*(λval. if sWlsVal SEM val then A (val  $\hat{\sim}$ [z1 ∧ z2]-zs) else undefined)*

*<proof>*

**lemma** *asIMOD-igSubst*:

*igSubst (asIMOD SEM) ys Y y X =*

*(λval. if sWlsVal SEM val then X (val (y := Y val)-ys) else undefined)*

*<proof>*

**lemma** *asIMOD-igSubstAbs*:

*igSubstAbs (asIMOD SEM) ys Y y A =*

*(λval. 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  $\longleftrightarrow$*

*(∀ val. liftAll (λX. sWlsVal SEM val ∨ X val = undefined) inp) ∧*

$(\forall \text{ val. } sWlsVal \text{ SEM } \text{val} \longrightarrow sWlsInp \text{ SEM } \text{delta} (\text{lift } (\lambda X. X \text{ val}) \text{ inp}))$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igSwapInp*:  
 $sWlsVal \text{ SEM } \text{val} \Longrightarrow$   
 $\text{lift } (\lambda X. X \text{ val}) (\text{igSwapInp } (asIMOD \text{ SEM}) \text{ zs } z1 \text{ z2 } \text{inp}) =$   
 $\text{lift } (\lambda X. X (\text{swapVal } \text{zs } z1 \text{ z2 } \text{val})) \text{inp}$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igSubstInp*:  
 $sWlsVal \text{ SEM } \text{val} \Longrightarrow$   
 $\text{lift } (\lambda X. X \text{ val}) (\text{igSubstInp } (asIMOD \text{ SEM}) \text{ ys } Y \text{ y } \text{inp}) =$   
 $\text{lift } (\lambda X. X (\text{val } (\text{y} := Y \text{ val})\text{-ys})) \text{inp}$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igWlsBinp*:  
**assumes**  $sWlsNE \text{ SEM}$   
**shows**  
 $\text{igWlsBinp } (asIMOD \text{ SEM}) \text{ delta } \text{binp} =$   
 $(\forall \text{ val. } \text{liftAll } (\lambda X. sWlsVal \text{ SEM } \text{val} \vee X \text{ val} = \text{undefined}) \text{binp}) \wedge$   
 $(\forall \text{ val. } sWlsVal \text{ SEM } \text{val} \longrightarrow sWlsBinp \text{ SEM } \text{delta} (\text{lift } (\lambda X. X \text{ val}) \text{binp}))$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igSwapBinp*:  
 $sWlsVal \text{ SEM } \text{val} \Longrightarrow$   
 $\text{lift } (\lambda A. A \text{ val}) (\text{igSwapBinp } (asIMOD \text{ SEM}) \text{ zs } z1 \text{ z2 } \text{binp}) =$   
 $\text{lift } (\lambda A. A (\text{swapVal } \text{zs } z1 \text{ z2 } \text{val})) \text{binp}$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igSubstBinp*:  
 $sWlsVal \text{ SEM } \text{val} \Longrightarrow$   
 $\text{lift } (\lambda A. A \text{ val}) (\text{igSubstBinp } (asIMOD \text{ SEM}) \text{ ys } Y \text{ y } \text{binp}) =$   
 $\text{lift } (\lambda A. A (\text{val } (\text{y} := Y \text{ val})\text{-ys})) \text{binp}$   
 $\langle \text{proof} \rangle$

### 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*:  
 $sWlsNE \text{ SEM} \Longrightarrow sWlsDisj \text{ SEM} \Longrightarrow \text{igWlsDisj } (asIMOD \text{ SEM})$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igWlsAbsDisj*:  
 $sWlsNE \text{ SEM} \Longrightarrow sWlsDisj \text{ SEM} \Longrightarrow \text{igWlsAbsDisj } (asIMOD \text{ SEM})$   
 $\langle \text{proof} \rangle$

**lemma** *asIMOD-igWlsAllDisj*:  
 $sWlsNE\ SEM \implies sWlsDisj\ SEM \implies igWlsAllDisj\ (asIMOD\ SEM)$   
*<proof>*

Only "bound arit" abstraction domains are inhabited:

**lemma** *asIMOD-igWlsAbsIsInBar*:  
 $sWlsNE\ SEM \implies igWlsAbsIsInBar\ (asIMOD\ SEM)$   
*<proof>*

Domain preservation by the operators

The constructs preserve the domains:

**lemma** *asIMOD-igVarIPresIGWls*:  $igVarIPresIGWls\ (asIMOD\ SEM)$   
*<proof>*

**lemma** *asIMOD-igAbsIPresIGWls*:  
 $sWlsDisj\ SEM \implies igAbsIPresIGWls\ (asIMOD\ SEM)$   
*<proof>*

**lemma** *asIMOD-igOpIPresIGWls*:  
 $sOpPrSWls\ SEM \implies sWlsNE\ SEM \implies igOpIPresIGWls\ (asIMOD\ SEM)$   
*<proof>*

**lemma** *asIMOD-igConsIPresIGWls*:  
 $wlsSEM\ SEM \implies igConsIPresIGWls\ (asIMOD\ SEM)$   
*<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*  $\implies$  *igFreshIGAbs1* (*asIMOD SEM*)  
{*proof*}

**lemma** *asIMOD-igFreshIGAbs2*:  
*sWlsDisj SEM*  $\implies$  *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*:  $eqBut (val (x := sX)-xs) ((val (y := sX)-xs) \frown [y \wedge x]-xs) xs y$   
*<proof>*

**lemma** *asIMOD-igAbsCongS*:  $sWlsDisj SEM \implies igAbsCongS (asIMOD SEM)$   
*<proof>*

The abstraction-renaming clause holds:

**lemma** *asIMOD-igAbs3*:

**assumes** *sWlsDisj SEM* **and** *igWls (asIMOD SEM) s X*

**shows**

$igAbs (asIMOD SEM) xs y (igSubst (asIMOD SEM) xs (igVar (asIMOD SEM) xs y) x X) =$

$(\lambda val. \text{if } sWlsVal SEM val$   
 $\quad \text{then } sAbs xs (\lambda sX. \text{if } sWls SEM (asSort xs) sX$   
 $\quad \quad \text{then } (igSubst (asIMOD SEM) xs (igVar (asIMOD SEM)$   
 $xs y) x X) (val (y := sX)-xs)$   
 $\quad \quad \text{else } sDummy SEM s)$   
 $\quad \text{else undefined})$

*<proof>*

**lemma** *asIMOD-igAbsRen*:

$sWlsDisj SEM \implies igAbsRen (asIMOD SEM)$

*<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*:  $wlsSEM SEM \implies iwlsFSwSb (asIMOD SEM)$

*<proof>*

**lemma** *asIMOD-wlsFSbSw*:  $wlsSEM SEM \implies iwlsFSbSw (asIMOD SEM)$

*<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-sbst algebras would do the job, and, moreover, if we did not care about swapping, fresh-sbst algebras would do the job. Therefore, our exhaustive study of the model from previous section had a degree of redundancy w.r.t. to our main goal – 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 algebras to establish the required results. (Recall however that either algebraic route we take, the initial morphism turns out to be the same function.)

**definition** *semInt* **where**  $semInt\ SEM \equiv iter\ (asIMOD\ SEM)$

**definition** *semIntAbs* **where**  $semIntAbs\ SEM \equiv iterAbs\ (asIMOD\ SEM)$

**lemma** *semIntAll-termFSwSbImorph*:

$wlsSEM\ SEM \implies$

$termFSwSbImorph\ (semInt\ SEM)\ (semIntAbs\ SEM)\ (asIMOD\ SEM)$   
 $\langle proof \rangle$

**lemma** *semInt-prWls*:

$wlsSEM\ SEM \implies prWls\ (semInt\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semIntAbs-prWlsAbs*:

$wlsSEM\ SEM \implies prWlsAbs\ (semIntAbs\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semIntAll-prWlsAll*:

$wlsSEM\ SEM \implies prWlsAll\ (semInt\ SEM)\ (semIntAbs\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semInt-prVar*:

$wlsSEM\ SEM \implies prVar\ (semInt\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semIntAll-prAbs*:

**fixes**  $SEM :: ('index, 'bindex, 'varSort, 'sort, 'opSym, 'sTerm) semDom$

**assumes**  $wlsSEM\ SEM$

**shows**  $prAbs\ (semInt\ SEM)\ (semIntAbs\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semIntAll-prOp*:

**assumes**  $wlsSEM\ SEM$

**shows**  $prOp\ (semInt\ SEM)\ (semIntAbs\ SEM)\ SEM$   
 $\langle proof \rangle$

**lemma** *semIntAll-prCons*:

**assumes**  $wlsSEM\ SEM$

**shows**  $prCons (semInt SEM) (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semInt-prFresh$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prFresh (semInt SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAbs-prFreshAbs$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prFreshAbs (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAll-prFreshAll$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prFreshAll (semInt SEM) (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semInt-prSwap$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSwap (semInt SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAbs-prSwapAbs$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSwapAbs (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAll-prSwapAll$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSwapAll (semInt SEM) (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semInt-prSubst$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSubst (semInt SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAbs-prSubstAbs$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSubstAbs (semInt SEM) (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**lemma**  $semIntAll-prSubstAll$ :  
**assumes**  $wlsSEM SEM$   
**shows**  $prSubstAll (semInt SEM) (semIntAbs SEM) SEM$   
 $\langle proof \rangle$

**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  $\Rightarrow$  'gTerm  $\Rightarrow$  bool
  gWlsAbs :: 'varSort  $\times$  'sort  $\Rightarrow$  'gAbs  $\Rightarrow$  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**

$gFreshInp \text{ MOD } ys \ y \ inp' \ inp \equiv liftAll2 \ (gFresh \text{ MOD } ys \ y) \ inp' \ inp$

**definition**  $gFreshBinp$  **where**

$gFreshBinp \text{ MOD } ys \ y \ binp' \ binp \equiv liftAll2 \ (gFreshAbs \text{ MOD } ys \ y) \ binp' \ binp$

**definition** *gSwapInp* **where**

*gSwapInp* *MOD* *zs z1 z2 inp' inp*  $\equiv$  *lift2* (*gSwap* *MOD* *zs z1 z2*) *inp' inp*

**definition** *gSwapBinp* **where**

*gSwapBinp* *MOD* *zs z1 z2 binp' binp*  $\equiv$  *lift2* (*gSwapAbs* *MOD* *zs z1 z2*) *binp' binp*

**definition** *gSubstInp* **where**

*gSubstInp* *MOD* *ys Y' Y y inp' inp*  $\equiv$  *lift2* (*gSubst* *MOD* *ys Y' Y y*) *inp' inp*

**definition** *gSubstBinp* **where**

*gSubstBinp* *MOD* *ys Y' Y y binp' binp*  $\equiv$  *lift2* (*gSubstAbs* *MOD* *ys Y' Y y*) *binp' binp*

**context** *FixSyn*

**begin**

**definition** *gWlsInp* **where**

*gWlsInp* *MOD* *delta inp*  $\equiv$   
*wlsOpS* *delta*  $\wedge$  *sameDom* (*arOf* *delta*) *inp*  $\wedge$  *liftAll2* (*gWls* *MOD*) (*arOf* *delta*)  
*inp*

**lemmas** *gWlsInp-defs* = *gWlsInp-def* *sameDom-def* *liftAll2-def*

**definition** *gWlsBinp* **where**

*gWlsBinp* *MOD* *delta binp*  $\equiv$   
*wlsOpS* *delta*  $\wedge$  *sameDom* (*barOf* *delta*) *binp*  $\wedge$  *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*  $\equiv \forall s s' X. gWls MOD s X \wedge gWls MOD s' X \longrightarrow s = s'$

**definition** *gWlsAbsDisj* **where**  
*gWlsAbsDisj MOD*  $\equiv \forall xs s xs' s' A.$   
 $isInBar (xs,s) \wedge isInBar (xs',s') \wedge$   
 $gWlsAbs MOD (xs,s) A \wedge gWlsAbs MOD (xs',s') A$   
 $\longrightarrow xs = xs' \wedge s = s'$

**definition** *gWlsAllDisj* **where**  
*gWlsAllDisj MOD*  $\equiv gWlsDisj MOD \wedge gWlsAbsDisj MOD$

**lemmas** *gWlsAllDisj-defs =*  
*gWlsAllDisj-def gWlsDisj-def gWlsAbsDisj-def*

Abstraction domains inhabited only within bound arities:

**definition** *gWlsAbsIsInBar* **where**  
*gWlsAbsIsInBar MOD*  $\equiv \forall us s A. gWlsAbs MOD (us,s) A \longrightarrow isInBar (us,s)$

Domain preservation by the operators

The constructs preserve the domains:

**definition** *gVarPresGWls* **where**  
*gVarPresGWls MOD*  $\equiv \forall xs x. gWls MOD (asSort xs) (gVar MOD xs x)$

**definition** *gAbsPresGWls* **where**  
*gAbsPresGWls MOD*  $\equiv \forall xs s x X' X.$   
 $isInBar (xs,s) \wedge wls s X' \wedge gWls MOD s X \longrightarrow$   
 $gWlsAbs MOD (xs,s) (gAbs MOD xs x X' X)$

**definition** *gOpPresGWls* **where**  
*gOpPresGWls MOD*  $\equiv \forall delta inp' inp binp' binp.$   
 $wlsInp delta inp' \wedge gWlsInp MOD delta inp \wedge wlsBinp delta binp' \wedge gWlsBinp$   
 $MOD delta binp$   
 $\longrightarrow gWls MOD (stOf delta) (gOp MOD delta inp' inp binp' binp)$

**definition** *gConsPresGWls* **where**

$gConsPresGWls\ MOD \equiv gVarPresGWls\ MOD \wedge gAbsPresGWls\ MOD \wedge gOpPresGWls\ MOD$

**lemmas**  $gConsPresGWls-defs = gConsPresGWls-def$   
 $gVarPresGWls-def\ gAbsPresGWls-def\ gOpPresGWls-def$

“swap” preserves the domains:

**definition**  $gSwapPresGWls$  **where**  
 $gSwapPresGWls\ MOD \equiv \forall\ zs\ z1\ z2\ s\ X'\ X.$   
 $wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$   
 $gWls\ MOD\ s\ (gSwap\ MOD\ zs\ z1\ z2\ X'\ X)$

**definition**  $gSwapAbsPresGWlsAbs$  **where**  
 $gSwapAbsPresGWlsAbs\ MOD \equiv \forall\ zs\ z1\ z2\ us\ s\ A'\ A.$   
 $isInBar\ (us,s) \wedge wlsAbs\ (us,s)\ A' \wedge gWlsAbs\ MOD\ (us,s)\ A \longrightarrow$   
 $gWlsAbs\ MOD\ (us,s)\ (gSwapAbs\ MOD\ zs\ z1\ z2\ A'\ A)$

**definition**  $gSwapAllPresGWlsAll$  **where**  
 $gSwapAllPresGWlsAll\ MOD \equiv gSwapPresGWls\ MOD \wedge gSwapAbsPresGWlsAbs\ MOD$

**lemmas**  $gSwapAllPresGWlsAll-defs =$   
 $gSwapAllPresGWlsAll-def\ gSwapPresGWls-def\ gSwapAbsPresGWlsAbs-def$

“subst” preserves the domains:

**definition**  $gSubstPresGWls$  **where**  
 $gSubstPresGWls\ MOD \equiv \forall\ ys\ Y'\ Y\ y\ s\ X'\ X.$   
 $wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \wedge wls\ s\ X' \wedge gWls\ MOD\ s\ X$   
 $\longrightarrow$   
 $gWls\ MOD\ s\ (gSubst\ MOD\ ys\ Y'\ Y\ y\ X'\ X)$

**definition**  $gSubstAbsPresGWlsAbs$  **where**  
 $gSubstAbsPresGWlsAbs\ MOD \equiv \forall\ ys\ Y'\ Y\ y\ us\ s\ A'\ A.$   
 $isInBar\ (us,s) \wedge$   
 $wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \wedge wlsAbs\ (us,s)\ A' \wedge gWlsAbs$   
 $MOD\ (us,s)\ A \longrightarrow$   
 $gWlsAbs\ MOD\ (us,s)\ (gSubstAbs\ MOD\ ys\ Y'\ Y\ y\ A'\ A)$

**definition**  $gSubstAllPresGWlsAll$  **where**  
 $gSubstAllPresGWlsAll\ MOD \equiv gSubstPresGWls\ MOD \wedge gSubstAbsPresGWlsAbs\ MOD$

**lemmas**  $gSubstAllPresGWlsAll-defs =$   
 $gSubstAllPresGWlsAll-def\ gSubstPresGWls-def\ gSubstAbsPresGWlsAbs-def$

Clauses for fresh:

**definition**  $gFreshGVar$  **where**  
 $gFreshGVar\ MOD \equiv \forall\ ys\ y\ xs\ x.$   
 $(ys \neq xs \vee y \neq x) \longrightarrow$



$gFresh\ MOD\ ys\ y\ (Var\ xs\ x)\ (gVar\ MOD\ xs\ x)$

**definition**  $gFreshGAbs1$  **where**

$gFreshGAbs1\ MOD \equiv \forall\ ys\ y\ s\ X'\ X.$   
 $isInBar\ (ys,s) \wedge wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$   
 $gFreshAbs\ MOD\ ys\ y\ (Abs\ ys\ y\ X')\ (gAbs\ MOD\ ys\ y\ X'\ X)$

**definition**  $gFreshGAbs2$  **where**

$gFreshGAbs2\ MOD \equiv \forall\ ys\ y\ xs\ x\ s\ X'\ X.$   
 $isInBar\ (xs,s) \wedge wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$   
 $fresh\ ys\ y\ X' \wedge gFresh\ MOD\ ys\ y\ X'\ X \longrightarrow$   
 $gFreshAbs\ MOD\ ys\ y\ (Abs\ xs\ x\ X')\ (gAbs\ MOD\ xs\ x\ X'\ X)$

**definition**  $gFreshGOp$  **where**

$gFreshGOp\ MOD \equiv \forall\ ys\ y\ delta\ inp'\ inp\ binp'\ binp.$   
 $wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \wedge wlsBinp\ delta\ binp' \wedge gWlsBinp$   
 $MOD\ delta\ binp \longrightarrow$   
 $freshInp\ ys\ y\ inp' \wedge gFreshInp\ MOD\ ys\ y\ inp'\ inp \wedge$   
 $freshBinp\ ys\ y\ binp' \wedge gFreshBinp\ MOD\ ys\ y\ binp'\ binp \longrightarrow$   
 $gFresh\ MOD\ ys\ y\ (Op\ delta\ inp'\ binp')\ (gOp\ MOD\ delta\ inp'\ inp\ binp'\ binp)$

**definition**  $gFreshCls$  **where**

$gFreshCls\ MOD \equiv gFreshGVar\ MOD \wedge gFreshGAbs1\ MOD \wedge gFreshGAbs2\ MOD$   
 $\wedge gFreshGOp\ MOD$

**lemmas**  $gFreshCls-defs = gFreshCls-def$

$gFreshGVar-def\ gFreshGAbs1-def\ gFreshGAbs2-def\ gFreshGOp-def$

**definition**  $gSwapGVar$  **where**

$gSwapGVar\ MOD \equiv \forall\ zs\ z1\ z2\ xs\ x.$   
 $gSwap\ MOD\ zs\ z1\ z2\ (Var\ xs\ x)\ (gVar\ MOD\ xs\ x) =$   
 $gVar\ MOD\ xs\ (x\ @xs[z1 \wedge z2]-zs)$

**definition**  $gSwapGAbs$  **where**

$gSwapGAbs\ MOD \equiv \forall\ zs\ z1\ z2\ xs\ x\ s\ X'\ X.$   
 $isInBar\ (xs,s) \wedge wls\ s\ X' \wedge gWls\ MOD\ s\ X \longrightarrow$   
 $gSwapAbs\ MOD\ zs\ z1\ z2\ (Abs\ xs\ x\ X')\ (gAbs\ MOD\ xs\ x\ X'\ X) =$   
 $gAbs\ MOD\ xs\ (x\ @xs[z1 \wedge z2]-zs)\ (X'\ #[z1 \wedge z2]-zs)\ (gSwap\ MOD\ zs\ z1\ z2\ X'$   
 $X)$

**definition**  $gSwapGOp$  **where**

$gSwapGOp\ MOD \equiv \forall\ zs\ z1\ z2\ delta\ inp'\ inp\ binp'\ binp.$   
 $wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \wedge wlsBinp\ delta\ binp' \wedge gWlsBinp$   
 $MOD\ delta\ binp \longrightarrow$   
 $gSwap\ MOD\ zs\ z1\ z2\ (Op\ delta\ inp'\ binp')\ (gOp\ MOD\ delta\ inp'\ inp\ binp'\ binp)$   
 $=$   
 $gOp\ MOD\ delta$

$$(inp' \%[z1 \wedge z2]-zs) (gSwapInp MOD zs z1 z2 inp' inp)$$

$$(binp' \%[z1 \wedge z2]-zs) (gSwapBinp MOD zs z1 z2 binp' binp)$$

**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**

$$gSubstGAbs MOD \equiv \forall ys y Y' Y xs x s X' X.$$

$$isInBar (xs,s) \wedge$$

$$wls (asSort ys) Y' \wedge gWls MOD (asSort ys) Y \wedge$$

$$wls s X' \wedge gWls MOD s X \longrightarrow$$

$$(xs \neq ys \vee x \neq y) \wedge fresh xs x Y' \wedge gFresh MOD xs x Y' Y \longrightarrow$$

$$gSubstAbs MOD ys Y' Y y (Abs xs x X') (gAbs MOD xs x X' X) =$$

$$gAbs MOD xs x (X' \#[Y' / y]-ys) (gSubst MOD ys Y' Y y X' X)$$

**definition** *gSubstGOp* **where**

$$gSubstGOp MOD \equiv \forall ys y Y' Y delta inp' inp binp' binp.$$

$$wls (asSort ys) Y' \wedge gWls MOD (asSort ys) Y \wedge$$

$$wlsInp delta inp' \wedge gWlsInp MOD delta inp \wedge$$

$$wlsBinp delta binp' \wedge gWlsBinp MOD delta binp \longrightarrow$$

$$gSubst MOD ys Y' Y y (Op delta inp' binp') (gOp MOD delta inp' inp binp' binp) =$$

$$gOp MOD delta$$

$$(inp' \%[Y' / y]-ys) (gSubstInp MOD ys Y' Y y inp' inp)$$

$$(binp' \%[Y' / y]-ys) (gSubstBinp MOD ys Y' Y y binp' binp)$$

**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**

$$\begin{aligned}
gAbsCongS \text{ MOD} &\equiv \forall \ xs \ x \ x2 \ y \ s \ X' \ X \ X2' \ X2. \\
&\quad isInBar \ (xs,s) \wedge \\
&\quad wls \ s \ X' \wedge gWls \ \text{MOD} \ s \ X \wedge \\
&\quad wls \ s \ X2' \wedge gWls \ \text{MOD} \ s \ X2 \longrightarrow \\
&\quad fresh \ xs \ y \ X' \wedge gFresh \ \text{MOD} \ xs \ y \ X' \ X \wedge \\
&\quad fresh \ xs \ y \ X2' \wedge gFresh \ \text{MOD} \ xs \ y \ X2' \ X2 \wedge \\
&\quad (X' \ #[y \wedge x]-xs) = (X2' \ #[y \wedge x2]-xs) \longrightarrow \\
&\quad gSwap \ \text{MOD} \ xs \ y \ x \ X' \ X = gSwap \ \text{MOD} \ xs \ y \ x2 \ X2' \ X2 \longrightarrow \\
&\quad gAbs \ \text{MOD} \ xs \ x \ X' \ X = gAbs \ \text{MOD} \ xs \ x2 \ X2' \ X2
\end{aligned}$$

**definition**  $gAbsRen$  **where**

$$\begin{aligned}
gAbsRen \ \text{MOD} &\equiv \forall \ xs \ y \ x \ s \ X' \ X. \\
&\quad isInBar \ (xs,s) \wedge wls \ s \ X' \wedge gWls \ \text{MOD} \ s \ X \longrightarrow \\
&\quad fresh \ xs \ y \ X' \wedge gFresh \ \text{MOD} \ xs \ y \ X' \ X \longrightarrow \\
&\quad gAbs \ \text{MOD} \ xs \ y \ (X' \ #[y \ / \ / \ x]-xs) \ (gSubst \ \text{MOD} \ xs \ (Var \ xs \ y) \ (gVar \ \text{MOD} \ xs \\
&\quad y) \ x \ X' \ X) = \\
&\quad gAbs \ \text{MOD} \ xs \ x \ X' \ X
\end{aligned}$$

Well-sorted fresh-swap models:

**definition**  $wlsFSw$  **where**

$$\begin{aligned}
wlsFSw \ \text{MOD} &\equiv gWlsAllDisj \ \text{MOD} \wedge gWlsAbsIsInBar \ \text{MOD} \wedge \\
&\quad gConsPresGWls \ \text{MOD} \wedge gSwapAllPresGWlsAll \ \text{MOD} \wedge \\
&\quad gFreshCls \ \text{MOD} \wedge gSwapCls \ \text{MOD} \wedge gAbsCongS \ \text{MOD}
\end{aligned}$$

**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**

$$\begin{aligned}
wlsFSb \ \text{MOD} &\equiv gWlsAllDisj \ \text{MOD} \wedge gWlsAbsIsInBar \ \text{MOD} \wedge \\
&\quad gConsPresGWls \ \text{MOD} \wedge gSubstAllPresGWlsAll \ \text{MOD} \wedge \\
&\quad gFreshCls \ \text{MOD} \wedge gSubstCls \ \text{MOD} \wedge gAbsRen \ \text{MOD}
\end{aligned}$$

**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**  
 $wlsFSwSb\ MOD \equiv wlsFSw\ MOD \wedge gSubstAllPresGWlsAll\ MOD \wedge gSubstCls\ MOD$

**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**  
 $wlsFSbSw\ MOD \equiv wlsFSb\ MOD \wedge gSwapAllPresGWlsAll\ MOD \wedge gSwapCls\ MOD$

**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**  
 $gSwapInpPresGWlsInp\ MOD \equiv \forall\ zs\ z1\ z2\ delta\ inp'\ inp.$   
 $wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \longrightarrow$   
 $gWlsInp\ MOD\ delta\ (gSwapInp\ MOD\ zs\ z1\ z2\ inp'\ inp)$

**definition**  $gSubstInpPresGWlsInp$  **where**  
 $gSubstInpPresGWlsInp\ MOD \equiv \forall\ ys\ y\ Y'\ Y\ delta\ inp'\ inp.$   
 $wls\ (asSort\ ys)\ Y' \wedge gWls\ MOD\ (asSort\ ys)\ Y \wedge$   
 $wlsInp\ delta\ inp' \wedge gWlsInp\ MOD\ delta\ inp \longrightarrow$   
 $gWlsInp\ MOD\ delta\ (gSubstInp\ MOD\ ys\ Y'\ Y\ y\ inp'\ inp)$

**lemma**  $imp-gSwapInpPresGWlsInp$ :  
 $gSwapPresGWls\ MOD \implies gSwapInpPresGWlsInp\ MOD$   
 $\langle proof \rangle$

**lemma** *imp-gSubstInpPresGWlsInp*:  
 $gSubstPresGWls \text{ MOD} \implies gSubstInpPresGWlsInp \text{ MOD}$   
 ⟨proof⟩

Then for bound inputs:

**definition** *gSwapBinpPresGWlsBinp* **where**  
 $gSwapBinpPresGWlsBinp \text{ MOD} \equiv \forall zs \ z1 \ z2 \ \text{delta} \ \text{binp}' \ \text{binp}.$   
 $wlsBinp \ \text{delta} \ \text{binp}' \wedge gWlsBinp \ \text{MOD} \ \text{delta} \ \text{binp} \longrightarrow$   
 $gWlsBinp \ \text{MOD} \ \text{delta} \ (gSwapBinp \ \text{MOD} \ zs \ z1 \ z2 \ \text{binp}' \ \text{binp})$

**definition** *gSubstBinpPresGWlsBinp* **where**  
 $gSubstBinpPresGWlsBinp \ \text{MOD} \equiv \forall ys \ y \ Y' \ Y \ \text{delta} \ \text{binp}' \ \text{binp}.$   
 $wls \ (\text{asSort} \ ys) \ Y' \wedge gWls \ \text{MOD} \ (\text{asSort} \ ys) \ Y \wedge$   
 $wlsBinp \ \text{delta} \ \text{binp}' \wedge gWlsBinp \ \text{MOD} \ \text{delta} \ \text{binp} \longrightarrow$   
 $gWlsBinp \ \text{MOD} \ \text{delta} \ (gSubstBinp \ \text{MOD} \ ys \ Y' \ Y \ \text{binp}' \ \text{binp})$

**lemma** *imp-gSwapBinpPresGWlsBinp*:  
 $gSwapAbsPresGWlsAbs \ \text{MOD} \implies gSwapBinpPresGWlsBinp \ \text{MOD}$   
 ⟨proof⟩

**lemma** *imp-gSubstBinpPresGWlsBinp*:  
 $gSubstAbsPresGWlsAbs \ \text{MOD} \implies gSubstBinpPresGWlsBinp \ \text{MOD}$   
 ⟨proof⟩

### 10.3 Model morphisms from the term model

**definition** *presWls* **where**  
 $presWls \ h \ \text{MOD} \equiv \forall s \ X. \ wls \ s \ X \longrightarrow gWls \ \text{MOD} \ s \ (h \ X)$

**definition** *presWlsAbs* **where**  
 $presWlsAbs \ hA \ \text{MOD} \equiv \forall us \ s \ A. \ wlsAbs \ (us, s) \ A \longrightarrow gWlsAbs \ \text{MOD} \ (us, s) \ (hA \ A)$

**definition** *presWlsAll* **where**  
 $presWlsAll \ h \ hA \ \text{MOD} \equiv presWls \ h \ \text{MOD} \wedge presWlsAbs \ hA \ \text{MOD}$

**lemmas** *presWlsAll-defs* = *presWlsAll-def presWls-def presWlsAbs-def*

**definition** *presVar* **where**  
 $presVar \ h \ \text{MOD} \equiv \forall xs \ x. \ h \ (\text{Var} \ xs \ x) = gVar \ \text{MOD} \ xs \ x$

**definition** *presAbs* **where**  
 $presAbs \ h \ hA \ \text{MOD} \equiv \forall xs \ x \ s \ X.$   
 $isInBar \ (xs, s) \wedge wls \ s \ X \longrightarrow$   
 $hA \ (\text{Abs} \ xs \ x \ X) = gAbs \ \text{MOD} \ xs \ x \ X \ (h \ X)$

**definition** *presOp* **where**  
 $presOp \ h \ hA \ \text{MOD} \equiv \forall \ \text{delta} \ \text{inp} \ \text{binp}.$   
 $wlsInp \ \text{delta} \ \text{inp} \wedge wlsBinp \ \text{delta} \ \text{binp} \longrightarrow$

$$h (Op \text{ delta } inp \text{ binp}) = \\ gOp \text{ MOD } \text{ delta } inp (lift \ h \ inp) \text{ binp } (lift \ hA \ binp)$$

**definition** *presCons* **where**

$$presCons \ h \ hA \ MOD \equiv presVar \ h \ MOD \wedge presAbs \ h \ hA \ MOD \wedge presOp \ h \ hA \ MOD$$

**lemmas** *presCons-defs* = *presCons-def*  
*presVar-def presAbs-def presOp-def*

**definition** *presFresh* **where**

$$presFresh \ h \ MOD \equiv \forall \ ys \ y \ s \ X. \\ wls \ s \ X \longrightarrow \\ fresh \ ys \ y \ X \longrightarrow gFresh \ MOD \ ys \ y \ X \ (h \ X)$$

**definition** *presFreshAbs* **where**

$$presFreshAbs \ hA \ MOD \equiv \forall \ ys \ y \ us \ s \ A. \\ wlsAbs \ (us, s) \ A \longrightarrow \\ freshAbs \ ys \ y \ A \longrightarrow gFreshAbs \ MOD \ ys \ y \ A \ (hA \ A)$$

**definition** *presFreshAll* **where**

$$presFreshAll \ h \ hA \ MOD \equiv presFresh \ h \ MOD \wedge presFreshAbs \ hA \ MOD$$

**lemmas** *presFreshAll-defs* = *presFreshAll-def*  
*presFresh-def presFreshAbs-def*

**definition** *presSwap* **where**

$$presSwap \ h \ MOD \equiv \forall \ zs \ z1 \ z2 \ s \ X. \\ wls \ s \ X \longrightarrow \\ h \ (X \ #[z1 \wedge z2]-zs) = gSwap \ MOD \ zs \ z1 \ z2 \ X \ (h \ X)$$

**definition** *presSwapAbs* **where**

$$presSwapAbs \ hA \ MOD \equiv \forall \ zs \ z1 \ z2 \ us \ s \ A. \\ wlsAbs \ (us, s) \ A \longrightarrow \\ hA \ (A \ \$[z1 \wedge z2]-zs) = gSwapAbs \ MOD \ zs \ z1 \ z2 \ A \ (hA \ A)$$

**definition** *presSwapAll* **where**

$$presSwapAll \ h \ hA \ MOD \equiv presSwap \ h \ MOD \wedge presSwapAbs \ hA \ MOD$$

**lemmas** *presSwapAll-defs* = *presSwapAll-def*  
*presSwap-def presSwapAbs-def*

**definition** *presSubst* **where**

$$presSubst \ h \ MOD \equiv \forall \ ys \ Y \ y \ s \ X. \\ wls \ (asSort \ ys) \ Y \wedge wls \ s \ X \longrightarrow \\ h \ (subst \ ys \ Y \ y \ X) = gSubst \ MOD \ ys \ Y \ (h \ Y) \ y \ X \ (h \ X)$$

**definition** *presSubstAbs* **where**

$$presSubstAbs \ h \ hA \ MOD \equiv \forall \ ys \ Y \ y \ us \ s \ A.$$

$$\begin{aligned} & wls (asSort ys) Y \wedge wlsAbs (us,s) A \longrightarrow \\ & hA (A \$[Y / y]-ys) = gSubstAbs MOD ys Y (h Y) y A (hA A) \end{aligned}$$

**definition** *presSubstAll* **where**

$$presSubstAll h hA MOD \equiv presSubst h MOD \wedge presSubstAbs h hA MOD$$

**lemmas** *presSubstAll-defs* = *presSubstAll-def*

*presSubst-def presSubstAbs-def*

**definition** *termFSwMorph* **where**

$$\begin{aligned} termFSwMorph h hA MOD & \equiv presWlsAll h hA MOD \wedge presCons h hA MOD \wedge \\ & presFreshAll h hA MOD \wedge presSwapAll h hA MOD \end{aligned}$$

**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**

$$\begin{aligned} termFSbMorph h hA MOD & \equiv presWlsAll h hA MOD \wedge presCons h hA MOD \wedge \\ & presFreshAll h hA MOD \wedge presSubstAll h hA MOD \end{aligned}$$

**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**

$$termFSwSbMorph h hA MOD \equiv termFSwMorph h hA MOD \wedge presSubstAll h hA MOD$$

**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*:

$$wlsInp delta inp \Longrightarrow presWls h MOD \Longrightarrow gWlsInp MOD delta (lift h inp)$$

*\langle proof \rangle*

. for bound inputs:

**lemma** *presWls-wlsBinp*:

$$wlsBinp delta binp \Longrightarrow presWlsAbs hA MOD \Longrightarrow gWlsBinp MOD delta (lift hA binp)$$

*<proof>*

## 10.4 From models to iterative models

The transition map:

**definition** *fromMOD* ::

$(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'var}, \text{'gTerm}, \text{'gAbs}) \text{ model}$

$\Rightarrow$

$(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'sort}, \text{'opSym}, \text{'var},$   
 $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'var}, \text{'opSym}) \text{term} \times \text{'gTerm},$   
 $(\text{'index}, \text{'bindex}, \text{'varSort}, \text{'var}, \text{'opSym}) \text{abs} \times \text{'gAbs}) \text{ Iteration.model}$

**where**

*fromMOD MOD*  $\equiv$

$\langle$

$igWls = \lambda s X'X. wls\ s\ (fst\ X'X) \wedge gWls\ MOD\ s\ (snd\ X'X),$

$igWlsAbs = \lambda us-s\ A'A. wlsAbs\ us-s\ (fst\ A'A) \wedge gWlsAbs\ MOD\ us-s\ (snd\ A'A),$

$igVar = \lambda xs\ x. (Var\ xs\ x, gVar\ MOD\ xs\ x),$

$igAbs = \lambda xs\ x\ X'X. (Abs\ xs\ x\ (fst\ X'X), gAbs\ MOD\ xs\ x\ (fst\ X'X)\ (snd\ X'X)),$

$igOp =$

$\lambda delta\ iinp\ biinp.$

$(Op\ delta\ (lift\ fst\ iinp)\ (lift\ fst\ biinp),$

$gOp\ MOD\ delta$

$(lift\ fst\ iinp)\ (lift\ snd\ iinp)$

$(lift\ fst\ biinp)\ (lift\ snd\ biinp)),$

$igFresh =$

$\lambda ys\ y\ X'X. fresh\ ys\ y\ (fst\ X'X) \wedge gFresh\ MOD\ ys\ y\ (fst\ X'X)\ (snd\ X'X),$

$igFreshAbs =$

$\lambda ys\ y\ A'A. freshAbs\ ys\ y\ (fst\ A'A) \wedge gFreshAbs\ MOD\ ys\ y\ (fst\ A'A)\ (snd\ A'A),$

$igSwap =$

$\lambda zs\ z1\ z2\ X'X. ((fst\ X'X)\ \#[z1\ \wedge\ z2]-zs, gSwap\ MOD\ zs\ z1\ z2\ (fst\ X'X)\ (snd\ X'X)),$

$igSwapAbs =$

$\lambda zs\ z1\ z2\ A'A. ((fst\ A'A)\ \$[z1\ \wedge\ z2]-zs, gSwapAbs\ MOD\ zs\ z1\ z2\ (fst\ A'A)\ (snd\ A'A)),$

$igSubst =$

$\lambda ys\ Y'Y\ y\ X'X.$

$((fst\ X'X)\ \#[(fst\ Y'Y)\ / y]-ys,$

$gSubst\ MOD\ ys\ (fst\ Y'Y)\ (snd\ Y'Y)\ y\ (fst\ X'X)\ (snd\ X'X)),$

$igSubstAbs =$

$\lambda ys\ Y'Y\ y\ A'A.$

$((fst\ A'A)\ \$[(fst\ Y'Y)\ / y]-ys,$

$gSubstAbs\ MOD\ ys\ (fst\ Y'Y)\ (snd\ Y'Y)\ y\ (fst\ A'A)\ (snd\ A'A))$

$\rangle$

Basic simplification rules:



**lemma** *fromMOD-basic-simps[simp]*:

$$\text{igWls } (\text{fromMOD } \text{MOD}) \text{ } s \text{ } X'X = \\ (\text{wls } s \text{ } (\text{fst } X'X) \wedge \text{gWls } \text{MOD } s \text{ } (\text{snd } X'X))$$

$$\text{igWlsAbs } (\text{fromMOD } \text{MOD}) \text{ } us\text{-}s \text{ } A'A = \\ (\text{wlsAbs } us\text{-}s \text{ } (\text{fst } A'A) \wedge \text{gWlsAbs } \text{MOD } us\text{-}s \text{ } (\text{snd } A'A))$$

$$\text{igVar } (\text{fromMOD } \text{MOD}) \text{ } xs \text{ } x = (\text{Var } xs \text{ } x, \text{gVar } \text{MOD } xs \text{ } x)$$

$$\text{igAbs } (\text{fromMOD } \text{MOD}) \text{ } xs \text{ } x \text{ } X'X = (\text{Abs } xs \text{ } x \text{ } (\text{fst } X'X), \text{gAbs } \text{MOD } xs \text{ } x \text{ } (\text{fst } X'X) \text{ } (\text{snd } X'X))$$

$$\text{igOp } (\text{fromMOD } \text{MOD}) \text{ } \text{delta } \text{iinp } \text{biinp} = \\ (\text{Op } \text{delta } (\text{lift } \text{fst } \text{iinp}) (\text{lift } \text{fst } \text{biinp}), \\ \text{gOp } \text{MOD } \text{delta} \\ (\text{lift } \text{fst } \text{iinp}) (\text{lift } \text{snd } \text{iinp}) \\ (\text{lift } \text{fst } \text{biinp}) (\text{lift } \text{snd } \text{biinp}))$$

$$\text{igFresh } (\text{fromMOD } \text{MOD}) \text{ } ys \text{ } y \text{ } X'X = \\ (\text{fresh } ys \text{ } y \text{ } (\text{fst } X'X) \wedge \text{gFresh } \text{MOD } ys \text{ } y \text{ } (\text{fst } X'X) \text{ } (\text{snd } X'X))$$

$$\text{igFreshAbs } (\text{fromMOD } \text{MOD}) \text{ } ys \text{ } y \text{ } A'A = \\ (\text{freshAbs } ys \text{ } y \text{ } (\text{fst } A'A) \wedge \text{gFreshAbs } \text{MOD } ys \text{ } y \text{ } (\text{fst } A'A) \text{ } (\text{snd } A'A))$$

$$\text{igSwap } (\text{fromMOD } \text{MOD}) \text{ } zs \text{ } z1 \text{ } z2 \text{ } X'X = \\ ((\text{fst } X'X) \#[z1 \wedge z2]\text{-}zs, \text{gSwap } \text{MOD } zs \text{ } z1 \text{ } z2 \text{ } (\text{fst } X'X) \text{ } (\text{snd } X'X))$$

$$\text{igSwapAbs } (\text{fromMOD } \text{MOD}) \text{ } zs \text{ } z1 \text{ } z2 \text{ } A'A = \\ ((\text{fst } A'A) \#[z1 \wedge z2]\text{-}zs, \text{gSwapAbs } \text{MOD } zs \text{ } z1 \text{ } z2 \text{ } (\text{fst } A'A) \text{ } (\text{snd } A'A))$$

$$\text{igSubst } (\text{fromMOD } \text{MOD}) \text{ } ys \text{ } Y'Y \text{ } y \text{ } X'X = \\ ((\text{fst } X'X) \#[(\text{fst } Y'Y) / y]\text{-}ys, \\ \text{gSubst } \text{MOD } ys \text{ } (\text{fst } Y'Y) \text{ } (\text{snd } Y'Y) \text{ } y \text{ } (\text{fst } X'X) \text{ } (\text{snd } X'X))$$

$$\text{igSubstAbs } (\text{fromMOD } \text{MOD}) \text{ } ys \text{ } Y'Y \text{ } y \text{ } A'A = \\ ((\text{fst } A'A) \#[(\text{fst } Y'Y) / y]\text{-}ys, \\ \text{gSubstAbs } \text{MOD } ys \text{ } (\text{fst } Y'Y) \text{ } (\text{snd } Y'Y) \text{ } y \text{ } (\text{fst } A'A) \text{ } (\text{snd } A'A)) \\ \langle \text{proof} \rangle$$

Simps for inputs

. for free inputs:

**lemma** *igWlsInp-fromMOD[simp]*:

$$\text{igWlsInp } (\text{fromMOD } \text{MOD}) \text{ } \text{delta } \text{iinp} \longleftrightarrow \\ \text{wlsInp } \text{delta } (\text{lift } \text{fst } \text{iinp}) \wedge \text{gWlsInp } \text{MOD } \text{delta } (\text{lift } \text{snd } \text{iinp}) \\ \langle \text{proof} \rangle$$

**lemma** *igFreshInp-fromMOD[simp]*:

$$\text{igFreshInp } (\text{fromMOD } \text{MOD}) \text{ } ys \text{ } y \text{ } \text{iinp} \longleftrightarrow \\ \text{freshInp } ys \text{ } y \text{ } (\text{lift } \text{fst } \text{iinp}) \wedge \text{gFreshInp } \text{MOD } ys \text{ } y \text{ } (\text{lift } \text{fst } \text{iinp}) \text{ } (\text{lift } \text{snd } \text{iinp})$$

*<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  $\longleftrightarrow$*   
*(wlsBinp delta (lift fst biinp)  $\wedge$  gWlsBinp MOD delta (lift snd biinp))*

*<proof>*

**lemma** *igFreshBinp-fromMOD[simp]*:

*igFreshBinp (fromMOD MOD) ys y biinp  $\longleftrightarrow$*   
*(freshBinp ys y (lift fst biinp)  $\wedge$*   
*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]*:  
 $gWlsDisj\ MOD \implies igWlsDisj\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igWlsAbsDisj-fromMOD[simp]*:  
 $gWlsAbsDisj\ MOD \implies igWlsAbsDisj\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igWlsAllDisj-fromMOD[simp]*:  
 $gWlsAllDisj\ MOD \implies igWlsAllDisj\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemmas** *igWlsAllDisj-fromMOD-simps =*  
 $igWlsDisj-fromMOD\ igWlsAbsDisj-fromMOD\ igWlsAllDisj-fromMOD$

Abstractions only within IsInBar:

**lemma** *igWlsAbsIsInBar-fromMOD[simp]*:  
 $gWlsAbsIsInBar\ MOD \implies igWlsAbsIsInBar\ (fromMOD\ MOD)$   
 ⟨proof⟩

The constructs preserve the domains:

**lemma** *igVarIPresIGWls-fromMOD[simp]*:  
 $gVarPresGWls\ MOD \implies igVarIPresIGWls\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igAbsIPresIGWls-fromMOD[simp]*:  
 $gAbsPresGWls\ MOD \implies igAbsIPresIGWls\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igOpIPresIGWls-fromMOD[simp]*:  
 $gOpPresGWls\ MOD \implies igOpIPresIGWls\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igConsIPresIGWls-fromMOD[simp]*:  
 $gConsPresGWls\ MOD \implies igConsIPresIGWls\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemmas** *igConsIPresIGWls-fromMOD-simps =*  
 $igVarIPresIGWls-fromMOD\ igAbsIPresIGWls-fromMOD$   
 $igOpIPresIGWls-fromMOD\ igConsIPresIGWls-fromMOD$

Swap preserves the domains:

**lemma** *igSwapIPresIGWls-fromMOD[simp]*:  
 $gSwapPresGWls\ MOD \implies igSwapIPresIGWls\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igSwapAbsIPresIGWlsAbs-fromMOD[simp]*:  
 $gSwapAbsPresGWlsAbs\ MOD \implies igSwapAbsIPresIGWlsAbs\ (fromMOD\ MOD)$   
 ⟨proof⟩

**lemma** *igSwapAllIPresIGWlsAll-fromMOD[simp]*:  
 $gSwapAllPresGWlsAll \text{ MOD} \implies igSwapAllIPresIGWlsAll \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemmas** *igSwapAllIPresIGWlsAll-fromMOD-simps =*  
*igSwapIPresIGWls-fromMOD igSwapAbsIPresIGWlsAbs-fromMOD igSwapAllIPresIG-*  
*WlsAll-fromMOD*

Subst preserves the domains:

**lemma** *igSubstIPresIGWls-fromMOD[simp]*:  
 $gSubstPresGWls \text{ MOD} \implies igSubstIPresIGWls \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igSubstAbsIPresIGWlsAbs-fromMOD[simp]*:  
 $gSubstAbsPresGWlsAbs \text{ MOD} \implies igSubstAbsIPresIGWlsAbs \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igSubstAllIPresIGWlsAll-fromMOD[simp]*:  
 $gSubstAllPresGWlsAll \text{ MOD} \implies igSubstAllIPresIGWlsAll \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemmas** *igSubstAllIPresIGWlsAll-fromMOD-simps =*  
*igSubstIPresIGWls-fromMOD igSubstAbsIPresIGWlsAbs-fromMOD igSubstAllIPresIG-*  
*WlsAll-fromMOD*

The fresh clauses:

**lemma** *igFreshIGVar-fromMOD[simp]*:  
 $gFreshGVar \text{ MOD} \implies igFreshIGVar \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igFreshIGAbs1-fromMOD[simp]*:  
 $gFreshGAbs1 \text{ MOD} \implies igFreshIGAbs1 \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igFreshIGAbs2-fromMOD[simp]*:  
 $gFreshGAbs2 \text{ MOD} \implies igFreshIGAbs2 \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igFreshIGOp-fromMOD[simp]*:  
 $gFreshGOp \text{ MOD} \implies igFreshIGOp \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemma** *igFreshCls-fromMOD[simp]*:  
 $gFreshCls \text{ MOD} \implies igFreshCls \text{ (fromMOD MOD)}$   
 ⟨proof⟩

**lemmas** *igFreshCls-fromMOD-simps =*  
*igFreshIGVar-fromMOD igFreshIGAbs1-fromMOD igFreshIGAbs2-fromMOD*

*igFreshIGOp-fromMOD igFreshCls-fromMOD*

The swap clauses

**lemma** *igSwapIGVar-fromMOD[simp]:*  
*gSwapGVar MOD  $\implies$  igSwapIGVar (fromMOD MOD)*  
(proof)

**lemma** *igSwapIGAbs-fromMOD[simp]:*  
*gSwapGAbs MOD  $\implies$  igSwapIGAbs (fromMOD MOD)*  
(proof)

**lemma** *igSwapIGOp-fromMOD[simp]:*  
*gSwapGOp MOD  $\implies$  igSwapIGOp (fromMOD MOD)*  
(proof)

**lemma** *igSwapCls-fromMOD[simp]:*  
*gSwapCls MOD  $\implies$  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  $\implies$  igSubstIGVar1 (fromMOD MOD)*  
(proof)

**lemma** *igSubstIGVar2-fromMOD[simp]:*  
*gSubstGVar2 MOD  $\implies$  igSubstIGVar2 (fromMOD MOD)*  
(proof)

**lemma** *igSubstIGAbs-fromMOD[simp]:*  
*gSubstGAbs MOD  $\implies$  igSubstIGAbs (fromMOD MOD)*  
(proof)

**lemma** *igSubstIGOp-fromMOD[simp]:*  
*gSubstGOp MOD  $\implies$  igSubstIGOp (fromMOD MOD)*  
(proof)

**lemma** *igSubstCls-fromMOD[simp]:*  
*gSubstCls MOD  $\implies$  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  $\implies$  igAbsRen (fromMOD MOD)*  
*<proof>*

Models:

**lemma** *iwlsFSw-fromMOD[simp]*:  
*wlsFSw MOD  $\implies$  iwlsFSw (fromMOD MOD)*  
*<proof>*

**lemma** *iwlsFSb-fromMOD[simp]*:  
*wlsFSb MOD  $\implies$  iwlsFSb (fromMOD MOD)*  
*<proof>*

**lemma** *iwlsFSwSb-fromMOD[simp]*:  
*wlsFSwSb MOD  $\implies$  iwlsFSwSb (fromMOD MOD)*  
*<proof>*

**lemma** *iwlsFSbSw-fromMOD[simp]*:  
*wlsFSbSw MOD  $\implies$  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**  $(wls\ s\ X \longrightarrow fst\ (h\ X) = X) \wedge (wlsAbs\ (us,s')\ A \longrightarrow fst\ (hA\ A) = A)$

*<proof>*

**lemma** *ipresCons-fromMOD-fst-simps[simp]*:

$\llbracket ipresCons\ h\ hA\ (fromMOD\ MOD); wls\ s\ X \rrbracket$

$\Longrightarrow\ fst\ (h\ X) = X$

$\llbracket ipresCons\ h\ hA\ (fromMOD\ MOD); wlsAbs\ (us,s')\ A \rrbracket$

$\Longrightarrow\ fst\ (hA\ A) = A$

*<proof>*

**lemma** *ipresCons-fromMOD-fst-inp[simp]*:

$ipresCons\ h\ hA\ (fromMOD\ MOD) \Longrightarrow wlsInp\ delta\ inp \Longrightarrow lift\ (fst\ o\ h)\ inp = inp$

*<proof>*

**lemma** *ipresCons-fromMOD-fst-binp[simp]*:

$ipresCons\ h\ hA\ (fromMOD\ MOD) \Longrightarrow wlsBinp\ delta\ binp \Longrightarrow lift\ (fst\ o\ hA)\ binp$

$=\ binp$

*<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* ::

$((\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)term \Rightarrow$

$(\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)term \times\ 'gTerm)$

$\Rightarrow$

$((\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)term \Rightarrow\ 'gTerm)$

**where** *fromIMor h*  $\equiv\ snd\ o\ h$

**definition** *fromIMorAbs* ::

$((\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)abs \Rightarrow$

$(\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)abs \times\ 'gAbs)$

$\Rightarrow$

$((\ 'index,\ 'bindex,\ 'varSort,\ 'var,\ 'opSym)abs \Rightarrow\ 'gAbs)$

**where** *fromIMorAbs hA*  $\equiv\ snd\ o\ hA$

Basic simplification rules:

**lemma** *fromIMor[simp]*:  $\text{fromIMor } h \ X' = \text{snd } (h \ X')$   
<proof>

**lemma** *fromIMorAbs[simp]*:  $\text{fromIMorAbs } hA \ A' = \text{snd } (hA \ A')$   
<proof>

**lemma** *fromIMor-snd-inp[simp]*:  
 $\text{wlsInp } \text{delta } \text{inp} \implies \text{lift } (\text{fromIMor } h) \ \text{inp} = \text{lift } (\text{snd } \circ h) \ \text{inp}$   
<proof>

**lemma** *fromIMorAbs-snd-binp[simp]*:  
 $\text{wlsBinp } \text{delta } \text{binp} \implies \text{lift } (\text{fromIMorAbs } hA) \ \text{binp} = \text{lift } (\text{snd } \circ hA) \ \text{binp}$   
<proof>

**lemmas** *fromIMor-basic-simps* =  
*fromIMor fromIMorAbs fromIMor-snd-inp fromIMorAbs-snd-binp*

Predicate simplification rules

Domain preservation

**lemma** *presWls-fromIMor[simp]*:  
 $\text{ipresWls } h \ (\text{fromMOD } \text{MOD}) \implies \text{presWls } (\text{fromIMor } h) \ \text{MOD}$   
<proof>

**lemma** *presWlsAbs-fromIMorAbs[simp]*:  
 $\text{ipresWlsAbs } hA \ (\text{fromMOD } \text{MOD}) \implies \text{presWlsAbs } (\text{fromIMorAbs } hA) \ \text{MOD}$   
<proof>

**lemma** *presWlsAll-fromIMorAll[simp]*:  
 $\text{ipresWlsAll } h \ hA \ (\text{fromMOD } \text{MOD}) \implies \text{presWlsAll } (\text{fromIMor } h) \ (\text{fromIMorAbs } hA) \ \text{MOD}$   
<proof>

**lemmas** *presWlsAll-fromIMorAll-simps* =  
*presWls-fromIMor presWlsAbs-fromIMorAbs presWlsAll-fromIMorAll*

Preservation of the constructs

**lemma** *presVar-fromIMor[simp]*:  
 $\text{ipresCons } h \ hA \ (\text{fromMOD } \text{MOD}) \implies \text{presVar } (\text{fromIMor } h) \ \text{MOD}$   
<proof>

**lemma** *presAbs-fromIMor[simp]*:  
**assumes** *ipresCons*  $h \ hA \ (\text{fromMOD } \text{MOD})$   
**shows** *presAbs*  $(\text{fromIMor } h) \ (\text{fromIMorAbs } hA) \ \text{MOD}$   
<proof>

**lemma** *presOp-fromIMor[simp]*:  
**assumes** *ipresCons*  $h \ hA \ (\text{fromMOD } \text{MOD})$   
**shows** *presOp*  $(\text{fromIMor } h) \ (\text{fromIMorAbs } hA) \ \text{MOD}$



*<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)  $\implies$  ipresFresh h (fromMOD MOD)*  
 *$\implies$  presFresh (fromIMor h) MOD*  
*<proof>*

**lemma** *presFreshAbs-fromIMor[simp]*:  
*ipresCons h hA (fromMOD MOD)  $\implies$  ipresFreshAbs hA (fromMOD MOD)*  
 *$\implies$  presFreshAbs (fromIMorAbs hA) MOD*  
*<proof>*

**lemma** *presFreshAll-fromIMor[simp]*:  
*ipresCons h hA (fromMOD MOD)  $\implies$  ipresFreshAll h hA (fromMOD MOD)*  
 *$\implies$  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)  $\implies$  ipresSwap h (fromMOD MOD)*  
 *$\implies$  presSwap (fromIMor h) MOD*  
*<proof>*

**lemma** *presSwapAbs-fromIMor[simp]*:  
*ipresCons h hA (fromMOD MOD)  $\implies$  ipresSwapAbs hA (fromMOD MOD)*  
 *$\implies$  presSwapAbs (fromIMorAbs hA) MOD*  
*<proof>*

**lemma** *presSwapAll-fromIMor[simp]*:  
*ipresCons h hA (fromMOD MOD)  $\implies$  ipresSwapAll h hA (fromMOD MOD)*  
 *$\implies$  presSwapAll (fromIMor h) (fromIMorAbs hA) MOD*  
*<proof>*

**lemmas** *presSwapAll-fromIMor-simps =*  
*presSwap-fromIMor presSwapAbs-fromIMor presSwapAll-fromIMor*

Preservation of subst

**lemma** *presSubst-fromIMor[simp]*:  
 $ipresCons\ h\ hA\ (fromMOD\ MOD) \implies ipresSubst\ h\ (fromMOD\ MOD)$   
 $\implies presSubst\ (fromIMor\ h)\ MOD$   
 ⟨proof⟩

**lemma** *presSubstAbs-fromIMor[simp]*:  
 $ipresCons\ h\ hA\ (fromMOD\ MOD) \implies ipresSubstAbs\ h\ hA\ (fromMOD\ MOD)$   
 $\implies presSubstAbs\ (fromIMor\ h)\ (fromIMorAbs\ hA)\ MOD$   
 ⟨proof⟩

**lemma** *presSubstAll-fromIMor[simp]*:  
 $ipresCons\ h\ hA\ (fromMOD\ MOD) \implies ipresSubstAll\ h\ hA\ (fromMOD\ MOD)$   
 $\implies presSubstAll\ (fromIMor\ h)\ (fromIMorAbs\ hA)\ MOD$   
 ⟨proof⟩

**lemmas** *presSubstAll-fromIMor-simps =*  
*presSubst-fromIMor presSubstAbs-fromIMor presSubstAll-fromIMor*

Morphisms

**lemma** *fromIMor-termFSwMorph[simp]*:  
 $termFSwImorph\ h\ hA\ (fromMOD\ MOD) \implies termFSwMorph\ (fromIMor\ h)\ (fromIMorAbs\ hA)\ MOD$   
 ⟨proof⟩

**lemma** *fromIMor-termFSbMorph[simp]*:  
 $termFSbImorph\ h\ hA\ (fromMOD\ MOD) \implies termFSbMorph\ (fromIMor\ h)\ (fromIMorAbs\ hA)\ MOD$   
 ⟨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*  $rec\ MOD \equiv fromIMor\ (iter\ (fromMOD\ MOD))$

**definition** *recAbs where*  $recAbs\ MOD \equiv fromIMorAbs\ (iterAbs\ (fromMOD\ MOD))$

Existence:

**theorem** *wlsFSw-recAll-termFSwMorph:*

$wlsFSw\ MOD \implies termFSwMorph\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

**theorem** *wlsFSb-recAll-termFSbMorph:*

$wlsFSb\ MOD \implies termFSbMorph\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

**theorem** *wlsFSwSb-recAll-termFSwSbMorph:*

$wlsFSwSb\ MOD \implies termFSwSbMorph\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

**theorem** *wlsFSbSw-recAll-termFSwSbMorph:*

$wlsFSbSw\ MOD \implies termFSwSbMorph\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

Uniqueness:

**lemma** *presCons-unique:*

**assumes** *presCons*  $f\ fA\ MOD$  **and** *presCons*  $g\ gA\ MOD$

**shows**  $(wls\ s\ X \longrightarrow f\ X = g\ X) \wedge (wlsAbs\ (us,s')\ A \longrightarrow fA\ A = gA\ A)$   
 $\langle proof \rangle$

**theorem** *wlsFSw-recAll-unique-presCons:*

**assumes** *wlsFSw*  $MOD$  **and** *presCons*  $h\ hA\ MOD$

**shows**  $(wls\ s\ X \longrightarrow h\ X = rec\ MOD\ X) \wedge$   
 $(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = recAbs\ MOD\ A)$   
 $\langle proof \rangle$

**theorem** *wlsFSb-recAll-unique-presCons:*

**assumes** *wlsFSb*  $MOD$  **and** *presCons*  $h\ hA\ MOD$

**shows**  $(wls\ s\ X \longrightarrow h\ X = rec\ MOD\ X) \wedge$   
 $(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = recAbs\ MOD\ A)$   
 $\langle proof \rangle$

**theorem** *wlsFSwSb-recAll-unique-presCons:*

**assumes** *wlsFSwSb*  $MOD$  **and** *presCons*  $h\ hA\ MOD$

**shows**  $(wls\ s\ X \longrightarrow h\ X = rec\ MOD\ X) \wedge$   
 $(wlsAbs\ (us,s')\ A \longrightarrow hA\ A = recAbs\ MOD\ A)$   
 $\langle proof \rangle$

**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**  
 $gFreshClsRev\ MOD \equiv gFreshGVarRev\ MOD \wedge gFreshGAbsRev\ MOD \wedge gFresh-$   
 $GOpRev\ MOD$

**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**

$gAbsInj \text{ MOD} \equiv \forall xs \ s \ x \ X' \ X \ X1' \ X1.$   
 $isInBar \ (xs,s) \wedge wls \ s \ X' \wedge gWls \ \text{MOD} \ s \ X \wedge wls \ s \ X1' \wedge gWls \ \text{MOD} \ s \ X1 \wedge$   
 $gAbs \ \text{MOD} \ xs \ x \ X' \ X = gAbs \ \text{MOD} \ xs \ x \ X1' \ X1$   
 $\longrightarrow$   
 $X = X1$

**definition**  $gOpInj$  **where**

$gOpInj \ \text{MOD} \equiv \forall \ \text{delta} \ \text{delta1} \ \text{inp}' \ \text{binp}' \ \text{inp} \ \text{binp} \ \text{inp1}' \ \text{binp1}' \ \text{inp1} \ \text{binp1}.$   
 $wlsInp \ \text{delta} \ \text{inp}' \wedge wlsBinp \ \text{delta} \ \text{binp}' \wedge gWlsInp \ \text{MOD} \ \text{delta} \ \text{inp} \wedge gWlsBinp$   
 $\text{MOD} \ \text{delta} \ \text{binp} \wedge$   
 $wlsInp \ \text{delta1} \ \text{inp1}' \wedge wlsBinp \ \text{delta1} \ \text{binp1}' \wedge gWlsInp \ \text{MOD} \ \text{delta1} \ \text{inp1} \wedge$   
 $gWlsBinp \ \text{MOD} \ \text{delta1} \ \text{binp1} \wedge$   
 $stOf \ \text{delta} = stOf \ \text{delta1} \wedge$   
 $gOp \ \text{MOD} \ \text{delta} \ \text{inp}' \ \text{inp} \ \text{binp}' \ \text{binp} = gOp \ \text{MOD} \ \text{delta1} \ \text{inp1}' \ \text{inp1} \ \text{binp1}' \ \text{binp1}$   
 $\longrightarrow$   
 $\text{delta} = \text{delta1} \wedge \text{inp} = \text{inp1} \wedge \text{binp} = \text{binp1}$

**definition**  $gVarGOpInj$  **where**

$gVarGOpInj \ \text{MOD} \equiv \forall \ xs \ x \ \text{delta} \ \text{inp}' \ \text{binp}' \ \text{inp} \ \text{binp}.$   
 $wlsInp \ \text{delta} \ \text{inp}' \wedge wlsBinp \ \text{delta} \ \text{binp}' \wedge gWlsInp \ \text{MOD} \ \text{delta} \ \text{inp} \wedge gWlsBinp$   
 $\text{MOD} \ \text{delta} \ \text{binp} \wedge$   
 $asSort \ xs = stOf \ \text{delta}$   
 $\longrightarrow$   
 $gVar \ \text{MOD} \ xs \ x \neq gOp \ \text{MOD} \ \text{delta} \ \text{inp}' \ \text{inp} \ \text{binp}' \ \text{binp}$

**definition**  $gConsInj$  **where**

$gConsInj \ \text{MOD} \equiv gVarInj \ \text{MOD} \wedge gAbsInj \ \text{MOD} \wedge gOpInj \ \text{MOD} \wedge gVarGOpInj$   
 $\text{MOD}$

**lemmas**  $gConsInj\text{-defs} = gConsInj\text{-def}$   
 $gVarInj\text{-def} \ gAbsInj\text{-def} \ gOpInj\text{-def} \ gVarGOpInj\text{-def}$

Abstraction renaming for swapping

**definition**  $gAbsRenS$  **where**

$gAbsRenS \ \text{MOD} \equiv \forall \ xs \ y \ x \ s \ X' \ X.$   
 $isInBar \ (xs,s) \wedge wls \ s \ X' \wedge gWls \ \text{MOD} \ s \ X \longrightarrow$   
 $fresh \ xs \ y \ X' \wedge gFresh \ \text{MOD} \ xs \ y \ X' \ X \longrightarrow$   
 $gAbs \ \text{MOD} \ xs \ y \ (X' \ #[y \wedge x]\text{-xs}) \ (gSwap \ \text{MOD} \ xs \ y \ x \ X' \ X) =$   
 $gAbs \ \text{MOD} \ xs \ x \ X' \ X$

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**

$gAbsIndif \ \text{MOD} \equiv \forall \ xs \ s \ x \ X1' \ X2' \ X.$

$$\text{isInBar } (xs,s) \wedge \text{wls } s \ X1' \wedge \text{wls } s \ X2' \wedge \text{gWls } MOD \ s \ X \longrightarrow \\ \text{gAbs } MOD \ xs \ x \ X1' \ X = \text{gAbs } MOD \ xs \ x \ X2' \ X$$

**definition** *gOpIndif* **where**

$$\text{gOpIndif } MOD \equiv \forall \ \text{delta } \text{inp1}' \ \text{inp2}' \ \text{inp} \ \text{binp1}' \ \text{binp2}' \ \text{binp}. \\ \text{wlsInp } \text{delta } \ \text{inp1}' \wedge \text{wlsBinp } \text{delta } \ \text{binp1}' \wedge \text{wlsInp } \text{delta } \ \text{inp2}' \wedge \text{wlsBinp } \text{delta} \\ \text{binp2}' \wedge \\ \text{gWlsInp } MOD \ \text{delta } \ \text{inp} \wedge \text{gWlsBinp } MOD \ \text{delta } \ \text{binp} \\ \longrightarrow \\ \text{gOp } MOD \ \text{delta } \ \text{inp1}' \ \text{inp} \ \text{binp1}' \ \text{binp} = \text{gOp } MOD \ \text{delta } \ \text{inp2}' \ \text{inp} \ \text{binp2}' \ \text{binp}$$

**definition** *gConsIndif* **where**

$$\text{gConsIndif } MOD \equiv \text{gOpIndif } MOD \wedge \text{gAbsIndif } MOD$$

**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**

$$\text{gInduct } MOD \equiv \forall \ \text{phi } \text{phiAbs } s \ X \ us \ s' \ A. \\ ( \\ (\forall \ xs \ x. \ \text{phi } (\text{asSort } xs) \ (\text{gVar } MOD \ xs \ x)) \\ \wedge \\ (\forall \ \text{delta } \ \text{inp}' \ \text{inp} \ \text{binp}' \ \text{binp}. \\ \text{wlsInp } \text{delta } \ \text{inp}' \wedge \text{wlsBinp } \text{delta } \ \text{binp}' \wedge \text{gWlsInp } MOD \ \text{delta } \ \text{inp} \wedge \text{gWlsBinp} \\ MOD \ \text{delta } \ \text{binp} \wedge \\ \text{liftAll2 } \text{phi } (\text{arOf } \text{delta}) \ \text{inp} \wedge \text{liftAll2 } \text{phiAbs } (\text{barOf } \text{delta}) \ \text{binp} \\ \longrightarrow \text{phi } (\text{stOf } \text{delta}) \ (\text{gOp } MOD \ \text{delta } \ \text{inp}' \ \text{inp} \ \text{binp}' \ \text{binp})) \\ \wedge \\ (\forall \ xs \ s \ x \ X' \ X. \\ \text{isInBar } (xs,s) \wedge \text{wls } s \ X' \wedge \text{gWls } MOD \ s \ X \wedge \\ \text{phi } s \ X \\ \longrightarrow \text{phiAbs } (xs,s) \ (\text{gAbs } MOD \ xs \ x \ X' \ X)) \\ ) \\ \longrightarrow \\ (\text{gWls } MOD \ s \ X \longrightarrow \text{phi } s \ X) \wedge \\ (\text{gWlsAbs } MOD \ (us,s') \ A \longrightarrow \text{phiAbs } (us,s') \ A)$$

**lemma** *gInduct-elim*:

**assumes** *gInduct* *MOD* **and**

*Var*:  $\bigwedge \ xs \ x. \ \text{phi } (\text{asSort } xs) \ (\text{gVar } MOD \ xs \ x)$  **and**

*Op*:

$\bigwedge \ \text{delta } \ \text{inp}' \ \text{inp} \ \text{binp}' \ \text{binp}.$

$\llbracket \text{wlsInp } \text{delta } \ \text{inp}'; \text{wlsBinp } \text{delta } \ \text{binp}'; \text{gWlsInp } MOD \ \text{delta } \ \text{inp}; \text{gWlsBinp } MOD \\ \text{delta } \ \text{binp};$

$\text{liftAll2 } \text{phi } (\text{arOf } \text{delta}) \ \text{inp}; \text{liftAll2 } \text{phiAbs } (\text{barOf } \text{delta}) \ \text{binp} \rrbracket$

$\implies \text{phi} (\text{stOf delta}) (\text{gOp MOD delta inp' inp binp' binp})$  **and**  
**Abs:**  
 $\bigwedge xs\ s\ x\ X'\ X.$   
 $\llbracket \text{isInBar} (xs,s); \text{wls}\ s\ X'; \text{gWls MOD}\ s\ X; \text{phi}\ s\ X \rrbracket$   
 $\implies \text{phiAbs} (xs,s) (\text{gAbs MOD}\ xs\ x\ X'\ X)$   
**shows**  
 $(\text{gWls MOD}\ s\ X \longrightarrow \text{phi}\ s\ X) \wedge$   
 $(\text{gWlsAbs MOD}\ (us,s')\ A \longrightarrow \text{phiAbs}\ (us,s')\ A)$   
 $\langle \text{proof} \rangle$

## 10.8.2 Relevant predicates on maps from the term model

Reflection of freshness

**definition** *reflFresh* **where**

$\text{reflFresh}\ h\ \text{MOD} \equiv \forall\ ys\ y\ s\ X.$   
 $\text{wls}\ s\ X \longrightarrow$   
 $\text{gFresh MOD}\ ys\ y\ X\ (h\ X) \longrightarrow \text{fresh}\ ys\ y\ X$

**definition** *reflFreshAbs* **where**

$\text{reflFreshAbs}\ hA\ \text{MOD} \equiv \forall\ ys\ y\ us\ s\ A.$   
 $\text{wlsAbs}\ (us,s)\ A \longrightarrow$   
 $\text{gFreshAbs MOD}\ ys\ y\ A\ (hA\ A) \longrightarrow \text{freshAbs}\ ys\ y\ A$

**definition** *reflFreshAll* **where**

$\text{reflFreshAll}\ h\ hA\ \text{MOD} \equiv \text{reflFresh}\ h\ \text{MOD} \wedge \text{reflFreshAbs}\ hA\ \text{MOD}$

**lemmas** *reflFreshAll-defs* = *reflFreshAll-def*  
*reflFresh-def* *reflFreshAbs-def*

Injectiveness

**definition** *isInj* **where**

$\text{isInj}\ h \equiv \forall\ s\ X\ Y.$   
 $\text{wls}\ s\ X \wedge \text{wls}\ s\ Y \longrightarrow$   
 $h\ X = h\ Y \longrightarrow X = Y$

**definition** *isInjAbs* **where**

$\text{isInjAbs}\ hA \equiv \forall\ us\ s\ A\ B.$   
 $\text{wlsAbs}\ (us,s)\ A \wedge \text{wlsAbs}\ (us,s)\ B \longrightarrow$   
 $hA\ A = hA\ B \longrightarrow A = B$

**definition** *isInjAll* **where**

$\text{isInjAll}\ h\ hA \equiv \text{isInj}\ h \wedge \text{isInjAbs}\ hA$

**lemmas** *isInjAll-defs* = *isInjAll-def*  
*isInj-def* *isInjAbs-def*

Surjectiveness

**definition** *isSurj* **where**

$isSurj\ h\ MOD \equiv \forall\ s\ X.$   
 $gWls\ MOD\ s\ X \longrightarrow$   
 $(\exists\ X'.\ wls\ s\ X' \wedge h\ X' = X)$

**definition**  $isSurjAbs$  **where**

$isSurjAbs\ hA\ MOD \equiv \forall\ us\ s\ A.$   
 $gWlsAbs\ MOD\ (us,s)\ A \longrightarrow$   
 $(\exists\ A'.\ wlsAbs\ (us,s)\ A' \wedge hA\ A' = A)$

**definition**  $isSurjAll$  **where**

$isSurjAll\ h\ hA\ MOD \equiv isSurj\ h\ MOD \wedge isSurjAbs\ hA\ MOD$

**lemmas**  $isSurjAll-defs = isSurjAll-def$   
 $isSurj-def\ isSurjAbs-def$

### 10.8.3 Criterion for the reflection of freshness

First an auxiliary fact, independent of the type of model:

**lemma**  $gFreshClsRev-recAll-reflFreshAll$ :  
**assumes**  $pWls$ :  $presWlsAll\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
**and**  $pCons$ :  $presCons\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
**and**  $pFresh$ :  $presFreshAll\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
**and**  $**$ :  $gFreshClsRev\ MOD$   
**shows**  $reflFreshAll\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

For fresh-swap models

**theorem**  $wlsFSw-recAll-reflFreshAll$ :  
 $wlsFSw\ MOD \implies gFreshClsRev\ MOD \implies reflFreshAll\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

For fresh-subst models

**theorem**  $wlsFSb-recAll-reflFreshAll$ :  
 $wlsFSb\ MOD \implies gFreshClsRev\ MOD \implies reflFreshAll\ (rec\ MOD)\ (recAbs\ MOD)\ MOD$   
 $\langle proof \rangle$

### 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**  $isInjAll\ (rec\ MOD)\ (recAbs\ MOD)$   
 $\langle proof \rangle$

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*  $\implies$  *gConsIndif MOD*  $\implies$  *gInduct MOD*  
 $\implies$  *isSurjAll (rec MOD) (recAbs MOD) MOD*  
 ⟨*proof*⟩

For fresh-sbst models

**theorem** *wlsFSb-recAll-isSurjAll*:  
*wlsFSb MOD*  $\implies$  *gConsIndif MOD*  $\implies$  *gInduct MOD*  
 $\implies$  *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**