Mechanically Proving Determinacy of Hierarchical Block Diagram Translations

July 4, 2018

Contents

1 List Operations. Permutations and Substitutions

theory *ListProp* **imports** *Main* ~~/*src*/*HOL*/*Library*/*Permutation*
begin

lemma *perm-mset*: *perm x y* = (*mset x* = *mset y*)

lemma *perm-tp*: *perm* (*x*@*y*) (*y*@*x*)

lemma *perm-union-left*: *perm x z* ⟹ *perm* (*x* @ *y*) (*z* @ *y*)

lemma *perm-union-right*: *perm x z* ⟹ *perm* (*y* @ *x*) (*y* @ *z*)

lemma *perm-trans*: *perm x y* ⟹ *perm y z* ⟹ *perm x z*

lemma *perm-sym*: *perm x y* ⟹ *perm y x*

lemma *perm-length*: *perm u v* ⟹ *length u* = *length v*

lemma *perm-set-eq*: *perm x y* ⟹ *set x* = *set y*

lemma *perm-empty*[*simp*]: (*perm* [] *v*) = (*v* = []) **and** (*perm v* []) = (*v* = [])

lemma *perm-refl*[*simp*]: *perm x x*

lemma *dist-perm*: ⋀ *y* . *distinct x* ⟹ *perm x y* ⟹ *distinct y*

lemma *split-perm*: *perm* (*a* # *x*) *x'* = (∃ *y y'* . *x'* = *y* @ *a* # *y'* ∧ *perm x* (*y* @ *y'*))

fun *subst*:: *'a list* ⇒ *'a list* ⇒ *'a* ⇒ *'a* **where**
subst [] [] *c* = *c* |
subst (*a*#*x*) (*b*#*y*) *c* = (*if a* = *c then b else subst x y c*) |
subst x y c = *undefined*

lemma *subst-notin* [*simp*]: ⋀ *y* . *length x* = *length y* ⟹ *a* ∉ *set x* ⟹ *subst x y a* = *a*

lemma *subst-cons-a*:⋀ *y* . *distinct x* ⟹ *a* ∉ *set x* ⟹ *b* ∉ *set x* ⟹ *length x* = *length y* ⟹ *subst* (*a* # *x*) (*b* # *y*) *c* = (*subst x y* (*subst* [*a*] [*b*] *c*))

lemma *subst-eq*: *subst x x y* = *y*

fun *Subst* :: *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
Subst x y [] = [] |
Subst x y (*a* # *z*) = *subst x y a* # (*Subst x y z*)

lemma *Subst-empty*[*simp*]: *Subst* [] [] *y* = *y*

lemma *Subst-eq*: *Subst x x y* = *y*

lemma *Subst-append*: *Subst a b* (*x*@*y*) = *Subst a b x* @ *Subst a b y*

lemma *Subst-notin*[*simp*]: *a* ∉ *set z* ⟹ *Subst* (*a* # *x*) (*b* # *y*) *z* = *Subst x y z*

lemma *Subst-all*[*simp*]: ⋀ *v* . *distinct u* ⟹*length u* = *length v* ⟹ *Subst u v u* = *v*

lemma *Subst-inex*[*simp*]: ⋀ *b*. *set a* ∩ *set x* = {} ⟹ *length a* = *length b* ⟹ *Subst a b x* = *x*

lemma *set-Subst*: *set* (*Subst* [*a*] [*b*] *x*) = (*if a* ∈ *set x then* (*set x* − {*a*}) ∪ {*b*} *else set x*)

lemma *distinct-Subst*: *distinct* (*b*#*x*) ⟹ *distinct* (*Subst* [*a*] [*b*] *x*)

lemma *inter-Subst*: *distinct*(*b*#*y*) ⟹ *set x* ∩ *set y* = {} ⟹ *b* ∉ *set x* ⟹ *set x* ∩ *set* (*Subst* [*a*] [*b*] *y*) = {}

lemma *incl-Subst*: *distinct*(*b*#*x*) ⟹ *set y* ⊆ *set x* ⟹ *set* (*Subst* [*a*] [*b*] *y*) ⊆ *set* (*Subst* [*a*] [*b*] *x*)

lemma *subst-in-set*: ⋀*y*. *length x* = *length y* ⟹ *a* ∈ *set x* ⟹ *subst x y a* ∈ *set y*

lemma *Subst-set-incl*: *length x* = *length y* ⟹ *set z* ⊆ *set x* ⟹ *set* (*Subst x y z*) ⊆ *set y*

lemma *subst-not-in*: ⋀ *y* . *a* ∉ *set x'* ⟹ *length x* = *length y* ⟹ *length x'* = *length y'* ⟹ *subst* (*x* @ *x'*) (*y* @ *y'*) *a* = *subst x y a*

lemma *subst-not-in-b*: ⋀ *y* . *a* ∉ *set x* ⟹ *length x* = *length y* ⟹ *length x'* = *length y'* ⟹ *subst* (*x* @ *x'*) (*y* @ *y'*) *a* = *subst x' y' a*

lemma *Subst-not-in*: *set x'* ∩ *set z* = {} ⟹ *length x* = *length y* ⟹ *length x'* = *length y'* ⟹ *Subst* (*x* @ *x'*) (*y* @ *y'*) *z* = *Subst x y z*

lemma *Subst-not-in-a*: *set x* ∩ *set z* = {} ⟹ *length x* = *length y* ⟹ *length x'*

$= \mathit{length}\ y' \Longrightarrow \mathit{Subst}\ (x\ @\ x')\ (y\ @\ y')\ z = \mathit{Subst}\ x'\ y'\ z$

lemma *subst-cancel-right* [*simp*]: $\bigwedge\ y\ z\ .\ \mathit{set}\ x \cap \mathit{set}\ y = \{\} \Longrightarrow \mathit{length}\ y = \mathit{length}\ z \Longrightarrow \mathit{subst}\ (x\ @\ y)\ (x\ @\ z)\ a = \mathit{subst}\ y\ z\ a$

lemma *Subst-cancel-right*: $\mathit{set}\ x \cap \mathit{set}\ y = \{\} \Longrightarrow \mathit{length}\ y = \mathit{length}\ z \Longrightarrow \mathit{Subst}\ (x\ @\ y)\ (x\ @\ z)\ w = \mathit{Subst}\ y\ z\ w$

lemma *subst-cancel-left* [*simp*]: $\bigwedge\ y\ z\ .\ \mathit{set}\ x \cap \mathit{set}\ z = \{\} \Longrightarrow \mathit{length}\ x = \mathit{length}\ y \Longrightarrow \mathit{subst}\ (x\ @\ z)\ (y\ @\ z)\ a = \mathit{subst}\ x\ y\ a$

lemma *Subst-cancel-left*: $\mathit{set}\ x \cap \mathit{set}\ z = \{\} \Longrightarrow \mathit{length}\ x = \mathit{length}\ y \Longrightarrow \mathit{Subst}\ (x\ @\ z)\ (y\ @\ z)\ w = \mathit{Subst}\ x\ y\ w$

lemma *Subst-cancel-right-a*: $a \notin \mathit{set}\ y \Longrightarrow \mathit{length}\ y = \mathit{length}\ z \Longrightarrow \mathit{Subst}\ (a\ \#\ y)\ (a\ \#\ z)\ w = \mathit{Subst}\ y\ z\ w$

lemma *subst-subst-id* [*simp*]: $\bigwedge\ y\ .\ a \in \mathit{set}\ y \Longrightarrow \mathit{distinct}\ x \Longrightarrow \mathit{length}\ x = \mathit{length}\ y \Longrightarrow \mathit{subst}\ x\ y\ (\mathit{subst}\ y\ x\ a) = a$

lemma *Subst-Subst-id*[*simp*]: $\mathit{set}\ z \subseteq \mathit{set}\ y \Longrightarrow \mathit{distinct}\ x \Longrightarrow \mathit{length}\ x = \mathit{length}\ y \Longrightarrow \mathit{Subst}\ x\ y\ (\mathit{Subst}\ y\ x\ z) = z$

lemma *Subst-cons-aux-a*: $\mathit{set}\ x \cap \mathit{set}\ y = \{\} \Longrightarrow \mathit{distinct}\ y \Longrightarrow \mathit{length}\ y = \mathit{length}\ z \Longrightarrow \mathit{Subst}\ (x\ @\ y)\ (x\ @\ z)\ y = z$

lemma *Subst-set-empty* [*simp*]: $\mathit{set}\ z \cap \mathit{set}\ x = \{\} \Longrightarrow \mathit{length}\ x = \mathit{length}\ y \Longrightarrow \mathit{Subst}\ x\ y\ z = z$

lemma *length-Subst*[*simp*]: $\mathit{length}\ (\mathit{Subst}\ x\ y\ z) = \mathit{length}\ z$

lemma *subst-Subst*: $\bigwedge\ y\ y'\ .\ \mathit{length}\ y = \mathit{length}\ y' \Longrightarrow a \in \mathit{set}\ w \Longrightarrow \mathit{subst}\ w\ (\mathit{Subst}\ y\ y'\ w)\ a = \mathit{subst}\ y\ y'\ a$

lemma *Subst-Subst*: $\mathit{length}\ y = \mathit{length}\ y' \Longrightarrow \mathit{set}\ z \subseteq \mathit{set}\ w \Longrightarrow \mathit{Subst}\ w\ (\mathit{Subst}\ y\ y'\ w)\ z = \mathit{Subst}\ y\ y'\ z$

primrec *listinter* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** ⊗ *60*) **where**
[] ⊗ y = [] |
(a # x) ⊗ y = (*if* $a \in \mathit{set}\ y$ *then* a # (x ⊗ y) *else* x ⊗ y)

lemma *inter-filter*: $x \otimes y = \mathit{filter}\ (\lambda\ a\ .\ a \in \mathit{set}\ y)\ x$

lemma *inter-append*: $\mathit{set}\ y \cap \mathit{set}\ z = \{\} \Longrightarrow \mathit{perm}\ (x \otimes (y\ @\ z))\ ((x \otimes y)\ @\ (x \otimes z))$

lemma *append-inter*: $(x \mathbin{@} y) \otimes z = (x \otimes z) \mathbin{@} (y \otimes z)$

lemma *notin-inter* [*simp*]: $a \notin set\ x \Longrightarrow a \notin set\ (x \otimes y)$

lemma *distinct-inter*: $distinct\ x \Longrightarrow distinct\ (x \otimes y)$

lemma *set-inter*: $set\ (x \otimes y) = set\ x \cap set\ y$

primrec *diff* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ (**infixl** $\ominus$ *52*) **where**
$[] \ominus y = [] \mid$
$(a \# x) \ominus y = (if\ a \in set\ y\ then\ x \ominus y\ else\ a \# (x \ominus y))$

lemma *diff-filter*: $x \ominus y = filter\ (\lambda\ a\ .\ a \notin set\ y)\ x$

lemma *diff-distinct*: $set\ x \cap set\ y = \{\} \Longrightarrow (y \ominus x) = y$

lemma *set-diff*: $set\ (x \ominus y) = set\ x - set\ y$

lemma *distinct-diff*: $distinct\ x \Longrightarrow distinct\ (x \ominus y)$

definition *addvars* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ (**infixl** $\oplus$ *55*) **where**
$addvars\ x\ y = x \mathbin{@} (y \ominus x)$

lemma *addvars-distinct*: $set\ x \cap set\ y = \{\} \Longrightarrow x \oplus y = x \mathbin{@} y$

lemma *set-addvars*: $set\ (x \oplus y) = set\ x \cup set\ y$

lemma *distinct-addvars*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow distinct\ (x \oplus y)$

lemma *mset-inter-diff*: $mset\ oa = mset\ (oa \otimes ia) + mset\ (oa \ominus (oa \otimes ia))$

lemma *diff-inter-left*: $(x \ominus (x \otimes y)) = (x \ominus y)$

lemma *diff-inter-right*: $(x \ominus (y \otimes x)) = (x \ominus y)$

lemma *addvars-minus*: $(x \oplus y) \ominus z = (x \ominus z) \oplus (y \ominus z)$

lemma *addvars-assoc*: $x \oplus y \oplus z = x \oplus (y \oplus z)$

lemma *diff-sym*: $(x \ominus y \ominus z) = (x \ominus z \ominus y)$

lemma *diff-union*: $(x \ominus y \mathbin{@} z) = (x \ominus y \ominus z)$

lemma *diff-notin*: $set\ x \cap set\ z = \{\} \Longrightarrow (x \ominus (y \ominus z)) = (x \ominus y)$

lemma *union-diff*: $x \mathbin{@} y \ominus z = ((x \ominus z) \mathbin{@} (y \ominus z))$

lemma *diff-inter-empty*: $set\ x \cap set\ y = \{\} \Longrightarrow x \ominus y \otimes z = x$

lemma *inter-diff-empty*: $set\ x \cap set\ z = \{\} \Longrightarrow x \otimes (y \ominus z) = (x \otimes y)$

lemma *inter-diff-distrib*: $(x \ominus y) \otimes z = ((x \otimes z) \ominus (y \otimes z))$

lemma *diff-emptyset*: $x \ominus [] = x$

lemma *diff-eq*: $x \ominus x = []$

lemma *diff-subset*: $set\ x \subseteq set\ y \Longrightarrow x \ominus y = []$

lemma *empty-inter*: $set\ x \cap set\ y = \{\} \Longrightarrow x \otimes y = []$

lemma *empty-inter-diff*: $set\ x \cap set\ y = \{\} \Longrightarrow x \otimes (y \ominus z) = []$

lemma *inter-addvars-empty*: $set\ x \cap set\ z = \{\} \Longrightarrow x \otimes y\ @\ z = x \otimes y$

lemma *diff-disjoint*: $set\ x \cap set\ y = \{\} \Longrightarrow x \ominus y = x$

lemma *addvars-empty*[*simp*]: $x \oplus [] = x$

lemma *empty-addvars*[*simp*]: $[] \oplus x = x$

lemma *distrib-diff-addvars*: $x \ominus (y\ @\ z) = ((x \ominus y) \otimes (x \ominus z))$

lemma *inter-subset*: $x \otimes (x \ominus y) = (x \ominus y)$

lemma *diff-cancel*: $x \ominus y \ominus (z \ominus y) = (x \ominus y \ominus z)$

lemma *diff-cancel-set*: $set\ x \cap set\ u = \{\} \Longrightarrow x \ominus y \ominus (z \ominus u) = (x \ominus y \ominus z)$

lemma *inter-subset-l1*: $\bigwedge y.\ distinct\ x \Longrightarrow length\ y = 1 \Longrightarrow set\ y \subseteq set\ x \Longrightarrow x \otimes y = y$

lemma *perm-diff-left-inter*: $perm\ (x \ominus y)\ (((x \ominus y) \otimes z)\ @\ ((x \ominus y) \ominus z))$

lemma *perm-diff-right-inter*: $perm\ (x \ominus y)\ (((x \ominus y) \ominus z)\ @\ ((x \ominus y) \otimes z))$

lemma *perm-switch-aux-a*: $perm\ x\ ((x \ominus y)\ @\ (x \otimes y))$

lemma *perm-switch-aux-b*: $perm\ (x\ @\ (y \ominus x))\ ((x \ominus y)\ @\ (x \otimes y)\ @\ (y \ominus x))$

lemma *perm-switch-aux-c*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ ((y \otimes x)\ @\ (y \ominus x))\ y$

lemma *perm-switch-aux-d*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ (x \otimes y)\ (y \otimes x)$

lemma *perm-switch-aux-e*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ ((x \otimes y)\ @\ (y \ominus x))\ ((y \otimes x)\ @\ (y \ominus x))$

lemma *perm-switch-aux-f*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ ((x \otimes y)\ @\ (y \ominus x))\ y$

lemma *perm-switch-aux-h*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ ((x \ominus y)\ @\ (x \otimes y)\ @\ (y \ominus x))\ ((x \ominus y)\ @\ y)$

lemma *perm-switch*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ (x\ @\ (y \ominus x))\ ((x \ominus y)\ @\ y)$

lemma *perm-aux-a*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow x \otimes y = x \Longrightarrow perm\ (x\ @\ (y \ominus x))\ y$

lemma *ZZZ-a*: $x \oplus (y \ominus x) = (x \oplus y)$

lemma *ZZZ-b*: $set\ (y \otimes z) \cap set\ x = \{\} \Longrightarrow (x \ominus (y \ominus z) \ominus (z \ominus y)) = (x \ominus y \ominus z)$

lemma *subst-subst*: $\bigwedge y\ z\ .\ a \in set\ z \Longrightarrow distinct\ x \Longrightarrow length\ x = length\ y \Longrightarrow length\ z = length\ x$
$\Longrightarrow subst\ x\ y\ (subst\ z\ x\ a) = subst\ z\ y\ a$

lemma *Subst-Subst-a*: $set\ u \subseteq set\ z \Longrightarrow distinct\ x \Longrightarrow length\ x = length\ y \Longrightarrow length\ z = length\ x$
$\Longrightarrow Subst\ x\ y\ (Subst\ z\ x\ u) = (Subst\ z\ y\ u)$

lemma *subst-in*: $\bigwedge x'\ .\ length\ x = length\ x' \Longrightarrow a \in set\ x \Longrightarrow subst\ (x\ @\ y)\ (x'\ @\ y')\ a = subst\ x\ x'\ a$

lemma *subst-switch*: $\bigwedge x'\ .\ set\ x \cap set\ y = \{\} \Longrightarrow length\ x = length\ x' \Longrightarrow length\ y = length\ y'$
$\Longrightarrow subst\ (x\ @\ y)\ (x'\ @\ y')\ a = subst\ (y\ @\ x)\ (y'\ @\ x')\ a$

lemma *Subst-switch*: $set\ x \cap set\ y = \{\} \Longrightarrow length\ x = length\ x' \Longrightarrow length\ y = length\ y'$
$\Longrightarrow Subst\ (x\ @\ y)\ (x'@\ y')\ z = Subst\ (y\ @\ x)\ (y'@\ x')\ z$

lemma *subst-comp*: $\bigwedge x'\ .\ set\ x \cap set\ y = \{\} \Longrightarrow set\ x' \cap set\ y = \{\} \Longrightarrow length\ x = length\ x'$
$\Longrightarrow length\ y = length\ y' \Longrightarrow subst\ (x\ @\ y)\ (x'\ @\ y')\ a = subst\ y\ y'\ (subst\ x\ x'\ a)$

lemma *Subst-comp*: $set\ x \cap set\ y = \{\} \Longrightarrow set\ x' \cap set\ y = \{\} \Longrightarrow length\ x = length\ x'$
$\Longrightarrow length\ y = length\ y' \Longrightarrow Subst\ (x\ @\ y)\ (x'\ @\ y')\ z = Subst\ y\ y'\ (Subst\ x\ x'\ z)$

lemma *set-subst*: $\bigwedge u'$. *length* u = *length* $u' \Longrightarrow$ *subst* u u' $a \in$ *set* $u' \cup$ $(\{a\} -$ *set* $u)$

lemma *set-Subst-a*: *length* u = *length* $u' \Longrightarrow$ *set* (*Subst* u u' z) $\subseteq$ *set* $u' \cup$ (*set* $z -$ *set* u)

lemma *set-SubstI*: *length* u = *length* $u' \Longrightarrow$ *set* $u' \cup$ (*set* $z -$ *set* u) $\subseteq X \Longrightarrow$ *set* (*Subst* u u' z) $\subseteq X$

lemma *not-in-set-diff*: $a \notin$ *set* $x \Longrightarrow x \ominus ys$ @ $a \# zs = x \ominus ys$ @ zs

lemma [*simp*]: $(X \cap (Y \cup Z) = \{\}) = (X \cap Y = \{\} \wedge X \cap Z = \{\})$

lemma *Comp-assoc-new-subst-aux*: *set* $u \cap$ *set* $y \cap$ *set* $z = \{\} \Longrightarrow$ *distinct* z $\Longrightarrow$ *length* u = *length* u'
$\Longrightarrow$ *Subst* $(z \ominus v)$ (*Subst* u u' $(z \ominus v)$) z = *Subst* $(u \ominus y \ominus v)$ (*Subst* u u' $(u \ominus y \ominus v)$) z

lemma [*simp*]: $(x \ominus y \ominus (y \ominus z)) = (x \ominus y)$

lemma [*simp*]: $(x \ominus y \ominus (y \ominus z \ominus z')) = (x \ominus y)$

lemma *diff-addvars*: $x \ominus (y \oplus z) = (x \ominus y \ominus z)$

lemma *diff-redundant-a*: $x \ominus y \ominus z \ominus (y \ominus u) = (x \ominus y \ominus z)$

lemma *diff-redundant-b*: $x \ominus y \ominus z \ominus (z \ominus u) = (x \ominus y \ominus z)$

lemma *diff-redundant-c*: $x \ominus y \ominus z \ominus (y \ominus u \ominus v) = (x \ominus y \ominus z)$

lemma *diff-redundant-d*: $x \ominus y \ominus z \ominus (z \ominus u \ominus v) = (x \ominus y \ominus z)$

lemma *set-list-empty*: *set* $x = \{\} \Longrightarrow x = []$

lemma [*simp*]: $(x \ominus x \otimes y) \otimes (y \ominus x \otimes y) = []$

lemma [*simp*]: *set* $x \cap$ *set* $(y \ominus x) = \{\}$

lemma [*simp*]: *distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $x \subseteq$ *set* $y \Longrightarrow$ *perm* $(x$ @ $(y \ominus x))$ y

lemma [*simp*]: *perm* x $y \Longrightarrow$ *set* $x \subseteq$ *set* y
lemma [*simp*]: *perm* x $y \Longrightarrow$ *set* $y \subseteq$ *set* x

lemma [*simp*]: *set* $(x \ominus y) \subseteq$ *set* x

lemma *perm-diff* [*simp*]: $\bigwedge x'$. *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow$ *perm* $(x \ominus y)$ $(x' \ominus y')$

lemma [*simp*]: *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow$ *perm* $(x\ @\ y)\ (x'\ @\ y')$

lemma [*simp*]: *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow$ *perm* $(x \oplus y)\ (x' \oplus y')$

thm *distinct-diff*

declare *distinct-diff* [*simp*]

lemma [*simp*]: $\bigwedge x'$. *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow$ *perm* $(x \otimes y)\ (x' \otimes y')$

declare *distinct-inter* [*simp*]

lemma *perm-ops*: *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow f = op \otimes \vee f = op \ominus \vee f = op \oplus \Longrightarrow$ *perm* $(f\ x\ y)\ (f\ x'\ y')$

lemma [*simp*]: *perm* $x'\ x \Longrightarrow$ *perm* $y'\ y \Longrightarrow f = op \otimes \vee f = op \ominus \vee f = op \oplus \Longrightarrow$ *perm* $(f\ x\ y)\ (f\ x'\ y')$

lemma [*simp*]: *perm* $x\ x' \Longrightarrow$ *perm* $y'\ y \Longrightarrow f = op \otimes \vee f = op \ominus \vee f = op \oplus \Longrightarrow$ *perm* $(f\ x\ y)\ (f\ x'\ y')$

lemma [*simp*]: *perm* $x'\ x \Longrightarrow$ *perm* $y\ y' \Longrightarrow f = op \otimes \vee f = op \ominus \vee f = op \oplus \Longrightarrow$ *perm* $(f\ x\ y)\ (f\ x'\ y')$

lemma *diff-cons*: $(x \ominus (a\ \#\ y)) = (x \ominus [a] \ominus y)$

lemma [*simp*]: $x \oplus y \oplus x = x \oplus y$

lemma *subst-subst-inv*: $\bigwedge y$. *distinct* $y \Longrightarrow$ *length* $x =$ *length* $y \Longrightarrow a \in$ *set* $x \Longrightarrow$ *subst* $y\ x$ (*subst* $x\ y\ a$) $= a$

lemma *Subst-Subst-inv*: *distinct* $y \Longrightarrow$ *length* $x =$ *length* $y \Longrightarrow$ *set* $z \subseteq$ *set* $x \Longrightarrow$ *Subst* $y\ x$ (*Subst* $x\ y\ z$) $= z$

lemma *perm-append*: *perm* $x\ x' \Longrightarrow$ *perm* $y\ y' \Longrightarrow$ *perm* $(x\ @\ y)\ (x'\ @\ y')$

lemma $x' = y\ @\ a\ \#\ y' \Longrightarrow$ *perm* $x\ (y\ @\ y') \Longrightarrow$ *perm* $(a\ \#\ x)\ x'$

lemma *perm-diff-eq*: $perm\ y\ y' \Longrightarrow (x \ominus y) = (x \ominus y')$

lemma [*simp*]: $A \cap B = \{\} \Longrightarrow x \in A \Longrightarrow x \in B \Longrightarrow False$
lemma [*simp*]: $A \cap B = \{\} \Longrightarrow x \in A \Longrightarrow x \notin B$

lemma [*simp*]: $B \cap A = \{\} \Longrightarrow x \in A \Longrightarrow x \notin B$
lemma [*simp*]: $B \cap A = \{\} \Longrightarrow x \in A \Longrightarrow x \in B \Longrightarrow False$

lemma *distinct-perm-set-eq*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ x\ y = (set\ x = set\ y)$

lemma *set-perm*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow set\ x = set\ y \Longrightarrow perm\ x\ y$

lemma *distinct-perm-switch*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow perm\ (x \oplus y)\ (y \oplus x)$
lemma *listinter-diff*: $(x \otimes y) \ominus z = (x \ominus z) \otimes (y \ominus z)$

lemma *set-listinter*: $set\ y = set\ z \Longrightarrow x \otimes y = x \otimes z$

lemma *AAA-c*: $a \notin set\ x \Longrightarrow x \ominus [a] = x$

lemma *distinct-perm-cons*: $distinct\ x \Longrightarrow perm\ (a\ \#\ y)\ x \Longrightarrow perm\ y\ (x \ominus [a])$

lemma *listinter-empty*[*simp*]: $y \otimes [] = []$

lemma *subsetset-inter*: $set\ x \subseteq set\ y \Longrightarrow (x \otimes y) = x$

lemma *addvars-addsame*: $x \oplus y \oplus (x \ominus z) = x \oplus y$

lemma *ZZZ*: $x \ominus x \oplus y = []$

lemma *perm-dist-mem*: $distinct\ x \Longrightarrow a \in set\ x \Longrightarrow perm\ (a\ \#\ (x \ominus [a]))\ x$

lemma *addvars-diff*: $b\ \#\ (x \oplus (z \ominus [b])) = (b\ \#\ x) \oplus z$

lemma *perm-cons*: $a \in set\ y \Longrightarrow distinct\ y \Longrightarrow perm\ x\ (y \ominus [a]) \Longrightarrow perm\ (a\ \#\ x)\ y$

end

2 Translation of Hierarchical Block Diagrams

3 Abstract Algebra of Hierarchical Block Diagrams (except one axiom for feedback)

theory *HBDAlgebra* **imports** *ListProp*
begin

locale *BaseOperationFeedbackless* =

fixes *TI TO* :: *'a* ⇒ *'tp list*

fixes *ID* :: *'tp list* ⇒ *'a*
assumes [*simp*]: *TI*(*ID ts*) = *ts*
assumes [*simp*]: *TO*(*ID ts*) = *ts*

fixes *comp* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** *oo 70*)
assumes *TI-comp*[*simp*]: *TI S'* = *TO S* ⟹ *TI* (*S oo S'*) = *TI S*
assumes *TO-comp*[*simp*]: *TI S'* = *TO S* ⟹ *TO* (*S oo S'*) = *TO S'*
assumes *comp-id-left* [*simp*]: *ID* (*TI S*) *oo S* = *S*
assumes *comp-id-right* [*simp*]: *S oo ID* (*TO S*) = *S*
assumes *comp-assoc*: *TI T* = *TO S* ⟹ *TI R* = *TO T* ⟹ *S oo T oo R* = *S oo* (*T oo R*)

fixes *parallel* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ‖ *80*)
assumes *TI-par* [*simp*]: *TI* (*S* ‖ *T*) = *TI S* @ *TI T*
assumes *TO-par* [*simp*]: *TO* (*S* ‖ *T*) = *TO S* @ *TO T*
assumes *par-assoc*: *A* ‖ *B* ‖ *C* = *A* ‖ (*B* ‖ *C*)
assumes *empty-par*[*simp*]: *ID* [] ‖ *S* = *S*
assumes *par-empty*[*simp*]: *S* ‖ *ID* [] = *S*
assumes *parallel-ID* [*simp*]: *ID ts* ‖ *ID ts'* = *ID* (*ts* @ *ts'*)

assumes *comp-parallel-distrib*: *TO S* = *TI S'* ⟹ *TO T* = *TI T'* ⟹ (*S* ‖ *T*) *oo* (*S'* ‖ *T'*) = (*S oo S'*) ‖ (*T oo T'*)

fixes *Split* :: *'tp list* ⇒ *'a*
fixes *Sink* :: *'tp list* ⇒ *'a*
fixes *Switch* :: *'tp list* ⇒ *'tp list* ⇒ *'a*

assumes *TI-Split*[*simp*]: *TI* (*Split ts*) = *ts*
assumes *TO-Split*[*simp*]: *TO* (*Split ts*) = *ts* @ *ts*

assumes *TI-Sink*[*simp*]: *TI* (*Sink ts*) = *ts*

assumes *TO-Sink*[*simp*]: $TO\ (Sink\ ts) = []$

assumes *TI-Switch*[*simp*]: $TI\ (Switch\ ts\ ts') = ts\ @\ ts'$
assumes *TO-Switch*[*simp*]: $TO\ (Switch\ ts\ ts') = ts'\ @\ ts$

assumes *Split-Sink-id*[*simp*]: $Split\ ts\ oo\ Sink\ ts \parallel ID\ ts = ID\ ts$

assumes *Split-Switch*[*simp*]: $Split\ ts\ oo\ Switch\ ts\ ts = Split\ ts$
assumes *Split-assoc*: $Split\ ts\ oo\ ID\ ts \parallel Split\ ts = Split\ ts\ oo\ Split\ ts \parallel ID\ ts$

assumes *Switch-append*: $Switch\ ts\ (ts'\ @\ ts'') = Switch\ ts\ ts' \parallel ID\ ts''\ oo\ ID\ ts' \parallel Switch\ ts\ ts''$
assumes *Sink-append*: $Sink\ ts \parallel Sink\ ts' = Sink\ (ts\ @\ ts')$
assumes *Split-append*: $Split\ (ts\ @\ ts') = Split\ ts \parallel Split\ ts'\ oo\ ID\ ts \parallel Switch\ ts\ ts' \parallel ID\ ts'$

assumes *switch-par-no-vars*: $TI\ A = ti \Longrightarrow TO\ A = to \Longrightarrow TI\ B = ti' \Longrightarrow TO\ B = to' \Longrightarrow Switch\ ti\ ti'\ oo\ B \parallel A\ oo\ Switch\ to'\ to = A \parallel B$

fixes *fb* :: $'a \Rightarrow 'a$
assumes *TI-fb*: $TI\ S = t\ \#\ ts \Longrightarrow TO\ S = t\ \#\ ts' \Longrightarrow TI\ (fb\ S) = ts$
assumes *TO-fb*: $TI\ S = t\ \#\ ts \Longrightarrow TO\ S = t\ \#\ ts' \Longrightarrow TO\ (fb\ S) = ts'$
assumes *fb-comp*: $TI\ S = t\ \#\ TO\ A \Longrightarrow TO\ S = t\ \#\ TI\ B \Longrightarrow fb\ (ID\ [t] \parallel A\ oo\ S\ oo\ ID\ [t] \parallel B) = A\ oo\ fb\ S\ oo\ B$
assumes *fb-par-indep*: $TI\ S = t\ \#\ ts \Longrightarrow TO\ S = t\ \#\ ts' \Longrightarrow fb\ (S \parallel T) = fb\ S \parallel T$

assumes *fb-switch*: $fb\ (Switch\ [t]\ [t]) = ID\ [t]$

begin
definition *fbtype* $S\ tsa\ ts\ ts' = (TI\ S = tsa\ @\ ts \wedge TO\ S = tsa\ @\ ts')$

lemma *fb-comp-fbtype*: $fbtype\ S\ [t]\ (TO\ A)\ (TI\ B)$
$\Longrightarrow fb\ ((ID\ [t] \parallel A)\ oo\ S\ oo\ (ID\ [t] \parallel B)) = A\ oo\ fb\ S\ oo\ B$

lemma *fb-serial-no-vars*: $TO\ A = t\ \#\ ts \Longrightarrow TI\ B = t\ \#\ ts$
$\Longrightarrow fb\ (\ ID\ [t] \parallel A\ oo\ Switch\ [t]\ [t] \parallel ID\ ts\ oo\ ID\ [t] \parallel B) = A\ oo\ B$

lemma *TI-fb-fbtype*: $fbtype\ S\ [t]\ ts\ ts' \Longrightarrow TI\ (fb\ S) = ts$

lemma *TO-fb-fbtype*: $fbtype\ S\ [t]\ ts\ ts' \Longrightarrow TO\ (fb\ S) = ts'$

lemma *fb-par-indep-fbtype*: $fbtype\ S\ [t]\ ts\ ts' \Longrightarrow fb\ (S \parallel T) = fb\ S \parallel T$

lemma *comp-id-left-simp* [*simp*]: $TI\ S = ts \Longrightarrow ID\ ts\ oo\ S\ = S$

lemma *comp-id-right-simp* [*simp*]: $TO\ S = ts \Longrightarrow S\ oo\ ID\ ts = S$

lemma *par-Sink-comp*: $TI\ A = TO\ B \Longrightarrow B \parallel Sink\ t\ oo\ A = (B\ oo\ A) \parallel Sink\ t$

lemma *Sink-par-comp*: $TI\ A = TO\ B \Longrightarrow Sink\ t \parallel B\ oo\ A = Sink\ t \parallel (B\ oo\ A)$

lemma *Split-Sink-par*[*simp*]: $TI\ A = ts \Longrightarrow Split\ ts\ oo\ Sink\ ts \parallel A = A$

lemma *Switch-Switch-ID*[*simp*]: $Switch\ ts\ ts'\ oo\ Switch\ ts'\ ts = ID\ (ts\ @\ ts')$

lemma *Switch-parallel*: $TI\ A = ts' \Longrightarrow TI\ B = ts \Longrightarrow Switch\ ts\ ts'\ oo\ A \parallel B = B \parallel A\ oo\ Switch\ (TO\ B)\ (TO\ A)$

lemma *Switch-type-empty*[*simp*]: $Switch\ ts\ [] = ID\ ts$

lemma *Switch-empty-type*[*simp*]: $Switch\ []\ ts = ID\ ts$

lemma *Split-id-Sink*[*simp*]: $Split\ ts\ oo\ ID\ ts \parallel Sink\ ts = ID\ ts$

lemma *Split-par-Sink*[*simp*]: $TI\ A = ts \Longrightarrow Split\ ts\ oo\ A \parallel Sink\ ts = A$

lemma *Split-empty* [*simp*]: $Split\ [] = ID\ []$

lemma *Sink-empty*[*simp*]: $Sink\ [] = ID\ []$

lemma *Switch-Split*: $Switch\ ts\ ts' = Split\ (ts\ @\ ts')\ oo\ Sink\ ts \parallel ID\ ts' \parallel ID\ ts \parallel Sink\ ts'$

lemma *Sink-cons*: $Sink\ (t\ \#\ ts) = Sink\ [t] \parallel Sink\ ts$

lemma *Split-cons*: $Split\ (t\ \#\ ts) = Split\ [t] \parallel Split\ ts\ oo\ ID\ [t] \parallel Switch\ [t]\ ts \parallel ID\ ts$

lemma *Split-assoc-comp*: $TI\ A = ts \Longrightarrow TI\ B = ts \Longrightarrow TI\ C = ts \Longrightarrow Split\ ts\ oo\ A \parallel (Split\ ts\ oo\ B \parallel C) = Split\ ts\ oo\ (Split\ ts\ oo\ A \parallel B) \parallel C$

lemma *Split-Split-Switch*: $Split\ ts\ oo\ Split\ ts \parallel Split\ ts\ oo\ ID\ ts \parallel Switch\ ts\ ts \parallel ID\ ts = Split\ ts\ oo\ Split\ ts \parallel Split\ ts$

lemma *parallel-empty-commute*: $TI\ A = [] \Longrightarrow TO\ B = [] \Longrightarrow A \parallel B = B \parallel A$

lemma *comp-assoc-middle-ext*: *TI S2 = TO S1 ⟹ TI S3 = TO S2 ⟹ TI S4 = TO S3 ⟹ TI S5 = TO S4 ⟹*
S1 oo (S2 oo S3 oo S4) oo S5 = (S1 oo S2) oo S3 oo (S4 oo S5)

lemma *fb-gen-parallel*: *⋀ S . fbtype S tsa ts ts' ⟹ (fb ^^(length tsa)) (S ‖ T) = ((fb ^^(length tsa)) (S)) ‖ T*

lemmas *parallel-ID-sym = parallel-ID [THEN sym]*
declare *parallel-ID [simp del]*

lemma *fb-indep*: *⋀S. fbtype S tsa (TO A) (TI B) ⟹ (fb ^^(length tsa)) ((ID tsa ‖ A) oo S oo (ID tsa ‖ B)) = A oo (fb ^^(length tsa)) S oo B*

lemma *fb-indep-a*: *⋀S. fbtype S tsa (TO A) (TI B) ⟹ length tsa = n ⟹ (fb ^^ n) ((ID tsa ‖ A) oo S oo (ID tsa ‖ B)) = A oo (fb ^^ n) S oo B*

lemma *fb-comp-right*: *fbtype S [t] ts (TI B) ⟹ fb (S oo (ID [t] ‖ B)) = fb S oo B*

lemma *fb-comp-left*: *fbtype S [t] (TO A) ts ⟹ fb ((ID [t] ‖ A) oo S) = A oo fb S*

lemma *fb-indep-right*: *⋀S. fbtype S tsa ts (TI B) ⟹ (fb ^^(length tsa)) (S oo (ID tsa ‖ B)) = (fb ^^(length tsa)) S oo B*

lemma *fb-indep-left*: *⋀S. fbtype S tsa (TO A) ts ⟹ (fb ^^(length tsa)) ((ID tsa ‖ A) oo S) = A oo (fb ^^(length tsa)) S*

lemma *TI-fb-fbtype-n*: *⋀S. fbtype S t ts ts' ⟹ TI ((fb ^^(length t)) S) = ts*
and *TO-fb-fbtype-n*: *⋀S. fbtype S t ts ts' ⟹ TO ((fb ^^(length t)) S) = ts'*

declare *parallel-ID [simp]*
end

locale *BaseOperationFeedbacklessVars = BaseOperationFeedbackless +*
fixes *TV :: 'var ⇒ 'b*
fixes *newvar :: 'var list ⇒ 'b ⇒ 'var*
assumes *newvar-type[simp]*: *TV(newvar x t) = t*
assumes *newvar-distinct [simp]*: *newvar x t ∉ set x*
assumes *ID [TV a] = ID [TV a]*
begin
primrec *TVs::'var list ⇒ 'b list* **where**
TVs [] = [] |
TVs (a # x) = TV a # TVs x

lemma *TVs-append*: $TVs\ (x\ @\ y) = TVs\ x\ @\ TVs\ y$

definition $Arb\ t = fb\ (Split\ [t])$

lemma *TI-Arb*[*simp*]: $TI\ (Arb\ t) = []$

lemma *TO-Arb*[*simp*]: $TO\ (Arb\ t) = [t]$

fun *set-var*:: $'var\ list \Rightarrow 'var \Rightarrow 'a$ **where**
$set\text{-}var\ []\ b = Arb\ (TV\ b)\ |$
$set\text{-}var\ (a\ \#\ x)\ b = (if\ a = b\ then\ ID\ [TV\ a] \parallel Sink\ (TVs\ x)\ else\ Sink\ [TV\ a] \parallel set\text{-}var\ x\ b)$

lemma *TO-set-var*[*simp*]: $TO\ (set\text{-}var\ x\ a) = [TV\ a]$

lemma *TI-set-var*[*simp*]: $TI\ (set\text{-}var\ x\ a) = TVs\ x$

primrec *switch* :: $'var\ list \Rightarrow 'var\ list \Rightarrow 'a\ ([\text{-} \rightsquigarrow \text{-}])$ **where**
$[x \rightsquigarrow []] = Sink\ (TVs\ x)\ |$
$[x \rightsquigarrow a\ \#\ y] = Split\ (TVs\ x)\ oo\ set\text{-}var\ x\ a \parallel [x \rightsquigarrow y]$

lemma *TI-switch*[*simp*]: $TI\ [x \rightsquigarrow y] = TVs\ x$

lemma *TO-switch*[*simp*]: $TO\ [x \rightsquigarrow y] = TVs\ y$

lemma *switch-not-in-Sink*: $a \notin set\ y \Longrightarrow [a\ \#\ x \rightsquigarrow y] = Sink\ [TV\ a] \parallel [x \rightsquigarrow y]$

lemma *distinct-id*: $distinct\ x \Longrightarrow [x \rightsquigarrow x] = ID\ (TVs\ x)$

lemma *set-var-nin*: $a \notin set\ x \Longrightarrow set\text{-}var\ (x\ @\ y)\ a = Sink\ (TVs\ x) \parallel set\text{-}var\ y\ a$

lemma *set-var-in*: $a \in set\ x \Longrightarrow set\text{-}var\ (x\ @\ y)\ a = set\text{-}var\ x\ a \parallel Sink\ (TVs\ y)$

lemma *set-var-not-in*: $a \notin set\ y \Longrightarrow set\text{-}var\ y\ a = Arb\ (TV\ a) \parallel Sink\ (TVs\ y)$

lemma *set-var-in-a*: $a \notin set\ y \Longrightarrow set\text{-}var\ (x\ @\ y)\ a = set\text{-}var\ x\ a \parallel Sink\ (TVs\ y)$

lemma *switch-append*: $[x \rightsquigarrow y\ @\ z] = Split\ (TVs\ x)\ oo\ [x \rightsquigarrow y] \parallel [x \rightsquigarrow z]$

lemma *switch-nin-a-new*: $set\ x \cap set\ y' = \{\} \Longrightarrow [x\ @\ y \rightsquigarrow y'] = Sink\ (TVs\ x) \parallel [y \rightsquigarrow y']$

lemma *switch-nin-b-new*: $set\ y \cap set\ z = \{\} \Longrightarrow [x\ @\ y \rightsquigarrow z] = [x \rightsquigarrow z] \parallel Sink\ (TVs\ y)$

lemma *var-switch*: $distinct\ (x\ @\ y) \Longrightarrow [x\ @\ y \rightsquigarrow y\ @\ x] = Switch\ (TVs\ x)\ (TVs\ y)$

lemma *switch-par*: $distinct\ (x\ @\ y) \Longrightarrow distinct\ (u\ @\ v) \Longrightarrow TI\ S = TVs\ x \Longrightarrow TI\ T = TVs\ y \Longrightarrow TO\ S = TVs\ v \Longrightarrow TO\ T = TVs\ u \Longrightarrow$
$S \parallel T = [x\ @\ y \rightsquigarrow y\ @\ x]\ oo\ T \parallel S\ oo\ [u\ @\ v \rightsquigarrow v\ @\ u]$

lemma *par-switch*: $distinct\ (x\ @\ y) \Longrightarrow set\ x' \subseteq set\ x \Longrightarrow set\ y' \subseteq set\ y \Longrightarrow [x \rightsquigarrow x'] \parallel [y \rightsquigarrow y'] = [x\ @\ y \rightsquigarrow x'\ @\ y']$

lemma *set-var-sink*[*simp*]: $a \in set\ x \Longrightarrow (TV\ a) = t \Longrightarrow set\text{-}var\ x\ a\ oo\ Sink\ [t] = Sink\ (TVs\ x)$

lemma *switch-Sink*[*simp*]: $\bigwedge ts\ .\ set\ u \subseteq set\ x \Longrightarrow TVs\ u = ts \Longrightarrow [x \rightsquigarrow u]\ oo\ Sink\ ts = Sink\ (TVs\ x)$

lemma *set-var-dup*: $a \in set\ x \Longrightarrow TV\ a = t \Longrightarrow set\text{-}var\ x\ a\ oo\ Split\ [t] = Split\ (TVs\ x)\ oo\ set\text{-}var\ x\ a\ \parallel set\text{-}var\ x\ a$

lemma *switch-dup*: $\bigwedge ts\ .\ set\ y \subseteq set\ x \Longrightarrow TVs\ y = ts \Longrightarrow [x \rightsquigarrow y]\ oo\ Split\ ts = Split\ (TVs\ x)\ oo\ [x \rightsquigarrow y] \parallel [x \rightsquigarrow y]$

lemma *TVs-length-eq*: $\bigwedge y\ .\ TVs\ x = TVs\ y \Longrightarrow length\ x = length\ y$

lemma *set-var-comp-subst*: $\bigwedge y\ .\ set\ u \subseteq set\ x \Longrightarrow TVs\ u = TVs\ y \Longrightarrow a \in set\ y \Longrightarrow [x \rightsquigarrow u]\ oo\ set\text{-}var\ y\ a = set\text{-}var\ x\ (subst\ y\ u\ a)$

lemma *switch-comp-subst*: $set\ u \subseteq set\ x \Longrightarrow set\ v \subseteq set\ y \Longrightarrow TVs\ u = TVs\ y \Longrightarrow [x \rightsquigarrow u]\ oo\ [y \rightsquigarrow v] = [x \rightsquigarrow Subst\ y\ u\ v]$

declare *switch.simps* [*simp del*]

lemma *sw-hd-var*: $distinct\ (a\ \#\ b\ \#\ x) \Longrightarrow [a\ \#\ b\ \#\ x \rightsquigarrow b\ \#\ a\ \#\ x] = Switch\ [TV\ a]\ [TV\ b] \parallel ID\ (TVs\ x)$

lemma *fb-serial*: $distinct\ (a\ \#\ b\ \#\ x) \Longrightarrow TV\ a = TV\ b \Longrightarrow TO\ A = TVs\ (b\ \#\ x) \Longrightarrow TI\ B = TVs\ (a\ \#\ x) \Longrightarrow fb\ (([[a] \rightsquigarrow [a]] \parallel A)\ oo\ [a\ \#\ b\ \#\ x \rightsquigarrow b\ \#\ a\ \#\ x]\ oo\ ([[b] \rightsquigarrow [b]] \parallel B)) = A\ oo\ B$

lemma *Switch-Split*: $distinct\ x \Longrightarrow [x \rightsquigarrow x\ @\ x] = Split\ (TVs\ x)$

lemma *switch-comp*: $distinct\ x \Longrightarrow perm\ x\ y \Longrightarrow set\ z \subseteq set\ y \Longrightarrow [x \rightsquigarrow y]\ oo$

$[y \rightsquigarrow z] = [x \rightsquigarrow z]$

lemma *switch-comp-a*: *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *set y* $\subseteq$ *set x* $\Longrightarrow$ *set z* $\subseteq$ *set y* $\Longrightarrow$ $[x \rightsquigarrow y]$ *oo* $[y \rightsquigarrow z] = [x \rightsquigarrow z]$

primrec *newvars*::*'var list* $\Rightarrow$ *'b list* $\Rightarrow$ *'var list* **where**
newvars x [] = [] |
newvars x (*t* # *ts*) = (*let y* = *newvars x ts in newvar* (*y*@*x*) *t* # *y*)

lemma *newvars-type*[*simp*]: *TVs*(*newvars x ts*) = *ts*

lemma *newvars-distinct*[*simp*]: *distinct* (*newvars x ts*)

lemma *newvars-old-distinct*[*simp*]: *set* (*newvars x ts*) $\cap$ *set x* = {}

lemma *newvars-old-distinct-a*[*simp*]: *set x* $\cap$ *set* (*newvars x ts*) = {}

lemma *newvars-length*: *length*(*newvars x ts*) = *length ts*

lemma *TV-subst*[*simp*]: $\bigwedge$ *y* . *TVs x* = *TVs y* $\Longrightarrow$ *TV* (*subst x y a*) = *TV a*

lemma *TV-Subst*[*simp*]: *TVs x* = *TVs y* $\Longrightarrow$ *TVs* (*Subst x y z*) = *TVs z*

lemma *Subst-cons*: *distinct x* $\Longrightarrow$ *a* $\notin$ *set x* $\Longrightarrow$ *b* $\notin$ *set x* $\Longrightarrow$ *length x* = *length y*
$\Longrightarrow$ *Subst* (*a* # *x*) (*b* # *y*) *z* = *Subst x y* (*Subst* [*a*] [*b*] *z*)

declare *TVs-append* [*simp*]
declare *distinct-id* [*simp*]

lemma *par-empty-right*: *A* $\parallel$ [[] $\rightsquigarrow$ []] = *A*

lemma *par-empty-left*: [[] $\rightsquigarrow$ []] $\parallel$ *A* = *A*
lemma *distinct-vars-comp*: *distinct x* $\Longrightarrow$ *perm x y* $\Longrightarrow$ $[x \rightsquigarrow y]$ *oo* $[y \rightsquigarrow x]$ = *ID* (*TVs x*)

lemma *comp-switch-id*[*simp*]: *distinct x* $\Longrightarrow$ *TO S* = *TVs x* $\Longrightarrow$ *S oo* $[x \rightsquigarrow x]$ = *S*

lemma *comp-id-switch*[*simp*]: *distinct x* $\Longrightarrow$ *TI S* = *TVs x* $\Longrightarrow$ $[x \rightsquigarrow x]$ *oo S* = *S*

lemma *distinct-Subst-a*: $\bigwedge$ *v* . *a* $\neq$ *aa* $\Longrightarrow$ *a* $\notin$ *set v* $\Longrightarrow$ *aa* $\notin$ *set v* $\Longrightarrow$ *distinct v* $\Longrightarrow$ *length u* = *length v* $\Longrightarrow$ *subst u v a* $\neq$ *subst u v aa*

lemma *distinct-Subst-b*: $\bigwedge$ *v* . *a* $\notin$ *set x* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *a* $\notin$ *set v* $\Longrightarrow$ *distinct v* $\Longrightarrow$ *set v* $\cap$ *set x* = {} $\Longrightarrow$ *length u* = *length v* $\Longrightarrow$ *subst u v a* $\notin$ *set* (*Subst u v x*)

lemma *distinct-Subst*: *distinct u* $\Longrightarrow$ *distinct* (*v* @ *x*) $\Longrightarrow$ *length u* = *length v* $\Longrightarrow$ *distinct* (*Subst u v x*)

lemma *Subst-switch-more-general*: *distinct u* $\Longrightarrow$ *distinct* (*v* @ *x*) $\Longrightarrow$ *set y* $\subseteq$ *set x*
$\Longrightarrow$ *TVs u* = *TVs v* $\Longrightarrow$ [*x* $\rightsquigarrow$ *y*] = [*Subst u v x* $\rightsquigarrow$ *Subst u v y*]

lemma *id-par-comp*: *distinct x* $\Longrightarrow$ *TO A* = *TI B* $\Longrightarrow$ [*x* $\rightsquigarrow$ *x*] $\parallel$ (*A oo B*) = ([*x* $\rightsquigarrow$ *x*] $\parallel$ *A*) *oo* ([*x* $\rightsquigarrow$ *x*] $\parallel$ *B*)

lemma *par-id-comp*: *distinct x* $\Longrightarrow$ *TO A* = *TI B* $\Longrightarrow$ (*A oo B*) $\parallel$ [*x* $\rightsquigarrow$ *x*] = (*A* $\parallel$ [*x* $\rightsquigarrow$ *x*]) *oo* (*B* $\parallel$ [*x* $\rightsquigarrow$ *x*])

lemma *switch-parallel-a*: *distinct* (*x* @ *y*) $\Longrightarrow$ *distinct* (*u* @ *v*) $\Longrightarrow$ *TI S* = *TVs x* $\Longrightarrow$ *TI T* = *TVs y* $\Longrightarrow$ *TO S* = *TVs u* $\Longrightarrow$ *TO T* = *TVs v* $\Longrightarrow$
S $\parallel$ *T oo* [*u*@*v* $\rightsquigarrow$ *v*@*u*] = [*x*@*y*$\rightsquigarrow$*y*@*x*] *oo T* $\parallel$ *S*

declare *distinct-id* [*simp del*]

lemma *fb-gen-serial*: $\bigwedge$*A B v x* . *distinct* (*u* @ *v* @ *x*) $\Longrightarrow$ *TO A* = *TVs* (*v*@*x*) $\Longrightarrow$ *TI B* = *TVs* (*u* @ *x*) $\Longrightarrow$ *TVs u* = *TVs v*
$\Longrightarrow$ (*fb* ^^ *length u*) (([*u* $\rightsquigarrow$ *u*] $\parallel$ *A*) *oo* [*u* @ *v* @ *x* $\rightsquigarrow$ *v* @ *u* @ *x*] *oo* ([*v* $\rightsquigarrow$ *v*] $\parallel$ *B*)) = *A oo B*

lemma *fb-par-serial*: *distinct*(*u* @ *x* @ *x*′) $\Longrightarrow$ *distinct* (*u* @ *y* @ *x*′) $\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs* (*u*@*y*) $\Longrightarrow$ *TI B* = *TVs* (*u*@*x*′) $\Longrightarrow$ *TO B* = *TVs y*′ $\Longrightarrow$
(*fb*^^(*length u*)) ([*u* @ *x* @ *x*′ $\rightsquigarrow$ *x* @ *u* @ *x*′] *oo* (*A* $\parallel$ *B*)) = (*A* $\parallel$ *ID* (*TVs x*′) *oo* [*u* @ *y* @ *x*′ $\rightsquigarrow$ *y* @ *u* @ *x*′] *oo ID* (*TVs y*) $\parallel$ *B*)

lemma *switch-newvars*: *distinct x* $\Longrightarrow$ [*newvars w* (*TVs x*) $\rightsquigarrow$ *newvars w* (*TVs x*)] = [*x* $\rightsquigarrow$ *x*]

lemma *switch-par-comp-Subst*: *distinct x* $\Longrightarrow$ *distinct y*′ $\Longrightarrow$ *distinct z*′ $\Longrightarrow$ *set y* $\subseteq$ *set x*
$\Longrightarrow$ *set z* $\subseteq$ *set x*
$\Longrightarrow$ *set u* $\subseteq$ *set y*′ $\Longrightarrow$ *set v* $\subseteq$ *set z*′ $\Longrightarrow$ *TVs y* = *TVs y*′ $\Longrightarrow$ *TVs z* = *TVs z*′ $\Longrightarrow$
[*x* $\rightsquigarrow$ *y* @ *z*] *oo* [*y*′ $\rightsquigarrow$ *u*] $\parallel$ [*z*′ $\rightsquigarrow$ *v*] = [*x* $\rightsquigarrow$ *Subst y*′ *y u* @ *Subst z*′ *z v*]

lemma *switch-par-comp*: *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *distinct z* $\Longrightarrow$ *set y* $\subseteq$

set x $\Longrightarrow$ *set z* $\subseteq$ *set x*
$\Longrightarrow$ *set y'* $\subseteq$ *set y* $\Longrightarrow$ *set z'* $\subseteq$ *set z* $\Longrightarrow$ $[x \rightsquigarrow y \mathbin{@} z]$ *oo* $[y \rightsquigarrow y']$ $\parallel$ $[z \rightsquigarrow z'] = [x \rightsquigarrow y' \mathbin{@} z']$

lemma *par-switch-eq*: *distinct u* $\Longrightarrow$ *distinct v* $\Longrightarrow$ *distinct y'* $\Longrightarrow$ *distinct z'*

$\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs v* $\Longrightarrow$ *TI C* = *TVs v* @ *TVs y* $\Longrightarrow$ *TVs y* = *TVs y'* $\Longrightarrow$
TI C' = *TVs v* @ *TVs z* $\Longrightarrow$ *TVs z* = *TVs z'* $\Longrightarrow$
set x $\subseteq$ *set u* $\Longrightarrow$ *set y* $\subseteq$ *set u* $\Longrightarrow$ *set z* $\subseteq$ *set u* $\Longrightarrow$
$[v \rightsquigarrow v]$ $\parallel$ $[u \rightsquigarrow y]$ *oo C* = $[v \rightsquigarrow v]$ $\parallel$ $[u \rightsquigarrow z]$ *oo C'*
$\Longrightarrow$ $[u \rightsquigarrow x \mathbin{@} y]$ *oo* (*A* $\parallel$ $[y' \rightsquigarrow y']$) *oo C* = $[u \rightsquigarrow x \mathbin{@} z]$ *oo* (*A* $\parallel$ $[z' \rightsquigarrow z']$) *oo C'*

lemma *paralle-switch*: $\exists$ *x y u v*. *distinct* (*x* @ *y*) $\wedge$ *distinct* (*u* @ *v*) $\wedge$ *TVs x* = *TI A*
$\wedge$ *TVs u* = *TO A* $\wedge$ *TVs y* = *TI B* $\wedge$
TVs v = *TO B* $\wedge$ *A* $\parallel$ *B* = $[x \mathbin{@} y \rightsquigarrow y \mathbin{@} x]$ *oo* (*B* $\parallel$ *A*) *oo* $[v \mathbin{@} u \rightsquigarrow u \mathbin{@} v]$

lemma *par-switch-eq-dist*: *distinct* (*u* @ *v*) $\Longrightarrow$ *distinct y'* $\Longrightarrow$ *distinct z'* $\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs v* $\Longrightarrow$ *TI C* = *TVs v* @ *TVs y* $\Longrightarrow$ *TVs y* = *TVs y'* $\Longrightarrow$
TI C' = *TVs v* @ *TVs z* $\Longrightarrow$ *TVs z* = *TVs z'* $\Longrightarrow$
set x $\subseteq$ *set u* $\Longrightarrow$ *set y* $\subseteq$ *set u* $\Longrightarrow$ *set z* $\subseteq$ *set u* $\Longrightarrow$
$[v \mathbin{@} u \rightsquigarrow v \mathbin{@} y]$ *oo C* = $[v \mathbin{@} u \rightsquigarrow v \mathbin{@} z]$ *oo C'* $\Longrightarrow$ $[u \rightsquigarrow x \mathbin{@} y]$ *oo* (*A* $\parallel$ $[y' \rightsquigarrow y']$) *oo C* = $[u \rightsquigarrow x \mathbin{@} z]$ *oo* (*A* $\parallel$ $[z' \rightsquigarrow z']$) *oo C'*

lemma *par-switch-eq-dist-a*: *distinct* (*u* @ *v*) $\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs v* $\Longrightarrow$ *TI C* = *TVs v* @ *TVs y* $\Longrightarrow$ *TVs y* = *ty* $\Longrightarrow$ *TVs z* = *tz* $\Longrightarrow$
TI C' = *TVs v* @ *TVs z* $\Longrightarrow$ *set x* $\subseteq$ *set u* $\Longrightarrow$ *set y* $\subseteq$ *set u* $\Longrightarrow$ *set z* $\subseteq$ *set u* $\Longrightarrow$
$[v \mathbin{@} u \rightsquigarrow v \mathbin{@} y]$ *oo C* = $[v \mathbin{@} u \rightsquigarrow v \mathbin{@} z]$ *oo C'* $\Longrightarrow$ $[u \rightsquigarrow x \mathbin{@} y]$ *oo A* $\parallel$ *ID ty oo C* = $[u \rightsquigarrow x \mathbin{@} z]$ *oo A* $\parallel$ *ID tz oo C'*

lemma *par-switch-eq-a*: *distinct* (*u* @ *v*) $\Longrightarrow$ *distinct y'* $\Longrightarrow$ *distinct z'* $\Longrightarrow$ *distinct t'* $\Longrightarrow$ *distinct s'*
$\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs v* $\Longrightarrow$ *TI C* = *TVs t* @ *TVs v* @ *TVs y* $\Longrightarrow$ *TVs y* = *TVs y'* $\Longrightarrow$
TI C' = *TVs s* @ *TVs v* @ *TVs z* $\Longrightarrow$ *TVs z* = *TVs z'* $\Longrightarrow$ *TVs t* = *TVs t'* $\Longrightarrow$ *TVs s* = *TVs s'* $\Longrightarrow$
set t $\subseteq$ *set u* $\Longrightarrow$ *set x* $\subseteq$ *set u* $\Longrightarrow$ *set y* $\subseteq$ *set u* $\Longrightarrow$ *set s* $\subseteq$ *set u* $\Longrightarrow$ *set z* $\subseteq$ *set u* $\Longrightarrow$
$[u \mathbin{@} v \rightsquigarrow t \mathbin{@} v \mathbin{@} y]$ *oo C* = $[u \mathbin{@} v \rightsquigarrow s \mathbin{@} v \mathbin{@} z]$ *oo C'* $\Longrightarrow$
$[u \rightsquigarrow t \mathbin{@} x \mathbin{@} y]$ *oo* ($[t' \rightsquigarrow t']$ $\parallel$ *A* $\parallel$ $[y' \rightsquigarrow y']$) *oo C* = $[u \rightsquigarrow s \mathbin{@} x \mathbin{@} z]$ *oo* ($[s' \rightsquigarrow s']$ $\parallel$ *A* $\parallel$ $[z' \rightsquigarrow z']$) *oo C'*

lemma *length-TVs*: *length* (*TVs x*) = *length x*

lemma *comp-par*: *distinct x* $\Longrightarrow$ *set y* $\subseteq$ *set x* $\Longrightarrow$ $[x \rightsquigarrow x \mathbin{@} x]$ *oo* $[x \rightsquigarrow y]$ $\parallel$ $[x \rightsquigarrow y] = [x \rightsquigarrow y \mathbin{@} y]$

lemma *Subst-switch-a*: *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *set z* $\subseteq$ *set x* $\Longrightarrow$ *TVs x* = *TVs y* $\Longrightarrow$ $[x \rightsquigarrow z] = [y \rightsquigarrow$ *Subst x y z*$]$

lemma *change-var-names*: *distinct a* $\Longrightarrow$ *distinct b* $\Longrightarrow$ *TVs a* = *TVs b* $\Longrightarrow$ $[a \rightsquigarrow a \mathbin{@} a] = [b \rightsquigarrow b \mathbin{@} b]$

3.1 Deterministic diagrams

definition *deterministic S* = (*Split* (*TI S*) *oo S* $\parallel$ *S* = *S oo Split* (*TO S*))

lemma *deterministic-split*:
assumes *deterministic S*
and *distinct* (*a*#*x*)
and *TO S* = *TVs* (*a* # *x*)
shows *S* = *Split* (*TI S*) *oo* (*S oo* [*a* # *x* $\rightsquigarrow$ [*a*]]) $\parallel$ (*S oo* [*a* # *x* $\rightsquigarrow$ *x*])

lemma *deterministicE*: *deterministic A* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs y*
$\Longrightarrow$ $[x \rightsquigarrow x \mathbin{@} x]$ *oo* (*A* $\parallel$ *A*) = *A oo* $[y \rightsquigarrow y \mathbin{@} y]$

lemma *deterministicI*: *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *TI A* = *TVs x* $\Longrightarrow$ *TO A* = *TVs y* $\Longrightarrow$
$[x \rightsquigarrow x \mathbin{@} x]$ *oo A* $\parallel$ *A* = *A oo* $[y \rightsquigarrow y \mathbin{@} y]$ $\Longrightarrow$ *deterministic A*

lemma *deterministic-switch*: *distinct x* $\Longrightarrow$ *set y* $\subseteq$ *set x* $\Longrightarrow$ *deterministic* $[x \rightsquigarrow y]$

lemma *deterministic-comp*: *deterministic A* $\Longrightarrow$ *deterministic B* $\Longrightarrow$ *TO A* = *TI B* $\Longrightarrow$ *deterministic* (*A oo B*)

lemma *deterministic-par*: *deterministic A* $\Longrightarrow$ *deterministic B* $\Longrightarrow$ *deterministic* (*A* $\parallel$ *B*)

end

end

4 Abstract Algebra of Hierarchical Block Diagrams with All Axioms

theory *ExtendedHBDAlgebra* **imports** *HBDAlgebra*

begin

locale *BaseOperation* = *BaseOperationFeedbackless* +
assumes *fb-twice-switch-no-vars*: *TI S* = *t′* # *t* # *ts* ⟹ *TO S* = *t′* # *t* # *ts′*
⟹ (*fb* ^^ (*2*::*nat*)) (*Switch* [*t*] [*t′*] ‖ *ID ts oo S oo Switch* [*t′*] [*t*] ‖ *ID ts′*) = (*fb* ^^ (*2*:: *nat*)) *S*

locale *BaseOperationVars* = *BaseOperation* + *BaseOperationFeedbacklessVars*
begin
lemma *fb-twice-switch*: *distinct* (*a* # *b* # *x*) ⟹ *distinct* (*a* # *b* # *y*) ⟹ *TI S* = *TVs* (*b* # *a* # *x*) ⟹ *TO S* = *TVs* (*b* # *a* # *y*)
⟹ (*fb* ^^ (*2*::*nat*)) ([*a* # *b* # *x* ⇝ *b* # *a* # *x*] *oo S oo* [*b* # *a* # *y* ⇝ *a* # *b* # *y*]) = (*fb* ^^ (*2*:: *nat*)) *S*

lemma *fb-switch-a*: ⋀ *S* . *distinct* (*a* # *z* @ *x*) ⟹ *distinct* (*a* # *z* @ *y*) ⟹ *TI S* = *TVs* (*z* @ *a* # *x*) ⟹ *TO S* = *TVs* (*z* @ *a* # *y*)
⟹ (*fb* ^^ (*Suc* (*length z*))) ([*a* # *z* @ *x* ⇝ *z* @ *a* # *x*] *oo S oo* [*z* @ *a* # *y* ⇝ *a* # *z* @ *y*]) = (*fb* ^^ (*Suc* (*length z*))) *S*

lemma *swap-power*: (*f* ^^ *n*) ((*f* ^^ *m*) *S*) = (*f* ^^ *m*) ((*f* ^^ *n*) *S*)

lemma *fb-switch-b*: ⋀ *v x y S* . *distinct* (*u* @ *v* @ *x*) ⟹ *distinct* (*u* @ *v* @ *y*) ⟹ *TI S* = *TVs* (*v* @ *u* @ *x*) ⟹ *TO S* = *TVs* (*v* @ *u* @ *y*)
⟹ (*fb* ^^ (*length* (*u* @ *v*))) ([*u* @ *v* @ *x* ⇝ *v* @ *u* @ *x*] *oo S oo* [*v* @ *u* @ *y* ⇝ *u* @ *v* @ *y*]) = (*fb* ^^ (*length* (*u* @ *v*))) *S*

theorem *fb-perm*: ⋀ *v S* . *perm u v* ⟹ *distinct* (*u* @ *x*) ⟹ *distinct* (*u* @ *y*) ⟹ *fbtype S* (*TVs u*) (*TVs x*) (*TVs y*)
⟹ (*fb* ^^ (*length u*)) ([*v* @ *x* ⇝ *u* @ *x*] *oo S oo* [*u* @ *y* ⇝ *v* @ *y*]) = (*fb* ^^ (*length u*)) *S*

end

end

5 Constructive Functions

theory *Constructive* **imports** *Main*
begin

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*)
and *sup* (**infixl** ⊔ *65*)

class *order-bot-max* = *order-bot* +
fixes *maximal* :: $'a \Rightarrow bool$
assumes *maximal-def*: $maximal\ x = (\forall\ y\ .\ \neg\ x < y)$
assumes [*simp*]: $\neg\ maximal\ \bot$
begin
lemma *ex-not-le-bot*[*simp*]: $\exists\ a.\ \neg\ a \leq \bot$
end

instantiation *option* :: (*type*) *order-bot-max*
begin
definition *bot-option-def*: $(\bot::'a\ option) = None$
definition *le-option-def*: $((x::'a\ option) \leq y) = (x = None \vee x = y)$
definition *less-option-def*: $((x::'a\ option) < y) = (x \leq y \wedge \neg\ (y \leq x))$
definition *maximal-option-def*: $maximal\ (x::'a\ option) = (\forall\ y\ .\ \neg\ x < y)$

instance

lemma [*simp*]: $None \leq x$
end

context *order-bot*
begin
definition *is-lfp f x* = $((f\ x = x) \wedge (\forall\ y\ .\ f\ y = y \longrightarrow x \leq y))$
definition *emono f* = $(\forall\ x\ y.\ x \leq y \longrightarrow f\ x \leq f\ y)$

definition *Lfp f* = *Eps* (*is-lfp f*)

lemma *lfp-unique*: *is-lfp f x* $\Longrightarrow$ *is-lfp f y* $\Longrightarrow$ $x = y$

lemma *lfp-exists*: *is-lfp f x* $\Longrightarrow$ *Lfp f* = x

lemma *emono-a*: *emono f* $\Longrightarrow$ $x \leq y \Longrightarrow f\ x \leq f\ y$

lemma *emono-fix*: *emono f* $\Longrightarrow$ $f\ y = y \Longrightarrow (f\ \hat{}\hat{}\ n)\ \bot \leq y$

lemma *emono-is-lfp*: *emono* $(f::'a \Rightarrow 'a) \Longrightarrow (f\ \hat{}\hat{}\ (n + 1))\ \bot = (f\ \hat{}\hat{}\ n)\ \bot$ $\Longrightarrow$ *is-lfp f* $((f\ \hat{}\hat{}\ n)\ \bot)$

lemma *emono-lfp-bot*: *emono* $(f::'a \Rightarrow 'a) \Longrightarrow (f\ \hat{}\hat{}\ (n + 1))\ \bot = (f\ \hat{}\hat{}\ n)$ $\bot \Longrightarrow$ *Lfp f* = $((f\ \hat{}\hat{}\ n)\ \bot)$

lemma *emono-up*: *emono f* $\Longrightarrow (f\ \hat{}\hat{}\ n)\ \bot \leq (f\ \hat{}\hat{}\ (Suc\ n))\ \bot$
end

context *order*
begin
definition *min-set A* = $(SOME\ n\ .\ n \in A \wedge (\forall\ x \in A\ .\ n \leq x))$
end

lemma *min-nonempty-nat-set-aux*: $\forall\ A\ .\ (n::nat) \in A \longrightarrow (\exists\ k \in A\ .\ (\forall\ x \in A\ .\ k \leq x))$

lemma *min-nonempty-nat-set*: $(n::nat) \in A \Longrightarrow (\exists\ k\ .\ k \in A \wedge (\forall\ x \in A\ .\ k \leq x))$

thm *someI-ex*

lemma *min-set-nat-aux*: $(n::nat) \in A \Longrightarrow min\text{-}set\ A \in A \wedge (\forall\ x \in A\ .\ min\text{-}set\ A \leq x)$

lemma $(n::nat) \in A \Longrightarrow min\text{-}set\ A \in A \wedge min\text{-}set\ A \leq n$

lemma *min-set-in*: $(n::nat) \in A \Longrightarrow min\text{-}set\ A \in A$

lemma *min-set-less*: $(n::nat) \in A \Longrightarrow min\text{-}set\ A \leq n$

definition *mono-a* $f = (\forall\ a\ b\ a'\ b'.\ (a::'a::order) \leq a' \wedge (b::'b::order) \leq b' \longrightarrow f\ a\ b \leq f\ a'\ b')$

class *fin-cpo* = *order-bot-max* +

assumes *fin-up-chain*: $(\forall\ i::\ nat\ .\ a\ i \leq a\ (Suc\ i)) \Longrightarrow \exists\ n\ .\ \forall\ i \geq n\ .\ a\ i = a\ n$

begin

lemma *emono-ex-lfp*: $emono\ f \Longrightarrow \exists\ n\ .\ is\text{-}lfp\ f\ ((f\ \hat{}\hat{}\ n)\ \bot)$

lemma *emono-lfp*: $emono\ f \Longrightarrow \exists\ n\ .\ Lfp\ f = (f\ \hat{}\hat{}\ n)\ \bot$

lemma *emono-is-lfp*: $emono\ f \Longrightarrow is\text{-}lfp\ f\ (Lfp\ f)$

definition *lfp-index* $(f::'a \Rightarrow 'a) = min\text{-}set\ \{n\ .\ (f\ \hat{}\hat{}\ n)\ \bot = (f\ \hat{}\hat{}\ (n + 1))\ \bot\}$

lemma *lfp-index-aux*: $emono\ f \Longrightarrow (\forall\ i < (lfp\text{-}index\ f)\ .\ (f\ \hat{}\hat{}\ i)\ \bot < (f\ \hat{}\hat{}\ (i + 1))\ \bot) \wedge (f\ \hat{}\hat{}\ (lfp\text{-}index\ f))\ \bot = (f\ \hat{}\hat{}\ ((lfp\text{-}index\ f) + 1))\ \bot$

lemma [*simp*]: $emono\ f \Longrightarrow i < lfp\text{-}index\ f \Longrightarrow (f\ \hat{}\hat{}\ i)\ \bot < f\ ((f\ \hat{}\hat{}\ i)\ \bot)$

lemma [*simp*]: $emono\ f \Longrightarrow f\ ((f\ \hat{}\hat{}\ (lfp\text{-}index\ f))\ \bot) = (f\ \hat{}\hat{}\ (lfp\text{-}index\ f))\ \bot$

lemma $emono\ f \Longrightarrow Lfp\ f = (f\ \hat{}\hat{}\ lfp\text{-}index\ f)\ \bot$

lemma *AA-aux*: $emono\ f \Longrightarrow (\bigwedge\ b\ .\ b \leq a \Longrightarrow f\ b \leq a) \Longrightarrow (f\ \hat{}\hat{}\ n)\ \bot \leq a$

lemma *AA*: $emono\ f \Longrightarrow (\bigwedge\ b\ .\ b \leq a \Longrightarrow f\ b \leq a) \Longrightarrow Lfp\ f \leq a$

lemma *BB*: $emono\ f \Longrightarrow f\ (Lfp\ f) = Lfp\ f$

lemma *Lfp-mono*: $emono\ f \Longrightarrow emono\ g \Longrightarrow (\bigwedge\ a\ .\ f\ a \leq g\ a) \Longrightarrow Lfp\ f \leq Lfp\ g$

end
declare [[*show-types*]]

lemma [*simp*]: $mono\text{-}a\ f \Longrightarrow emono\ (f\ a)$

lemma [*simp*]: $mono\text{-}a\ f \Longrightarrow emono\ (\lambda\ a\ .\ f\ a\ b)$

lemma *mono-aD*: $mono\text{-}a\ f \Longrightarrow a \leq a' \Longrightarrow b \leq b' \Longrightarrow f\ a\ b \leq f\ a'\ b'$

lemma [*simp*]: $mono\text{-}a\ (f::'a::fin\text{-}cpo \Rightarrow 'b::fin\text{-}cpo \Rightarrow 'b) \Longrightarrow mono\text{-}a\ g \Longrightarrow emono\ (\lambda b.\ f\ (Lfp\ (g\ b))\ b)$

lemma *CCC*: $mono\text{-}a\ \ (f::'a::fin\text{-}cpo \Rightarrow 'b::fin\text{-}cpo \Rightarrow 'b) \Longrightarrow mono\text{-}a\ g \Longrightarrow Lfp\ (\lambda a.\ g\ (Lfp\ (f\ a))\ a) \leq Lfp\ (g\ (Lfp\ (\lambda b.\ f\ (Lfp\ (g\ b))\ b)))$

lemma *Lfp-commute*: $mono\text{-}a\ (f::'a::fin\text{-}cpo \Rightarrow 'b::fin\text{-}cpo \Rightarrow 'b::fin\text{-}cpo) \Longrightarrow mono\text{-}a\ g \Longrightarrow Lfp\ (\lambda\ b\ .\ f\ \ (Lfp\ (\lambda\ a\ .\ (g\ (Lfp\ (f\ a)))\ a))\ b) = Lfp\ (\lambda\ b\ .\ f\ (Lfp\ (g\ b))\ b)$

instantiation *option* :: (*type*) *fin-cpo*
begin
lemma *fin-up-non-bot*: $(\forall\ i\ .\ (a::nat \Rightarrow 'a\ option)\ i \leq a\ (Suc\ i)) \Longrightarrow a\ n \neq \bot \Longrightarrow n \leq i \Longrightarrow a\ i = a\ n$

lemma *fin-up-chain-option*: $(\forall\ i::\ nat\ .\ (a::nat \Rightarrow 'a\ option)\ i \leq a\ (Suc\ i)) \Longrightarrow \exists\ n\ .\ \forall\ i \geq n\ .\ a\ i = a\ n$

instance
end

instantiation *prod* :: (*order-bot-max*, *order-bot-max*) *order-bot-max*
begin
definition *bot-prod-def*: $(\bot :: 'a \times 'b) = (\bot, \bot)$
definition *le-prod-def*: $(x \leq y) = (fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y)$
definition *less-prod-def*: $((x::'a \times 'b) < y) = (x \leq y \wedge \neg\ (y \leq x))$
definition *maximal-prod-def*: $maximal\ (x::'a \times 'b) = (\forall\ y\ .\ \neg\ x < y)$

instance
end

instantiation *prod* :: (*fin-cpo*, *fin-cpo*) *fin-cpo*
begin

lemma *fin-up-chain-prod*: $(\forall\ i::\ nat\ .\ (a::nat \Rightarrow 'a \times 'b)\ i \leq a\ (Suc\ i)) \Longrightarrow \exists\ n\ .\ \forall\ i \geq n\ .\ a\ i = a\ n$
instance
end

end

6 Constructive Functions are a Model of the HBD Algebra

theory *ConsFuncHBDModel* **imports** *ExtendedHBDAlgebra Constructive*
begin

datatype *Types* = *int* | *bool* | *nat*

datatype *Values* = *Inte* (*integer* : *int option*) | *Bool* (*boolean*: *bool option*) | *Nat* (*natural*: *nat option*)

primrec *tv* :: *Values* ⇒ *Types* **where**
tv (*Inte i*) = *int* |
tv (*Bool b*) = *bool* |
tv (*Nat n*) = *nat*

primrec *tp* :: *Values list* ⇒ *Types list* **where**
tp [] = [] |
tp (*a* # *v*) = *tv a* # *tp v*

fun *le-val* :: *Values* ⇒ *Values* ⇒ *bool* **where**
(*le-val* (*Inte v*) (*Inte u*)) = (*v* ≤ *u*) |
(*le-val* (*Bool v*) (*Bool u*)) = (*v* ≤ *u*) |
(*le-val* (*Nat v*) (*Nat u*)) = (*v* ≤ *u*) |
le-val - - = *False*

instantiation *Values* :: *order*
begin
definition *le-Values-def*: ((*v*::*Values*) ≤ *u*) = *le-val v u*
definition *less-Values-def*: ((*v*::*Values*) < *u*) = (*v* ≤ *u* ∧ ¬ *u* ≤ *v*)
instance
end

fun *le-list* :: *'a*::*order list* ⇒ *'a*::*order list* ⇒ *bool* **where**
le-list [] [] = *True* |
le-list (*a* # *x*) (*b* # *y*) = (*a* ≤ *b* ∧ *le-list x y*) |
le-list - - = *False*

instantiation *list* :: (*order*) *order*
begin
definition *le-list-def*: ((*v*::'*a list*) ≤ *u*) = *le-list u v*
definition *less-list-def*: ((*v*::'*a list*) < *u*) = (*v* ≤ *u* ∧ ¬ *u* ≤ *v*)
instance
end

lemma [*simp*]: *mono integer*

lemma [*simp*]: *mono boolean*

lemma [*simp*]: *mono natural*

definition *has-in-type x* = {*f* . (*dom f* = {*v* . *tp v* = *x*})}
definition *has-out-type x* = {*f* . (*image f* (*dom f*) ⊆ *Some* ' {*v* . *tp v* = *x*})}

definition *has-in-out-type x y* = *has-in-type x* ∩ *has-out-type y*

definition *ID-f x v* = (*if tp v* = *x then Some v else None*)

lemma [*simp*]: (*tp x* = []) = (*x* = [])

lemma *map-comp-type*: *f* ∈ *has-in-out-type x y* ⟹ *g* ∈ *has-in-out-type y z* ⟹ *g* ∘$_m$ *f* ∈ *has-in-out-type x z*

definition *TI-f f* = (*SOME x* . (∃ *y* . *f* ∈ *has-in-out-type x y*))

definition *TO-f f* = (*SOME y* . (∃ *x* . *f* ∈ *has-in-out-type x y*))

fun *pref* :: *Values list* ⇒ *Types list* ⇒ *Values list* **where**
pref v [] = [] |
pref (*a* # *v*) (*t* # *x*) = (*if tv a* = *t then a* # *pref v x else undefined*) |
pref v x = *undefined*

fun *suff* :: *Values list* ⇒ *Types list* ⇒ *Values list* **where**
suff v [] = *v* |
suff (*a* # *v*) (*t* # *x*) = (*if tv a* = *t then suff v x else undefined*) |
suff v x = *undefined*

lemma *tp-pref-suff*: ⋀ *x y* . *tp v* = *x* @ *y* ⟹ *tp* (*pref v x*) = *x* ∧ *tp* (*suff v x*) = *y*

definition *par-f f g v* = (*if tp v* = (*TI-f f*) @ (*TI-f g*) *then Some* (*the* (*f* (*pref v* (*TI-f f*))) @ (*the* (*g* (*suff v* (*TI-f f*))))) *else None*)

fun *some-v*:: *Types list* ⇒ *Values list* **where**
some-v [] = [] |

some-v (*int* # *x*) = (*Inte undefined*) # *some-v x* |
some-v (*bool* # *x*) = (*Bool undefined*) # *some-v x* |
some-v (*nat* # *x*) = (*Nat undefined*) # *some-v x*

lemma [*simp*]: *tp* (*some-v x*) = *x*

lemma *same-in-type*: *f* ∈ *has-in-type x* ⟹ *f* ∈ *has-in-type y* ⟹ *x* = *y*

lemma *same-out-type*: *f* ∈ *has-in-type z* ⟹ *f* ∈ *has-out-type x* ⟹ *f* ∈ *has-out-type y* ⟹ *x* = *y*

lemma *type-has-type*:
assumes *A*: *f* ∈ *has-in-out-type x y*
shows *TI-f f* = *x* **and** *TO-f f* = *y*

lemma *has-type-out-type*: *f* ∈ *has-in-out-type x y* ⟹ *tp v* = *x* ⟹ *tp* (*the* (*f v*)) = *y*

lemma *tp-append*: *tp* (*v* @ *u*) = *tp v* @ *tp u*

lemma *par-f-type*: *f* ∈ *has-in-out-type x y* ⟹ *g* ∈ *has-in-out-type x′ y′* ⟹ *par-f f g* ∈ *has-in-out-type* (*x* @ *x′*) (*y* @ *y′*)

definition *Dup-f x v* = (*if tp v* = *x then Some* (*v* @ *v*) *else None*)

lemma *Dup-has-in-out-type*: *Dup-f x* ∈ *has-in-out-type x* (*x* @ *x*)

definition *Sink-f x v* = (*if tp v* = *x then Some* [] *else None*)

lemma *Sink-has-in-out-type*: *Sink-f x* ∈ *has-in-out-type x* []

definition *Switch-f x y v* = (*if tp v* = *x* @ *y then Some* (*suff v x* @ *pref v x*) *else None*)

lemma *Switch-has-in-out-type*: *Switch-f x y* ∈ *has-in-out-type* (*x* @ *y*) (*y* @ *x*)

primrec *fb-t* :: *Types* ⇒ (*Values* ⇒ *Values*) ⇒ *Values* **where**
fb-t int f = *Inte* (*Lfp* (λ *a* . *integer* (*f* (*Inte a*)))) |
fb-t bool f = *Bool* (*Lfp* (λ *a* . *boolean* (*f* (*Bool a*)))) |
fb-t nat f = *Nat* (*Lfp* (λ *a* . *natural* (*f* (*Nat a*))))

definition *fb-f f v* = (*if tp v* = *tl* (*TI-f f*) *then Some* (*tl* (*the* (*f* ((*fb-t* (*hd* (*TI-f f*)) (λ *a* . *hd* (*the* (*f* (*a* # *v*))))) # *v*)))) *else None*)

thm *le-Values-def*
thm *le-val.simps*

lemma [*simp*]: *mono Inte*

lemma [*simp*]: *mono Bool*

lemma [*simp*]: *mono Nat*

thm *monoE*
thm *monoI*
thm *mono-aD*
lemma [*simp*]: *mono A* $\Longrightarrow$ *mono B* $\Longrightarrow$ *mono C* $\Longrightarrow$ *mono-a f* $\Longrightarrow$ *mono-a* (λa *b*. *C* (*f* (*A a*) (*B b*)))

lemma *fb-t-commute*: *mono-a f* $\Longrightarrow$ *mono-a g*
$\Longrightarrow$ *fb-t t* (λ *b* . *f* (*fb-t t'* (λ *a* . (*g* (*fb-t t* (*f a*))) *a*)) *b*) = *fb-t t* (λ *b* . *f* (*fb-t t'* (*g b*)) *b*)

lemma *fb-t-eq-type*: ($\bigwedge$ *a* . *tv a* = *t* $\Longrightarrow$ *f a* = *g a*) $\Longrightarrow$ *fb-t t f* = *fb-t t g*

lemma [*simp*]: *tv* (*fb-t t f*) = *t*

lemma *has-type-type-in*: *f v* = *Some u* $\Longrightarrow$ *f* $\in$ *has-in-out-type x y* $\Longrightarrow$ *tp v* = *x*

lemma *has-type-type-in-a*: *f v* = *None* $\Longrightarrow$ *f* $\in$ *has-in-out-type x y* $\Longrightarrow$ *tp v* $\neq$ *x*

lemma *has-type-defined*: *f* $\in$ *has-in-out-type x y* $\Longrightarrow$ *tp v* = *x* $\Longrightarrow$ $\exists$ *u* . *f v* = *Some u*

lemma *tp-tail*: *tp* (*tl x*) = *tl* (*tp x*)

lemma *fb-type*: *f* $\in$ *has-in-out-type* (*t* # *x*) (*t* # *y*) $\Longrightarrow$ *fb-f f* $\in$ *has-in-out-type x y*

lemma [*simp*]: *TI-f* (*Switch-f x y*) = *x* @ *y*

lemma *ID-f-type*[*simp*]: *ID-f ts* $\in$ *has-in-out-type ts ts*

lemma [*simp*]: *TI-f* (*ID-f ts*) = *ts*

lemma [*simp*]: *tp v* = *ts* $\Longrightarrow$ *ID-f ts v* = *Some v*

lemma *fb-switch-aux*: *f* $\in$ *has-in-out-type* (*t'* # *t* # *ts*) (*t'* # *t* # *ts'*) $\Longrightarrow$
par-f (*Switch-f* [*t'*] [*t*]) (*ID-f ts'*) $\circ_m$ (*f* $\circ_m$ *par-f* (*Switch-f* [*t*] [*t'*]) (*ID-f ts*)) =

$(\lambda\ v\ .\ (\mathit{if}\ \mathit{tp}\ v = t\ \#\ t'\ \#\ \mathit{ts}\ \mathit{then}\ \mathit{case}\ v\ \mathit{of}\ a\ \#\ b\ \#\ v' \Rightarrow (\mathit{case}\ f\ (b\ \#\ a\ \#\ v')\ \mathit{of}\ \mathit{Some}\ (c\ \#\ d\ \#\ u) \Rightarrow \mathit{Some}\ (d\ \#\ c\ \#\ u))\ \mathit{else}\ \mathit{None}))$

lemma *TI-f-fb-f*[*simp*]: $f \in$ *has-in-out-type* $(t\ \#\ \mathit{ts})\ (\ t\ \#\ \mathit{ts}') \Longrightarrow \mathit{TI\text{-}f}\ (\mathit{fb\text{-}f}\ f) = \mathit{ts}$

declare [[*show-types*=*false*]]

lemma *fb-t-type*: *fb-t t* $(\lambda a.\ \mathit{if}\ \mathit{tv}\ a = t\ \mathit{then}\ f\ a\ \mathit{else}\ g\ a) = \mathit{fb\text{-}t}\ t\ f$

lemma *le-values-same-type*: $a \leq b \Longrightarrow \mathit{tv}\ a = \mathit{tv}\ b$

thm *has-type-out-type*

definition *mono-f* $= \{f\ .\ (\forall\ x\ y\ .\ \mathit{le\text{-}list}\ x\ y \longrightarrow \mathit{le\text{-}list}\ (\mathit{the}\ (f\ x))\ (\mathit{the}\ (f\ y)))\}$

lemma [*simp*]: *le-list v v*

lemma *le-pref*: $\bigwedge\ v\ x\ .\ \mathit{le\text{-}list}\ u\ v \Longrightarrow \mathit{le\text{-}list}\ (\mathit{pref}\ u\ x)\ (\mathit{pref}\ v\ x)$

lemma *le-suff*: $\bigwedge\ v\ x\ .\ \mathit{le\text{-}list}\ u\ v \Longrightarrow \mathit{le\text{-}list}\ (\mathit{suff}\ u\ x)\ (\mathit{suff}\ v\ x)$

lemma *le-list-append*: $\bigwedge\ y\ .\ \mathit{le\text{-}list}\ x\ y \Longrightarrow \mathit{le\text{-}list}\ x'\ y' \Longrightarrow \mathit{le\text{-}list}\ (x\ @\ x')\ (y\ @\ y')$

thm *monoD*

lemma *mono-fD*: $f \in \mathit{mono\text{-}f} \Longrightarrow \mathit{le\text{-}list}\ x\ y \Longrightarrow \mathit{le\text{-}list}\ (\mathit{the}\ (f\ x))\ (\mathit{the}\ (f\ y))$

lemma *le-values-list-same-type*: $\bigwedge\ (y{::}\mathit{Values\ list})\ .\ \mathit{le\text{-}list}\ x\ y \Longrightarrow \mathit{tp}\ x = \mathit{tp}\ y$

lemma *map-comp-mono*: $f \in \mathit{mono\text{-}f} \Longrightarrow g \in \mathit{mono\text{-}f} \Longrightarrow (\bigwedge\ x\ y\ .\ \mathit{tp}\ x = \mathit{tp}\ y \Longrightarrow f\ x = \mathit{None} \Longrightarrow f\ y = \mathit{None}) \Longrightarrow (\bigwedge\ x\ y\ .\ \mathit{tp}\ x = \mathit{tp}\ y \Longrightarrow g\ x = \mathit{None} \Longrightarrow g\ y = \mathit{None}) \Longrightarrow g \circ_m f \in \mathit{mono\text{-}f}$

lemma *par-mono*: $f \in \mathit{mono\text{-}f} \Longrightarrow g \in \mathit{mono\text{-}f} \Longrightarrow (\bigwedge\ x\ y\ .\ \mathit{tp}\ x = \mathit{tp}\ y \Longrightarrow f\ x = \mathit{None} \Longrightarrow f\ y = \mathit{None}) \Longrightarrow (\bigwedge\ x\ y\ .\ \mathit{tp}\ x = \mathit{tp}\ y \Longrightarrow g\ x = \mathit{None} \Longrightarrow g\ y = \mathit{None}) \Longrightarrow \mathit{par\text{-}f}\ f\ g \in \mathit{mono\text{-}f}$

lemma *mono-f-emono*: $f \in \mathit{mono\text{-}f} \Longrightarrow (\bigwedge\ x\ y\ .\ \mathit{tp}\ x = \mathit{tp}\ y \Longrightarrow f\ x = \mathit{None} \Longrightarrow f\ y = \mathit{None}) \Longrightarrow \mathit{mono}\ A \Longrightarrow \mathit{mono}\ B \Longrightarrow \mathit{emono}\ (\lambda a.\ A\ (\mathit{hd}\ (\mathit{the}\ (f\ (B\ a\ \#\ x)))))$

lemma *mono-fb-t-aux*: $f \in \mathit{mono\text{-}f} \Longrightarrow$

le-list x y $\Longrightarrow$ ($\bigwedge$*x y. tp x = tp y* $\Longrightarrow$ *f x = None* $\Longrightarrow$ *f y = None*) $\Longrightarrow$ *mono* (*A*::*'a*::*order* $\Rightarrow$ *'b*::*fin-cpo*) $\Longrightarrow$ *mono B*
$\Longrightarrow$ *B* (*Lfp* (λ*a. A* (*hd* (*the* (*f* (*B a* # *x*)))))) $\leq$ *B* (*Lfp* (λ*a. A* (*hd* (*the* (*f* (*B a* # *y*))))))

thm *mono-fb-t-aux* [*of f x y integer*]

lemma *mono-fb-f*: *f* $\in$ *mono-f* $\Longrightarrow$ *le-list x y* $\Longrightarrow$ ($\bigwedge$ *x y . tp x = tp y* $\Longrightarrow$ *f x = None* $\Longrightarrow$ *f y = None*)
$\Longrightarrow$ *fb-t* (*hd* (*TI-f f*)) (λ*a. hd* (*the* (*f* (*a* # *x*)))) $\leq$ *fb-t* (*hd* (*TI-f f*)) (λ*a. hd* (*the* (*f* (*a* # *y*))))

lemma *fb-mono*: *f* $\in$ *mono-f* $\Longrightarrow$ ($\bigwedge$ *x y . tp x = tp y* $\Longrightarrow$ *f x = None* $\Longrightarrow$ *f y = None*) $\Longrightarrow$ *fb-f f* $\in$ *mono-f*

lemma *mono-f-mono-a*[*simp*]: *f* $\in$ *mono-f* $\Longrightarrow$ *f* $\in$ *has-in-out-type* (*t* # *t'* # *ts*) *ts'* $\Longrightarrow$ *tp v = ts* $\Longrightarrow$ *mono-a* (λ*a b. hd* (*the* (*f* (*b* # *a* # *v*))))

lemma *mono-f-mono-a-b*[*simp*]: *f* $\in$ *mono-f* $\Longrightarrow$ *f* $\in$ *has-in-out-type* (*t* # *t'* # *ts*) *ts'* $\Longrightarrow$ *tp v = ts* $\Longrightarrow$ *mono-a* (λ*a b. hd* (*tl* (*the* (*f* (*a* # *b* # *v*)))))

lemma [*simp*]: *Switch-f x y* $\in$ *mono-f*

lemma [*simp*]: *ID-f x* $\in$ *mono-f*

lemma *has-type-None*: *f* $\in$ *has-in-out-type x y* $\Longrightarrow$ *tp u = tp v* $\Longrightarrow$ *f u = None* $\Longrightarrow$ *f v = None*

lemma *fb-f-commute*: *f* $\in$ *mono-f* $\Longrightarrow$ *f* $\in$ *has-in-out-type* (*t'* # *t* # *ts*) (*t'* # *t* # *ts'*) $\Longrightarrow$
fb-f (*fb-f* (*par-f* (*Switch-f* [*t'*] [*t*]) (*ID-f ts'*) $\circ_m$ (*f* $\circ_m$ *par-f* (*Switch-f* [*t*] [*t'*]) (*ID-f ts*)))) = (*fb-f* (*fb-f f*))

definition *typed-func* = ($\bigcup$ *x* . ($\bigcup$ *y* . *has-in-out-type x y*)) $\cap$ *mono-f*

typedef *func* = *typed-func*

definition *fb-func f* = *Abs-func* (*fb-f* (*Rep-func f*))

definition *TI-func f* = (*TI-f* (*Rep-func f*))
definition *TO-func f* = (*TO-f* (*Rep-func f*))
definition *ID-func t* = *Abs-func* (*ID-f t*)

definition *comp-func f g* = *Abs-func* ((*Rep-func g*) $\circ_m$ (*Rep-func f*))

definition *parallel-func f g* = *Abs-func* (*par-f* (*Rep-func f*) (*Rep-func g*))

definition *Dup-func x = Abs-func (Dup-f x)*

definition *Sink-func x = Abs-func (Sink-f x)*
definition *Switch-func x y = Abs-func (Switch-f x y)*

lemma [*simp*]: *ID-f t ∈ typed-func*

lemma *map-comp-typed-func*[*simp*]: *f ∈ typed-func ⟹ g ∈ typed-func ⟹ TI-f g = TO-f f ⟹ (g ∘m f) ∈ typed-func*

lemma *par-typed-func*[*simp*]: *f ∈ typed-func ⟹ g ∈ typed-func ⟹ par-f f g ∈ typed-func*

lemma *fb-typed-func*[*simp*]: *f ∈ typed-func ⟹ TI-f f = t # x ⟹ TO-f f = t # y ⟹ fb-f f ∈ typed-func*

lemma [*simp*]: *Switch-f x y ∈ typed-func*

lemma [*simp*]: *Dup-f x ∈ mono-f*

lemma [*simp*]: *Dup-f x ∈ typed-func*

lemma [*simp*]: *Sink-f x ∈ mono-f*

lemma [*simp*]: *Sink-f x ∈ typed-func*

thm *Rep-func*
thm *Abs-func-inverse*
thm *Rep-func-inverse*

lemma *map-comp-assoc*: *(f ∘m g) ∘m h = f ∘m (g ∘m h)*

lemma *map-comp-id*: *f ∈ has-in-out-type x y ⟹ (f ∘m ID-f x) = f*

lemma *id-map-comp*: *f ∈ has-in-out-type x y ⟹ (ID-f y ∘m f) = f*

lemma [*simp*]: *⋀ x x′ . tp v = x @ x′ @ x″ ⟹ pref (pref v (x @ x′)) x = pref v x*

lemma [*simp*]: *⋀ x x′ . tp v = x @ x′ @ x″ ⟹ suff (pref v (x @ x′)) x = pref (suff v x) x′*

lemma [*simp*]: *⋀ x x′ . tp v = x @ x′ @ x″ ⟹ suff (suff v x) x′ = suff v (x @ x′)*

lemma *par-f-assoc*: *f ∈ has-in-out-type x y ⟹ g ∈ has-in-out-type x′ y′ ⟹ h*

$\in$ *has-in-out-type* x'' $y'' \Longrightarrow$
par-f (*par-f f g*) *h* = *par-f f* (*par-f g h*)

lemma $f \in$ *has-in-out-type x y* $\Longrightarrow$ *par-f* (*ID-f* []) *f* = *f*

lemma *id-par-f*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ *par-f* (*ID-f* []) *f* = *f*

lemma [*simp*]: $\bigwedge x$. *tp v* = *x* $\Longrightarrow$ *pref v x* = *v*

lemma [*simp*]: $\bigwedge x$. *tp v* = *x* $\Longrightarrow$ *suff v x* = []

lemma *par-f-id*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ *par-f f* (*ID-f* []) = *f*
lemma [*simp*]: $\bigwedge x$. *tp v* = *x* @ *y* $\Longrightarrow$ *pref v x* @ *suff v x* = *v*

lemma [*simp*]: $\bigwedge x$. *tp v* = *x* @ x' $\Longrightarrow$ *tp* (*pref v x*) = *x*

lemma [*simp*]: $\bigwedge x$. *tp v* = *x* @ x' $\Longrightarrow$ *tp* (*suff v x*) = x'

lemma [*simp*]: $\bigwedge x$. *tp u* = *x* $\Longrightarrow$ *pref* (*u* @ *v*) *x* = *u*

lemma [*simp*]: $\bigwedge x$. *tp u* = *x* $\Longrightarrow$ *suff* (*u* @ *v*) *x* = *v*

lemma *par-comp-distrib*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ $g \in$ *has-in-out-type y z* $\Longrightarrow$

$f' \in$ *has-in-out-type* x' $y' \Longrightarrow$ $g' \in$ *has-in-out-type* y' $z' \Longrightarrow$
par-f g $g' \circ_m$ *par-f f* $f' =$ (*par-f* (*g* $\circ_m$ *f*) ($g' \circ_m$ f'))

lemma *TI-f-par*: $f \in$ *typed-func* $\Longrightarrow$ $g \in$ *typed-func* $\Longrightarrow$ *TI-f* (*par-f f g*) = *TI-f f* @ *TI-f g*

lemma *TO-f-par*: $f \in$ *typed-func* $\Longrightarrow$ $g \in$ *typed-func* $\Longrightarrow$ *TO-f* (*par-f f g*) = *TO-f f* @ *TO-f g*

lemma *TI-f-map-comp*[*simp*]: $f \in$ *typed-func* $\Longrightarrow$ $g \in$ *typed-func* $\Longrightarrow$ *TO-f g* = *TI-f f* $\Longrightarrow$ *TI-f* (*f* $\circ_m$ *g*) = *TI-f g*

lemma *TO-f-map-comp*[*simp*]: $f \in$ *typed-func* $\Longrightarrow$ $g \in$ *typed-func* $\Longrightarrow$ *TO-f g* = *TI-f f* $\Longrightarrow$ *TO-f* (*f* $\circ_m$ *g*) = *TO-f f*

lemma [*simp*]: *TI-f* (*Sink-f ts*) = *ts*

lemma [*simp*]: *TO-f* (*Sink-f ts*) = []

lemma *suff-append*: $\bigwedge t$. *tp x* = *t* $\Longrightarrow$ *suff* (*x* @ *y*) *t* = *y*

lemma [*simp*]: *TI-f* (*Dup-f x*) = *x*

lemma [*simp*]: *TO-f* (*Dup-f x*) = (*x* @ *x*)

lemma [*simp*]: *pref* (*x* @ *y*) (*tp x*) = *x*

lemma [*simp*]: *TO-f* (*Switch-f x y*) = (*y* @ *x*)

lemma [*simp*]: *TO-f* (*ID-f x*) = *x*

declare *TO-f-par* [*simp*]

declare *TI-f-par* [*simp*]

lemma [*simp*]: ⋀ *ts* . *tp x* = *ts* @ *ts'* @ *ts''* ⟹ *pref* (*suff x ts*) *ts'* @ *suff x* (*ts* @ *ts'*) = *suff x ts*

lemma [*simp*]: ⋀*ts* . *tp x* = *ts* ⟹ *suff* (*x* @ *y*) (*ts* @ *ts'*) = *suff y ts'*

lemma *AAA*: *S x* ≠ *None* ⟹ *tv a* = *t* ⟹ *tp x* = *TI-f S* ⟹ *the* ((*par-f* (*ID-f* [*t*]) *S*) (*a* # *x*)) = *a* # *the* (*S x*)

lemma *AAAb*: *S x* ≠ *None* ⟹ *tv a* = *t* ⟹ *tp x* = *TI-f S* ⟹ ((*par-f* (*ID-f* [*t*]) *S*) (*a* # *x*)) = *Some* (*a* # *the* (*S x*))

lemma *pref-suff-append*: ⋀ *ts* . *tp x* = *ts* @ *ts'* ⟹ *pref x ts* @ *suff x ts* = *x*

lemma [*simp*]: *Lfp* (λ *b*. *a*) = *a*

lemma [*simp*]: *fb-t* (*tv a*) (λ *b* . *a*) = *a*

interpretation *func*: *BaseOperation TI-func TO-func ID-func comp-func parallel-func Dup-func Sink-func Switch-func fb-func*
end

7 Diagrams with Named Inputs and Outputs

theory *Diagrams* **imports** *HBDAlgebra*
begin

This file contains the definition and properties for the named input output diagrams

record ('*var*, '*a*) *Dgr* =
In:: '*var list*
Out:: '*var list*
Trs:: '*a*

context *BaseOperationFeedbacklessVars*
begin

definition *Var A B* = (*Out A*) ⊗ (*In B*)

definition *io-diagram A* = (*TVs* (*In A*) = *TI* (*Trs A*) ∧ *TVs* (*Out A*) = *TO* (*Trs A*) ∧ *distinct* (*In A*) ∧ *distinct* (*Out A*))

definition *Comp* :: ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* (**infixl** ;; *70*) **where**
A ;; *B* = (*let I* = *In B* ⊖ *Var A B in let O'* = *Out A* ⊖ *Var A B in*
(|*In* = (*In A*) ⊕ *I*, *Out* = *O'* @ *Out B*,
Trs = [(*In A*) ⊕ *I* ⇝ *In A* @ *I*] *oo Trs A* ‖ [*I* ⇝ *I*] *oo* [*Out A* @ *I* ⇝ *O'* @ *In B*] *oo* ([*O'* ⇝ *O'*] ‖ *Trs B*) |))

lemma *io-diagram-Comp*: *io-diagram A* ⟹ *io-diagram B*
⟹ *set* (*Out A* ⊖ *In B*) ∩ *set* (*Out B*) = {} ⟹ *io-diagram* (*A* ;; *B*)

lemma *Comp-in-disjoint*:
assumes *io-diagram A*
and *io-diagram B*
and *set* (*In A*) ∩ *set* (*In B*) = {}
shows *A* ;; *B* = (*let I* = *In B* ⊖ *Var A B in let O'* = *Out A* ⊖ *Var A B in*
(|*In* = (*In A*) @ *I*, *Out* = *O'* @ *Out B*, *Trs* = *Trs A* ‖ [*I* ⇝ *I*] *oo* [*Out A* @ *I* ⇝ *O'* @ *In B*] *oo* ([*O'* ⇝ *O'*] ‖ *Trs B*) |))

lemma *Comp-full*: *io-diagram A* ⟹ *io-diagram B* ⟹ *Out A* = *In B* ⟹
A ;; *B* = (|*In* = *In A*, *Out* = *Out B*, *Trs* = *Trs A oo Trs B* |)

lemma *Comp-in-out*: *io-diagram A* ⟹ *io-diagram B* ⟹ *set* (*Out A*) ⊆ *set* (*In B*) ⟹
A ;; *B* = (*let I* = *diff* (*In B*) (*Var A B*) *in let O'* = *diff* (*Out A*) (*Var A B*) *in*
(|*In* = *In A* ⊕ *I*, *Out* = *Out B*, *Trs* = [*In A* ⊕ *I* ⇝ *In A* @ *I*] *oo Trs A* ‖ [*I* ⇝ *I*] *oo* [*Out A* @ *I* ⇝ *In B*] *oo Trs B* |))

lemma *Comp-assoc-new*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹
set (*Out A* ⊖ *In B*) ∩ *set* (*Out B*) = {} ⟹ *set* (*Out A* ⊗ *In B*) ∩ *set* (*In C*) = {}
⟹ *A* ;; *B* ;; *C* = *A* ;; (*B* ;; *C*)

lemma *Comp-assoc-a*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹
set (*In B*) ∩ *set* (*In C*) = {} ⟹
set (*Out A*) ∩ *set* (*Out B*) = {} ⟹
A ;; *B* ;; *C* = *A* ;; (*B* ;; *C*)

definition *Parallel* :: ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* (**infixl** ||| *80*) **where**
A ||| *B* = (|*In* = *In A* ⊕ *In B*, *Out* = *Out A* @ *Out B*, *Trs* = [*In A* ⊕ *In B* ⇝

In A @ In B] *oo* (*Trs A* ∥ *Trs B*) ⦇⦈

lemma *io-diagram-Parallel*: *io-diagram A* ⟹ *io-diagram B* ⟹ *set* (*Out A*) ∩ *set* (*Out B*) = {} ⟹ *io-diagram* (*A* ||| *B*)

lemma *Parallel-indep*: *io-diagram A* ⟹ *io-diagram B* ⟹ *set* (*In A*) ∩ *set* (*In B*) = {} ⟹
A ||| *B* = ⦇*In* = *In A* @ *In B*, *Out* = *Out A* @ *Out B*, *Trs* = (*Trs A* ∥ *Trs B*) ⦈

lemma *Parallel-assoc-gen*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹
A ||| *B* ||| *C* = *A* ||| (*B* ||| *C*)

definition *VarFB A* = *Var A A*
definition *InFB A*= *In A* ⊖ *VarFB A*
definition *OutFB A* = *Out A* ⊖ *VarFB A*

definition *FB* :: (*'var*, *'a*) *Dgr* ⇒ (*'var*, *'a*) *Dgr* **where**
FB A = (*let I* = *In A* ⊖ *Var A A in let O'* = *Out A* ⊖ *Var A A in*
⦇*In* = *I*, *Out* = *O'*, *Trs* = (*fb* ^^ (*length* (*Var A A*))) ([*Var A A* @ *I* ⇝ *In A*] *oo Trs A oo* [*Out A* ⇝ *Var A A* @ *O'*]) ⦈)

lemma *Type-ok-FB*: *io-diagram A* ⟹ *io-diagram* (*FB A*)

lemma *perm-var-Par*: *io-diagram A* ⟹ *io-diagram B* ⟹ *set* (*In A*) ∩ *set* (*In B*) = {}
⟹ *perm* (*Var* (*A* ||| *B*) (*A* ||| *B*)) (*Var A A* @ *Var B B* @ *Var A B* @ *Var B A*)

lemma *distinct-Parallel-Var*[*simp*]: *io-diagram A* ⟹ *io-diagram B*
⟹ *set* (*Out A*) ∩ *set* (*Out B*) = {} ⟹ *distinct* (*Var* (*A* ||| *B*) (*A* ||| *B*))

lemma *distinct-Parallel-In*[*simp*]: *io-diagram A* ⟹ *io-diagram B* ⟹ *distinct* (*In* (*A* ||| *B*))

lemma *drop-assumption*: *p* ⟹ *True*

lemma *Dgr-eq*: *In A* = *x* ⟹ *Out A* = *y* ⟹ *Trs A* = *S* ⟹ ⦇*In* = *x*, *Out* = *y*, *Trs* = *S*⦈ = *A*

lemma *Var-FB*[*simp*]: *Var* (*FB A*) (*FB A*) = []

theorem *FB-idemp*: *io-diagram A* ⟹ *FB* (*FB A*) = *FB A*

definition *VarSwitch* :: *'var list* ⇒ *'var list* ⇒ (*'var*, *'a*) *Dgr* ([[- ⇝ -]]) **where** *VarSwitch x y* = (⌊*In* = *x*, *Out* = *y*, *Trs* = [*x* ⇝ *y*]⌋)

definition *in-equiv A B* = (*perm* (*In A*) (*In B*) ∧ *Trs A* = [*In A* ⇝ *In B*] *oo Trs B* ∧ *Out A* = *Out B*)
definition *out-equiv A B* = (*perm* (*Out A*) (*Out B*) ∧ *Trs A* = *Trs B oo* [*Out B* ⇝ *Out A*] ∧ *In A* = *In B*)

definition *in-out-equiv A B* = (*perm* (*In A*) (*In B*) ∧ *perm* (*Out A*) (*Out B*) ∧ *Trs A* = [*In A* ⇝ *In B*] *oo Trs B oo* [*Out B* ⇝ *Out A*])

lemma *in-equiv-io-diagram*: *in-equiv A B* ⟹ *io-diagram B* ⟹ *io-diagram A*

lemma *in-out-equiv-io-diagram*: *in-out-equiv A B* ⟹ *io-diagram B* ⟹ *io-diagram A*

lemma *in-equiv-sym*: *io-diagram B* ⟹ *in-equiv A B* ⟹ *in-equiv B A*

lemma *in-equiv-eq*: *io-diagram A* ⟹ *A* = *B* ⟹ *in-equiv A B*

lemma [*simp*]: *io-diagram A* ⟹ [*In A* ⇝ *In A*] *oo Trs A oo* [*Out A* ⇝ *Out A*] = *Trs A*

lemma *in-equiv-tran*: *io-diagram C* ⟹ *in-equiv A B* ⟹ *in-equiv B C* ⟹ *in-equiv A C*

lemma *in-out-equiv-refl*: *io-diagram A* ⟹ *in-out-equiv A A*

lemma *in-out-equiv-sym*: *io-diagram A* ⟹ *io-diagram B* ⟹ *in-out-equiv A B* ⟹ *in-out-equiv B A*

lemma *in-out-equiv-tran*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹ *in-out-equiv A B* ⟹ *in-out-equiv B C* ⟹ *in-out-equiv A C*

lemma [*simp*]: *distinct* (*Out A*) ⟹ *distinct* (*Var A B*)

lemma [*simp*]: *set* (*Var A B*) ⊆ *set* (*Out A*)
lemma [*simp*]: *set* (*Var A B*) ⊆ *set* (*In B*)

lemmas *fb-indep-sym* = *fb-indep* [*THEN sym*]

declare *length-TVs* [*simp*]

end

primrec *op-list* :: $'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ list \Rightarrow 'a$ **where**
op-list e opr [] = *e* |
op-list e opr (*a* # *x*) = *opr a* (*op-list e opr x*)

primrec *inter-set* :: $'a\ list \Rightarrow 'a\ set \Rightarrow 'a\ list$ **where**
inter-set [] *X* = [] |
inter-set (*x* # *xs*) *X* = (*if x* ∈ *X then x* # *inter-set xs X else inter-set xs X*)

lemma *list-inter-set*: *x* ⊗ *y* = *inter-set x* (*set y*)

fun *map2* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow bool$ **where**
map2 f [] [] = *True* |
map2 f (*a* # *x*) (*b* # *y*) = (*f a b* ∧ *map2 f x y*) |
map2 - - - = *False*

thm *map-def*

context *BaseOperationFeedbacklessVars*
begin
definition *ParallelId* :: $('var, 'a)\ Dgr$ (□)
where □ = (|*In* = [], *Out* = [], *Trs* = *ID* []|)

lemma [*simp*]: *Out* □ = []

lemma [*simp*]: *In* □ = []

lemma [*simp*]: *Trs* □ = *ID* []

lemma *ParallelId-right*[*simp*]: *io-diagram A* ⟹ *A* ||| □ = *A*

lemma *ParallelId-left*: *io-diagram A* ⟹ □ ||| *A* = *A*

definition *parallel-list* = *op-list* (*ID* []) (*op* ||)

definition *Parallel-list* = *op-list* □ (*op* |||)

lemma [*simp*]: *Parallel-list* [] = □

definition *io-distinct As* = (*distinct* (*concat* (*map In As*)) ∧ *distinct* (*concat* (*map Out As*)) ∧ (∀ *A* ∈ *set As* . *io-diagram A*))

definition *io-rel A* = *set* (*Out A*) × *set* (*In A*)

definition *IO-Rel As* = $\bigcup$ (*set* (*map io-rel As*))

definition *out A* = *hd* (*Out A*)

definition *Type-OK As* = ((∀ *B* ∈ *set As* . *io-diagram B* ∧ *length* (*Out B*) = *1*)
∧ *distinct* (*concat* (*map Out As*)))

lemma *concat-map-out*: (∀ *A* ∈ *set As* . *length* (*Out A*) = *1*) ⟹ *concat* (*map Out As*) = *map out As*

lemma *Type-OK-simp*: *Type-OK As* = ((∀ *B* ∈ *set As* . *io-diagram B* ∧ *length* (*Out B*) = *1*) ∧ *distinct* (*map out As*))

definition *single-out A* = (*io-diagram A* ∧ *length* (*Out A*) = *1*)

definition *CompA* :: ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* ⇒ ('*var*, '*a*) *Dgr* (**infixl** ▷ *75*) **where**

A ▷ *B* = (*if out A* ∈ *set* (*In B*) *then A* ;; *B else B*)

definition *internal As* = {*x* . (∃ *A* ∈ *set As* . ∃ *B* ∈ *set As* . *x* ∈ *set* (*Out A*) ∧ *x* ∈ *set* (*In B*))}

primrec *get-comp-out* :: '*var* ⇒ ('*var*, '*a*) *Dgr list* ⇒ ('*var*, '*a*) *Dgr* **where**
get-comp-out x [] = (|*In* = [*x*], *Out* = [*x*], *Trs* = [[*x*] ⇝ [*x*]]|) |
get-comp-out x (*A* # *As*) = (*if x* ∈ *set* (*Out A*) *then A else get-comp-out x As*)

primrec *get-other-out* :: '*c* ⇒ ('*c*, '*d*) *Dgr list* ⇒ ('*c*, '*d*) *Dgr list* **where**
get-other-out x [] = [] |
get-other-out x (*A* # *As*) = (*if x* ∈ *set* (*Out A*) *then get-other-out x As else A* # *get-other-out x As*)

definition *fb-less-step A As* = *map* (*CompA A*) *As*

definition *fb-out-less-step x As* = *fb-less-step* (*get-comp-out x As*) (*get-other-out x As*)

primrec *fb-less* :: '*var list* ⇒ ('*var*, '*a*) *Dgr list* ⇒ ('*var*, '*a*) *Dgr list* **where**
fb-less [] *As* = *As* |
fb-less (*x* # *xs*) *As* = *fb-less xs* (*fb-out-less-step x As*)

lemma [*simp*]: *VarFB* □ = []
lemma [*simp*]: *InFB* □ = []
lemma [*simp*]: *OutFB* □ = []

definition *loop-free As* = (∀ *x* . (*x*,*x*) ∉ (*IO-Rel As*)$^+$)

lemma [*simp*]: *Parallel-list* (*A* # *As*) = (*A* ||| *Parallel-list As*)

lemma [*simp*]: *Out* (*A* ||| *B*) = *Out A* @ *Out B*

lemma [*simp*]: *In* (*A* ||| *B*) = *In A* ⊕ *In B*

lemma *Type-OK-cons*: *Type-OK* (*A* # *As*) = (*io-diagram A* ∧ *length* (*Out A*) = *1* ∧ *set* (*Out A*) ∩ (⋃ *a*∈*set As*. *set* (*Out a*)) = {} ∧ *Type-OK As*)

lemma *Out-Parallel*: *Out* (*Parallel-list As*) = *concat* (*map Out As*)

lemma *internal-cons*: *internal* (*A* # *As*) = {*x*. *x* ∈ *set* (*Out A*) ∧ (*x* ∈ *set* (*In A*) ∨ (∃ *B*∈*set As*. *x* ∈ *set* (*In B*)))} ∪ {*x* . (∃ *Aa*∈*set As*. *x* ∈ *set* (*Out Aa*) ∧ (*x* ∈ *set* (*In A*)))}
∪ *internal As*

lemma *Out-out*: *length* (*Out A*) = *Suc 0* ⟹ *Out A* = [*out A*]

lemma *Type-OK-out*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *Out A* = [*out A*]

lemma *In-Parallel*: *In* (*Parallel-list As*) = *op-list* [] (*op* ⊕) (*map In As*)

lemma [*simp*]: *set* (*op-list* [] *op* ⊕ *xs*) = ⋃ *set* (*map set xs*)

lemma *internal-VarFB*: *Type-OK As* ⟹ *internal As* = *set* (*VarFB* (*Parallel-list As*))

lemma *map-Out-fb-less-step*: *length* (*Out A*) = *1* ⟹ *map Out* (*fb-less-step A As*) = *map Out As*

lemma *mem-get-comp-out*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *get-comp-out* (*out A*) *As* = *A*

lemma *map-Out-fb-out-less-step*: *A* ∈ *set As* ⟹ *Type-OK As* ⟹ *a* = *out A* ⟹ *map Out* (*fb-out-less-step a As*) = *map Out* (*get-other-out a As*)

lemma [*simp*]: *Type-OK* (*A* # *As*) ⟹ *Type-OK As*

lemma *Type-OK-Out*: *Type-OK* $(A \# As) \Longrightarrow Out\ A = [out\ A]$

lemma *concat-map-Out-get-other-out*: *Type-OK* $As \Longrightarrow concat\ (map\ Out\ (get\text{-}other\text{-}out\ a\ As)) = (concat\ (map\ Out\ As) \ominus [a])$

thm *Out-out*

lemma *VarFB-cons-out*: *Type-OK* $As \Longrightarrow VarFB\ (Parallel\text{-}list\ As) = a \# L \Longrightarrow \exists\ A \in set\ As\ .\ out\ A = a$

lemma *VarFB-cons-out-In*: *Type-OK* $As \Longrightarrow VarFB\ (Parallel\text{-}list\ As) = a \# L \Longrightarrow \exists\ B \in set\ As\ .\ a \in set\ (In\ B)$

lemma *AAA-a*: *Type-OK* $(A \# As) \Longrightarrow A \notin set\ As$

lemma *AAA-b*: $(\forall\ A \in set\ As.\ a \notin set\ (Out\ A)) \Longrightarrow get\text{-}other\text{-}out\ a\ As = As$

lemma *AAA-d*: *Type-OK* $(A \# As) \Longrightarrow \forall\, Aa{\in}set\ As.\ out\ A \neq out\ Aa$

lemma *mem-get-other-out*: *Type-OK* $As \Longrightarrow A \in set\ As \Longrightarrow get\text{-}other\text{-}out\ (out\ A)\ As = (As \ominus [A])$

lemma *In-CompA*: $In\ (A \rhd B) = (if\ out\ A \in set\ (In\ B)\ then\ In\ A \oplus (In\ B \ominus Out\ A)\ else\ In\ B)$

lemma *union-set-In-CompA*: $\bigwedge B\ .\ length\ (Out\ A) = 1 \Longrightarrow B \in set\ As \Longrightarrow out\ A \in set\ (In\ B)$
$\Longrightarrow (\bigcup x{\in}set\ As.\ set\ (In\ (CompA\ A\ x))) = set\ (In\ A) \cup ((\bigcup B \in set\ As\ .\ set\ (In\ B)) - \{out\ A\})$

lemma *BBBB-e*: *Type-OK* $As \Longrightarrow VarFB\ (Parallel\text{-}list\ As) = out\ A \# L \Longrightarrow A \in set\ As \Longrightarrow out\ A \notin set\ L$

lemma *BBBB-f*: *loop-free* $As \Longrightarrow$
Type-OK $As \Longrightarrow A \in set\ As \Longrightarrow B \in set\ As \Longrightarrow out\ A \in set\ (In\ B) \Longrightarrow B \neq A$

thm *union-set-In-CompA*

lemma [*simp*]: $x \in set\ (Out\ (get\text{-}comp\text{-}out\ x\ As))$

lemma *comp-out-in*: $A \in set\ As \Longrightarrow a \in set\ (Out\ A) \Longrightarrow (get\text{-}comp\text{-}out\ a\ As) \in set\ As$

lemma [*simp*]: $a \in internal\ As \Longrightarrow get\text{-}comp\text{-}out\ a\ As \in set\ As$

lemma *out-CompA*: $length\ (Out\ A) = 1 \Longrightarrow out\ (CompA\ A\ B) = out\ B$

lemma *Type-OK-loop-free-elem*: $Type\text{-}OK\ As \Longrightarrow loop\text{-}free\ As \Longrightarrow A \in set\ As \Longrightarrow out\ A \notin set\ (In\ A)$

lemma *BBB-a*: $length\ (Out\ A) = 1 \Longrightarrow Out\ (CompA\ A\ B) = Out\ B$

lemma *BBB-b*: $length\ (Out\ A) = 1 \Longrightarrow map\ (Out \circ CompA\ A)\ As = map\ Out\ As$

lemma *VarFB-fb-out-less-step-gen*:
assumes *loop-free As*
assumes *Type-OK As*
and *internal-a*: $a \in internal\ As$
shows $VarFB\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) = (VarFB\ (Parallel\text{-}list\ As)) \ominus [a]$

thm *internal-VarFB*
thm *VarFB-fb-out-less-step-gen*

lemma *VarFB-fb-out-less-step*: $loop\text{-}free\ As \Longrightarrow Type\text{-}OK\ As \Longrightarrow VarFB\ (Parallel\text{-}list\ As) = a\ \#\ L \Longrightarrow VarFB\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) = L$

lemma *Parallel-list-cons*:$Parallel\text{-}list\ (a\ \#\ As) = a\ |||\ Parallel\text{-}list\ As$

lemma *io-diagram-parallel-list*: $Type\text{-}OK\ As \Longrightarrow io\text{-}diagram\ (Parallel\text{-}list\ As)$

lemma *BBB-c*: $distinct\ (map\ f\ As) \Longrightarrow distinct\ (map\ f\ (As \ominus Bs))$

lemma *io-diagram-CompA*: $io\text{-}diagram\ A \Longrightarrow length\ (Out\ A) = 1 \Longrightarrow io\text{-}diagram\ B \Longrightarrow io\text{-}diagram\ (CompA\ A\ B)$

lemma *Type-OK-fb-out-less-step-aux*: $Type\text{-}OK\ As \Longrightarrow A \in set\ As \Longrightarrow Type\text{-}OK\ (fb\text{-}less\text{-}step\ A\ (As \ominus [A]))$

thm *VarFB-cons-out*

theorem *Type-OK-fb-out-less-step-new*: $Type\text{-}OK\ As \Longrightarrow$
$a \in internal\ As \Longrightarrow$

Bs = fb-out-less-step a As ⟹ Type-OK Bs

theorem *Type-OK-fb-out-less-step*: *loop-free As ⟹ Type-OK As ⟹*
VarFB (Parallel-list As) = a # L ⟹ Bs = fb-out-less-step a As ⟹ Type-OK Bs

lemma *perm-FB-Parallel*[*simp*]: *loop-free As ⟹ Type-OK As*
⟹ VarFB (Parallel-list As) = a # L ⟹ Bs = fb-out-less-step a As
⟹ perm (In (FB (Parallel-list As))) (In (FB (Parallel-list Bs)))

lemma [*simp*]: *loop-free As ⟹ Type-OK As ⟹*
VarFB (Parallel-list As) = a # L ⟹
Out (FB (Parallel-list (fb-out-less-step a As))) = Out (FB (Parallel-list As))

lemma *TI-Parallel-list*: *(∀ A ∈ set As . io-diagram A) ⟹ TI (Trs (Parallel-list As)) = TVs (op-list [] op ⊕ (map In As))*

lemma *TO-Parallel-list*: *(∀ A ∈ set As . io-diagram A) ⟹ TO (Trs (Parallel-list As)) = TVs (concat (map Out As))*

lemma *fbtype-aux*: *(Type-OK As) ⟹ loop-free As ⟹ VarFB (Parallel-list As) = a # L ⟹*
fbtype ([L @ (In (Parallel-list (fb-out-less-step a As)) ⊖ L) ⇝ In (Parallel-list (fb-out-less-step a As))] oo Trs (Parallel-list (fb-out-less-step a As)) oo
[Out (Parallel-list (fb-out-less-step a As)) ⇝ L @ (Out (Parallel-list (fb-out-less-step a As)) ⊖ L)])
(TVs L) (TO [In (Parallel-list As) ⊖ a # L ⇝ In (Parallel-list (fb-out-less-step a As)) ⊖ L]) (TVs (Out (Parallel-list (fb-out-less-step a As)) ⊖ L))

lemma *fb-indep-left-a*: *fbtype S tsa (TO A) ts ⟹ A oo (fb^^(length tsa)) S = (fb^^(length tsa)) ((ID tsa ∥ A) oo S)*

lemma *parallel-list-cons*: *parallel-list (A # As) = A ∥ parallel-list As*

lemma *TI-parallel-list*: *(∀ A ∈ set As . io-diagram A) ⟹ TI (parallel-list (map Trs As)) = TVs (concat (map In As))*

lemma *TO-parallel-list*: *(∀ A ∈ set As . io-diagram A) ⟹TO (parallel-list (map Trs As)) = TVs (concat (map Out As))*

lemma *Trs-Parallel-list-aux-a*: *Type-OK As ⟹ io-diagram a ⟹*

$[In\ a \oplus In\ (Parallel\text{-}list\ As) \rightsquigarrow In\ a\ @\ In\ (Parallel\text{-}list\ As)]\ oo\ Trs\ a \parallel ([In\ (Parallel\text{-}list\ As) \rightsquigarrow concat\ (map\ In\ As)]\ oo\ parallel\text{-}list\ (map\ Trs\ As)) =$
$[In\ a \oplus In\ (Parallel\text{-}list\ As) \rightsquigarrow In\ a\ @\ In\ (Parallel\text{-}list\ As)]\ oo\ ([In\ a \rightsquigarrow In\ a\ ] \parallel [In\ (Parallel\text{-}list\ As) \rightsquigarrow concat\ (map\ In\ As)]\ oo\ Trs\ a \parallel parallel\text{-}list\ (map\ Trs\ As))$

lemma *Trs-Parallel-list-aux-b* :*distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $z \subseteq$ *set* $y \Longrightarrow [x \oplus y \rightsquigarrow x\ @\ y]\ oo\ [x \rightsquigarrow x] \parallel [y \rightsquigarrow z] = [x \oplus y \rightsquigarrow x\ @\ z]$

lemma *Trs-Parallel-list*: *Type-OK* $As \Longrightarrow Trs\ (Parallel\text{-}list\ As) = [In\ (Parallel\text{-}list\ As) \rightsquigarrow concat\ (map\ In\ As)]\ oo\ parallel\text{-}list\ (map\ Trs\ As)$

lemma *CompA-Id*[*simp*]: $A \rhd \square = \square$

lemma *io-diagram-ParallelId*[*simp*]: *io-diagram* $\square$

lemma *in-equiv-aux-a* :*distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $z \subseteq$ *set* $x \Longrightarrow [x \oplus y \rightsquigarrow x\ @\ y]\ oo\ [x \rightsquigarrow z] \parallel [y \rightsquigarrow y] = [x \oplus y \rightsquigarrow z\ @\ y]$

lemma *in-equiv-Parallel-aux-d*: *distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $u \subseteq$ *set* $x \Longrightarrow$ *perm* $y\ v$
$\Longrightarrow [x \oplus y \rightsquigarrow x\ @\ v]\ oo\ [x \rightsquigarrow u] \parallel [v \rightsquigarrow v] = [x \oplus y \rightsquigarrow u\ @\ v]$

lemma *comp-par-switch-subst*: *distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $u \subseteq$ *set* $x \Longrightarrow$ *set* $v \subseteq$ *set* y
$\Longrightarrow [x \oplus y \rightsquigarrow x\ @\ y]\ oo\ [x \rightsquigarrow u] \parallel [y \rightsquigarrow v] = [x \oplus y \rightsquigarrow u\ @\ v]$

lemma *in-equiv-Parallel-aux-b* :*distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *perm* $u\ x \Longrightarrow$ *perm* $y\ v \Longrightarrow [x \oplus y \rightsquigarrow x\ @\ y]\ oo\ [x \rightsquigarrow u] \parallel [y \rightsquigarrow v] = [x \oplus y \rightsquigarrow u\ @\ v]$

lemma [*simp*]: *set* $x \subseteq$ *set* $(x \oplus y)$

lemma [*simp*]: *set* $y \subseteq$ *set* $(x \oplus y)$

declare *distinct-addvars* [*simp*]

lemma *in-equiv-Parallel*: *io-diagram* $B \Longrightarrow$ *io-diagram* $B' \Longrightarrow$ *in-equiv* $A\ B \Longrightarrow$ *in-equiv* $A'\ B' \Longrightarrow$ *in-equiv* $(A \mid\mid\mid A')\ (B \mid\mid\mid B')$

thm *local.BBB-a*

lemma *map-Out-CompA*: *length* (*Out A*) = *1* ⟹ *map* (*out* ∘ *CompA A*) *As* = *map out As*

lemma *CompA-in*[*simp*]: *out A* ∈ *set* (*In B*) ⟹ *A* ▷ *B* = *A* ;; *B*

lemma *CompA-not-in*[*simp*]: *out A* ∉ *set* (*In B*) ⟹ *A* ▷ *B* = *B*

lemma *in-equiv-CompA-Parallel-a*: *deterministic* (*Trs A*) ⟹ *length* (*Out A*) = *1* ⟹ *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C*
⟹ *out A* ∈ *set* (*In B*) ⟹ *out A* ∈ *set* (*In C*)
⟹ *in-equiv* ((*A* ▷ *B*) ||| (*A* ▷ *C*)) (*A* ▷ (*B* ||| *C*))

lemma *in-equiv-CompA-Parallel-c*: *length* (*Out A*) = *1* ⟹ *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹ *out A* ∉ *set* (*In B*) ⟹ *out A* ∈ *set* (*In C*) ⟹
in-equiv (*CompA A B* ||| *CompA A C*) (*CompA A* (*B* ||| *C*))

lemmas *distinct-addvars distinct-diff*

lemma *io-diagram-distinct*: **assumes** *A*: *io-diagram A* **shows** [*simp*]: *distinct* (*In A*)
and [*simp*]: *distinct* (*Out A*) **and** [*simp*]: *TI* (*Trs A*) = *TVs* (*In A*)
and [*simp*]: *TO* (*Trs A*) = *TVs* (*Out A*)

declare *Subst-not-in-a* [*simp*]
declare *Subst-not-in* [*simp*]

lemma [*simp*]: *set x′* ∩ *set z* = {} ⟹ *TVs x* = *TVs y* ⟹ *TVs x′* = *TVs y′* ⟹ *Subst* (*x* @ *x′*) (*y* @ *y′*) *z* = *Subst x y z*

lemma [*simp*]: *set x* ∩ *set z* = {} ⟹ *TVs x* = *TVs y* ⟹ *TVs x′* = *TVs y′* ⟹ *Subst* (*x* @ *x′*) (*y* @ *y′*) *z* = *Subst x′ y′ z*

lemma [*simp*]: *set x* ∩ *set z* = {} ⟹ *TVs x* = *TVs y* ⟹ *Subst x y z* = *z*

lemma [*simp*]: *distinct x* ⟹ *TVs x* = *TVs y* ⟹ *Subst x y x* = *y*

lemma *TVs x* = *TVs y* ⟹ *length x* = *length y*

thm *length-TVs*

lemma *in-equiv-switch-Parallel*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$
in-equiv (*A* ||| *B*) ((*B* ||| *A*) ;; [[*Out B* @ *Out A* $\rightsquigarrow$ *Out A* @ *Out B*]])

lemma *in-out-equiv-Parallel*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$ *in-out-equiv* (*A* ||| *B*) (*B* ||| *A*)

declare *Subst-eq* [*simp*]

lemma assumes *in-equiv A A'* **shows** [*simp*]: *perm* (*In A*) (*In A'*)

lemma *Subst-cancel-left-type*: *set x* $\cap$ *set z* = {} $\Longrightarrow$ *TVs x* = *TVs y* $\Longrightarrow$ *Subst* (*x* @ *z*) (*y* @ *z*) *w* = *Subst x y w*

lemma *diff-eq-set-right*: *set y* = *set z* $\Longrightarrow$ (*x* $\ominus$ *y*) = (*x* $\ominus$ *z*)

lemma [*simp*]: *set* (*y* $\ominus$ *x*) $\cap$ *set x* = {}

lemma *in-equiv-Comp*: *io-diagram A'* $\Longrightarrow$ *io-diagram B'* $\Longrightarrow$ *in-equiv A A'* $\Longrightarrow$ *in-equiv B B'* $\Longrightarrow$ *in-equiv* (*A* ;; *B*) (*A'* ;; *B'*)

lemma *io-diagram A'* $\Longrightarrow$ *io-diagram B'* $\Longrightarrow$ *in-equiv A A'* $\Longrightarrow$ *in-equiv B B'* $\Longrightarrow$ *in-equiv* (*CompA A B*) (*CompA A' B'*)

thm *in-equiv-tran*

thm *in-equiv-CompA-Parallel-c*

lemma *comp-parallel-distrib-a*: *TO A* = *TI B* $\Longrightarrow$ (*A oo B*) $\parallel$ *C* = (*A* $\parallel$ (*ID* (*TI C*))) *oo* (*B* $\parallel$ *C*)

lemma *comp-parallel-distrib-b*: *TO A* = *TI B* $\Longrightarrow$ *C* $\parallel$ (*A oo B*) = ((*ID* (*TI C*)) $\parallel$ *A*) *oo* (*C* $\parallel$ *B*)

thm *switch-comp-subst*

lemma *CCC-d*: *distinct x* $\Longrightarrow$ *distinct y'* $\Longrightarrow$ *set y* $\subseteq$ *set x* $\Longrightarrow$ *set z* $\subseteq$ *set x* $\Longrightarrow$ *set u* $\subseteq$ *set y'* $\Longrightarrow$ *TVs y* = *TVs y'* $\Longrightarrow$
TVs z = *ts* $\Longrightarrow$ [*x* $\rightsquigarrow$ *y* @ *z*] *oo* [*y'* $\rightsquigarrow$ *u*] $\parallel$ (*ID ts*) = [*x* $\rightsquigarrow$ *Subst y' y u* @

$z]$

lemma *CCC-e*: *distinct* $x \Longrightarrow$ *distinct* $y' \Longrightarrow$ *set* $y \subseteq$ *set* $x \Longrightarrow$ *set* $z \subseteq$ *set* $x \Longrightarrow$ *set* $u \subseteq$ *set* $y' \Longrightarrow$ *TVs* $y =$ *TVs* $y' \Longrightarrow$
TVs $z = ts \Longrightarrow [x \rightsquigarrow z \mathbin{@} y]$ *oo* $(ID\ ts) \parallel [y' \rightsquigarrow u] = [x \rightsquigarrow z \mathbin{@} Subst\ y'\ y\ u]$

lemma *CCC-a*: *distinct* $x \Longrightarrow$ *distinct* $y \Longrightarrow$ *set* $y \subseteq$ *set* $x \Longrightarrow$ *set* $z \subseteq$ *set* $x \Longrightarrow$ *set* $u \subseteq$ *set* $y \Longrightarrow$ *TVs* $z = ts$
$\Longrightarrow [x \rightsquigarrow y \mathbin{@} z]$ *oo* $[y \rightsquigarrow u] \parallel (ID\ ts) = [x \rightsquigarrow u \mathbin{@} z]$

lemma *CCC-b*: *distinct* $x \Longrightarrow$ *distinct* $z \Longrightarrow$ *set* $y \subseteq$ *set* $x \Longrightarrow$ *set* $z \subseteq$ *set* $x \Longrightarrow$ *set* $u \subseteq$ *set* z
$\Longrightarrow$ *TVs* $y = ts \Longrightarrow [x \rightsquigarrow y \mathbin{@} z]$ *oo* $(ID\ ts) \parallel [z \rightsquigarrow u] = [x \rightsquigarrow y \mathbin{@} u]$

thm *par-switch-eq-dist*

lemma *in-equiv-CompA-Parallel-b*: *length* $(Out\ A) = 1 \Longrightarrow$ *io-diagram* $A \Longrightarrow$ *io-diagram* $B \Longrightarrow$ *io-diagram* $C \Longrightarrow$ *out* $A \in$ *set* $(In\ B)$
$\Longrightarrow$ *out* $A \notin$ *set* $(In\ C) \Longrightarrow$ *in-equiv* $(CompA\ A\ B \mid\mid\mid CompA\ A\ C)\ (CompA\ A\ (B \mid\mid\mid C))$

lemma *in-equiv-CompA-Parallel-d*: *length* $(Out\ A) = 1 \Longrightarrow$ *io-diagram* $A \Longrightarrow$ *io-diagram* $B \Longrightarrow$ *io-diagram* $C \Longrightarrow$ *out* $A \notin$ *set* $(In\ B) \Longrightarrow$ *out* $A \notin$ *set* $(In\ C) \Longrightarrow$
in-equiv $(CompA\ A\ B \mid\mid\mid CompA\ A\ C)\ (CompA\ A\ (B \mid\mid\mid C))$

lemma *in-equiv-CompA-Parallel*: *deterministic* $(Trs\ A) \Longrightarrow$ *length* $(Out\ A) = 1 \Longrightarrow$ *io-diagram* $A \Longrightarrow$ *io-diagram* $B \Longrightarrow$ *io-diagram* $C \Longrightarrow$
in-equiv $((A \rhd B) \mid\mid\mid (A \rhd C))\ (A \rhd (B \mid\mid\mid C))$

lemma *fb-less-step-compA*: *deterministic* $(Trs\ A) \Longrightarrow$ *length* $(Out\ A) = 1 \Longrightarrow$ *io-diagram* $A \Longrightarrow$ *Type-OK* As
$\Longrightarrow$ *in-equiv* $(Parallel\text{-}list\ (fb\text{-}less\text{-}step\ A\ As))\ (CompA\ A\ (Parallel\text{-}list\ As))$

lemma *switch-eq-Subst*: *distinct* $x \Longrightarrow$ *distinct* $u \Longrightarrow$ *set* $y \subseteq$ *set* $x \Longrightarrow$ *set* $v \subseteq$ *set* $u \Longrightarrow$ *TVs* $x =$ *TVs* u
$\Longrightarrow$ *Subst* $x\ u\ y = v \Longrightarrow [x \rightsquigarrow y] = [u \rightsquigarrow v]$

lemma [*simp*]: *set y* $\subseteq$ *set y1* $\Longrightarrow$ *distinct x1* $\Longrightarrow$ *TVs x1* = *TVs y1* $\Longrightarrow$ *Subst x1 y1* (*Subst y1 x1 y*) = *y*

lemma [*simp*]: *set z* $\subseteq$ *set x* $\Longrightarrow$ *TVs x* = *TVs y* $\Longrightarrow$ *set* (*Subst x y z*) $\subseteq$ *set y*

thm *distinct-Subst*

lemma *distinct-Subst-aa*: $\bigwedge$ *y* .
distinct y $\Longrightarrow$ *length x* = *length y* $\Longrightarrow$ *a* $\notin$ *set y* $\Longrightarrow$ *set z* $\cap$ (*set y* − *set x*) = {} $\Longrightarrow$ *a* $\neq$ *aa*
$\Longrightarrow$ *a* $\notin$ *set z* $\Longrightarrow$ *aa* $\notin$ *set z* $\Longrightarrow$ *distinct z* $\Longrightarrow$ *aa* $\in$ *set x*
$\Longrightarrow$ *subst x y a* $\neq$ *subst x y aa*

lemma *distinct-Subst-ba*: *distinct y* $\Longrightarrow$ *length x* = *length y* $\Longrightarrow$ *set z* $\cap$ (*set y* − *set x*) = {}
$\Longrightarrow$ *a* $\notin$ *set z* $\Longrightarrow$ *distinct z* $\Longrightarrow$ *a* $\notin$ *set y* $\Longrightarrow$ *subst x y a* $\notin$ *set* (*Subst x y z*)

lemma *distinct-Subst-ca*: *distinct y* $\Longrightarrow$ *length x* = *length y* $\Longrightarrow$ *set z* $\cap$ (*set y* − *set x*) = {}
$\Longrightarrow$ *a* $\notin$ *set z* $\Longrightarrow$ *distinct z* $\Longrightarrow$ *a* $\in$ *set x* $\Longrightarrow$ *subst x y a* $\notin$ *set* (*Subst x y z*)

lemma [*simp*]: *set z* $\cap$ (*set y* − *set x*) = {} $\Longrightarrow$ *distinct y* $\Longrightarrow$ *distinct z* $\Longrightarrow$ *length x* = *length y*
$\Longrightarrow$ *distinct* (*Subst x y z*)

lemma *deterministic-Comp*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *deterministic* (*Trs A*) $\Longrightarrow$ *deterministic* (*Trs B*)
$\Longrightarrow$ *deterministic* (*Trs* (*A* ;; *B*))

lemma *deterministic-CompA*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *deterministic* (*Trs A*) $\Longrightarrow$ *deterministic* (*Trs B*)
$\Longrightarrow$ *deterministic* (*Trs* (*A* $\triangleright$ *B*))

lemma *parallel-list-empty*[*simp*]: *parallel-list* [] = *ID* []

lemma *parallel-list-append*: *parallel-list* (*As* @ *Bs*) = *parallel-list As* $\|$ *parallel-list Bs*

lemma *par-swap-aux*: *distinct p* $\Longrightarrow$ *distinct* (*v* @ *u* @ *w*) $\Longrightarrow$

$TI\ A = TVs\ x \Longrightarrow TI\ B = TVs\ y \Longrightarrow TI\ C = TVs\ z \Longrightarrow$
$TO\ A = TVs\ u \Longrightarrow TO\ B = TVs\ v \Longrightarrow TO\ C = TVs\ w \Longrightarrow$
$set\ x \subseteq set\ p \Longrightarrow set\ y \subseteq set\ p \Longrightarrow set\ z \subseteq set\ p \Longrightarrow set\ q \subseteq set\ (u\ @\ v\ @\ w) \Longrightarrow$
$[p \rightsquigarrow x\ @\ y\ @\ z]\ oo\ (A \parallel B \parallel C)\ oo\ [u\ @\ v\ @\ w \rightsquigarrow q] = [p \rightsquigarrow y\ @\ x\ @\ z]\ oo\ (B \parallel A \parallel C)\ oo\ [v\ @\ u\ @\ w \rightsquigarrow q]$

lemma *Type-OK-distinct*: $Type\text{-}OK\ As \Longrightarrow distinct\ As$

lemma *TI-parallel-list-a*: $TI\ (parallel\text{-}list\ As) = concat\ (map\ TI\ As)$

lemma *fb-CompA-aux*: $Type\text{-}OK\ As \Longrightarrow A \in set\ As \Longrightarrow out\ A = a \Longrightarrow a \notin set\ (In\ A) \Longrightarrow$
$InAs = In\ (Parallel\text{-}list\ As) \Longrightarrow OutAs = Out\ (Parallel\text{-}list\ As) \Longrightarrow perm\ (a\ \#\ y)\ InAs \Longrightarrow perm\ (a\ \#\ z)\ OutAs \Longrightarrow$
$InAs' = In\ (Parallel\text{-}list\ (As \ominus [A])) \Longrightarrow$
$fb\ ([a\ \#\ y \rightsquigarrow\ concat\ (map\ In\ As)]\ oo\ parallel\text{-}list\ (map\ Trs\ As)\ oo\ [OutAs \rightsquigarrow a\ \#\ z]) =$
$[y \rightsquigarrow In\ A\ @\ (InAs' \ominus [a])]$
$oo\ (Trs\ A \parallel [(InAs' \ominus [a]) \rightsquigarrow\ (InAs' \ominus [a])])$
$oo\ [a\ \#\ (InAs' \ominus [a]) \rightsquigarrow InAs']\ oo\ Trs\ (Parallel\text{-}list\ (As \ominus [A]))$
$oo\ [OutAs \ominus [a] \rightsquigarrow z]$ (**is** $- \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow fb\ ?Ta = ?Tb$)

lemma [*simp*]: $perm\ (a\ \#\ x)\ (a\ \#\ y) = perm\ x\ y$

lemma *fb-CompA*: $Type\text{-}OK\ As \Longrightarrow A \in set\ As \Longrightarrow out\ A = a \Longrightarrow a \notin set\ (In\ A) \Longrightarrow C = A \rhd (Parallel\text{-}list\ (As \ominus [A])) \Longrightarrow$
$OutAs = Out\ (Parallel\text{-}list\ As) \Longrightarrow perm\ y\ (In\ C) \Longrightarrow perm\ z\ (Out\ C) \Longrightarrow B \in set\ As - \{A\} \Longrightarrow a \in set\ (In\ B) \Longrightarrow$
$fb\ ([a\ \#\ y \rightsquigarrow concat\ (map\ In\ As)]\ oo\ parallel\text{-}list\ (map\ Trs\ As)\ oo\ [OutAs \rightsquigarrow a\ \#\ z]) = [y \rightsquigarrow In\ C]\ oo\ Trs\ C\ oo\ [Out\ C \rightsquigarrow z]$

definition $Deterministic\ As = (\forall\ A \in set\ As\ .\ deterministic\ (Trs\ A))$

lemma *Deterministic-fb-out-less-step*: $Type\text{-}OK\ As \Longrightarrow A \in set\ As \Longrightarrow a = out\ A \Longrightarrow Deterministic\ As \Longrightarrow Deterministic\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)$

lemma *in-equiv-fb-fb-less-step-TO-CHECK*: $loop\text{-}free\ As \Longrightarrow Type\text{-}OK\ As \Longrightarrow Deterministic\ As \Longrightarrow$
$VarFB\ (Parallel\text{-}list\ As) = a\ \#\ L \Longrightarrow Bs = fb\text{-}out\text{-}less\text{-}step\ a\ As$
$\Longrightarrow\ in\text{-}equiv\ (FB\ (Parallel\text{-}list\ As))\ (FB\ (Parallel\text{-}list\ Bs))$

lemma *io-diagram-FB-Parallel-list*: *Type-OK As* $\Longrightarrow$ *io-diagram* (*FB* (*Parallel-list As*))

lemma [*simp*]: *io-diagram A* $\Longrightarrow$ (|*In* = *In A*, *Out* = *Out A*, *Trs* = *Trs A*|) = *A*

thm *loop-free-def*

lemma *io-rel-compA*: *length* (*Out A*) = *1* $\Longrightarrow$ *io-rel* (*CompA A B*) $\subseteq$ *io-rel B* $\cup$ (*io-rel B O io-rel A*)

theorem *loop-free-fb-out-less-step*: *loop-free As* $\Longrightarrow$ *Type-OK As* $\Longrightarrow$ *A* $\in$ *set As* $\Longrightarrow$ *out A* = *a* $\Longrightarrow$ *loop-free* (*fb-out-less-step a As*)

theorem *in-equiv-FB-fb-less-delete*: $\bigwedge$ *As* . *Deterministic As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *Type-OK As* $\Longrightarrow$ *VarFB* (*Parallel-list As*) = *L* $\Longrightarrow$
in-equiv (*FB* (*Parallel-list As*)) (*Parallel-list* (*fb-less L As*)) $\wedge$ *io-diagram* (*Parallel-list* (*fb-less L As*))

lemmas [*simp*] = *diff-emptyset*

lemma [*simp*]: $\bigwedge$ *x* . *distinct x* $\Longrightarrow$ *distinct y* $\Longrightarrow$ *perm* (((*y* $\otimes$ *x*) @ (*x* $\ominus$ *y* $\otimes$ *x*))) *x*

lemma [*simp*]: *io-diagram X* $\Longrightarrow$ *perm* (*VarFB X* @ (*In X* $\ominus$ *VarFB X*)) (*In X*)

lemma *Type-OK-diff* [*simp*]: *Type-OK As* $\Longrightarrow$ *Type-OK* (*As* $\ominus$ *Bs*)

lemma *internal-fb-out-less-step*:
assumes [*simp*]: *loop-free As*
assumes [*simp*]: *Type-OK As*
and [*simp*]: *a* $\in$ *internal As*
shows *internal* (*fb-out-less-step a As*) = *internal As* $-$ {*a*}

end

context *BaseOperationFeedbacklessVars*
begin

lemma [*simp*]: *Type-OK As* $\Longrightarrow$ *a* $\in$ *internal As* $\Longrightarrow$ *out* (*get-comp-out a As*) = *a*

lemma *internal-Type-OK-simp*: *Type-OK As* $\Longrightarrow$ *internal As* = {*a* . (∃ *A* ∈ *set As* . *out A* = *a* ∧ (∃ *B* ∈ *set As*. *a* ∈ *set* (*In B*)))}

thm *Type-OK-def*

lemma *Type-OK-fb-less*: $\bigwedge$ *As* . *Type-OK As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *set x* ⊆ *internal As* $\Longrightarrow$ *Type-OK* (*fb-less x As*)

lemma *fb-Parallel-list-fb-out-less-step*:
assumes [*simp*]: *Type-OK As*
and *Deterministic As*
and *loop-free As*
and *internal*: *a* ∈ *internal As*
and *X*: *X* = *Parallel-list As*
and *Y*: *Y* = (*Parallel-list* (*fb-out-less-step a As*))
and [*simp*]: *perm y* (*In Y*)
and [*simp*]: *perm z* (*Out Y*)
shows *fb* ([*a* # *y* ⇝ *In X*] *oo Trs X oo* [*Out X* ⇝ *a* # *z*]) = [*y* ⇝ *In Y*] *oo Trs Y oo* [*Out Y* ⇝ *z*] **and** *perm* (*a* # *In Y*) (*In X*)

lemma *internal-In-Parallel-list*: *a* ∈ *internal As* $\Longrightarrow$ *a* ∈ *set* (*In* (*Parallel-list As*))

lemma *internal-Out-Parallel-list*: *a* ∈ *internal As* $\Longrightarrow$ *a* ∈ *set* (*Out* (*Parallel-list As*))

theorem *fb-power-internal-fb-less*: $\bigwedge$ *As X Y* . *Deterministic As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *Type-OK As* $\Longrightarrow$ *set L* ⊆ *internal As*
$\Longrightarrow$ *distinct L* $\Longrightarrow$
X = (*Parallel-list As*) $\Longrightarrow$ *Y* = *Parallel-list* (*fb-less L As*) $\Longrightarrow$
(*fb* ^^ *length* (*L*)) ([*L* @ (*In X* ⊖ *L*) ⇝ *In X*] *oo Trs X oo* [*Out X* ⇝ *L* @ (*Out X* ⊖ *L*)]) = [*In X* ⊖ *L* ⇝ *In Y*] *oo Trs Y*
∧ *perm* (*In X* ⊖ *L*) (*In Y*)

thm *fb-power-internal-fb-less*

theorem *FB-fb-less*:
assumes [*simp*]: *Deterministic As*
and [*simp*]: *loop-free As*
and [*simp*]: *Type-OK As*
and [*simp*]: *perm* (*VarFB X*) *L*
and *X*: *X* = (*Parallel-list As*)
and *Y*: *Y* = *Parallel-list* (*fb-less L As*)

shows $(fb \mathbin{\hat{}\hat{}} length\ (L))\ ([L \mathbin{@} InFB\ X \rightsquigarrow In\ X] \mathbin{oo} Trs\ X \mathbin{oo} [Out\ X \rightsquigarrow L \mathbin{@} OutFB\ X]) = [InFB\ X \rightsquigarrow In\ Y] \mathbin{oo} Trs\ Y$
and B: $perm\ (InFB\ X)\ (In\ Y)$

definition *fb-perm-eq* $A = (\forall\ x.\ perm\ x\ (VarFB\ A) \longrightarrow$
$(fb \mathbin{\hat{}\hat{}} length\ (VarFB\ A))\ ([VarFB\ A \mathbin{@} InFB\ A \rightsquigarrow In\ A] \mathbin{oo} Trs\ A \mathbin{oo} [Out\ A \rightsquigarrow VarFB\ A \mathbin{@} OutFB\ A]) =$
$(fb \mathbin{\hat{}\hat{}} length\ (VarFB\ A))\ ([x \mathbin{@} InFB\ A \rightsquigarrow In\ A] \mathbin{oo} Trs\ A \mathbin{oo} [Out\ A \rightsquigarrow x \mathbin{@} OutFB\ A]))$

lemma *fb-perm-eq-simp*: *fb-perm-eq* $A = (\forall\ x.\ perm\ x\ (VarFB\ A) \longrightarrow$
$Trs\ (FB\ A) = (fb \mathbin{\hat{}\hat{}} length\ (VarFB\ A))\ ([x \mathbin{@} InFB\ A \rightsquigarrow In\ A] \mathbin{oo} Trs\ A \mathbin{oo} [Out\ A \rightsquigarrow x \mathbin{@} OutFB\ A]))$

lemma *in-equiv-in-out-equiv*: *io-diagram* $B \Longrightarrow$ *in-equiv* $A\ B \Longrightarrow$ *in-out-equiv* $A\ B$

lemma [*simp*]: $distinct\ (concat\ (map\ f\ As)) \Longrightarrow distinct\ (concat\ (map\ f\ (As \ominus [A])))$

lemma *set-op-list-addvars*: $set\ (op\text{-}list\ []\ op \oplus x) = (\bigcup\ a \in set\ x\ .\ set\ a)$

end

context *BaseOperationFeedbacklessVars*

begin

lemma [*simp*]: $set\ (Out\ A) \subseteq set\ (In\ B) \Longrightarrow Out\ ((A\ ;;\ B)) = Out\ B$

lemma [*simp*]: $set\ (Out\ A) \subseteq set\ (In\ B) \Longrightarrow out\ ((A\ ;;\ B)) = out\ B$

lemma *switch-par-comp3*:
assumes [*simp*]: $distinct\ x$ **and**
[*simp*]: $distinct\ y$
and [*simp*]: $distinct\ z$
and [*simp*]: $distinct\ u$
and [*simp*]: $set\ y \subseteq set\ x$
and [*simp*]: $set\ z \subseteq set\ x$
and [*simp*]: $set\ u \subseteq set\ x$
and [*simp*]: $set\ y' \subseteq set\ y$

and [*simp*]: *set z′* ⊆ *set z*
and [*simp*]: *set u′* ⊆ *set u*
shows [*x* ⇝ *y* @ *z* @ *u*] *oo* [*y* ⇝ *y′*] ‖ [*z* ⇝ *z′*] ‖ [*u* ⇝ *u′*] = [*x* ⇝ *y′* @ *z′* @ *u′*]

lemma *switch-par-comp-Subst3*:
assumes [*simp*]: *distinct x* **and** [*simp*]: *distinct y′* **and** [*simp*]: *distinct z′* **and** [*simp*]: *distinct t′*
and [*simp*]: *set y* ⊆ *set x* **and** [*simp*]: *set z* ⊆ *set x* **and** [*simp*]: *set t* ⊆ *set x*
and [*simp*]: *set u* ⊆ *set y′* **and** [*simp*]: *set v* ⊆ *set z′* **and** [*simp*]: *set w* ⊆ *set t′*
and [*simp*]: *TVs y* = *TVs y′* **and** [*simp*]: *TVs z* = *TVs z′* **and** [*simp*]: *TVs t* = *TVs t′*

shows [*x* ⇝ *y* @ *z* @ *t*] *oo* [*y′* ⇝ *u*] ‖ [*z′* ⇝ *v*] ‖ [*t′* ⇝ *w*] = [*x* ⇝ *Subst y′ y u* @ *Subst z′ z v* @ *Subst t′ t w*]

lemma *Comp-assoc-single*: *length* (*Out A*) = *1* ⟹ *length* (*Out B*) = *1* ⟹ *out A* ≠ *out B* ⟹ *io-diagram A*
⟹ *io-diagram B* ⟹ *io-diagram C* ⟹ *out B* ∉ *set* (*In A*) ⟹
deterministic (*Trs A*) ⟹
out A ∈ *set* (*In B*) ⟹ *out A* ∈ *set* (*In C*) ⟹ *out B* ∈ *set* (*In C*) ⟹ (*A* ;; (*B* ;; *C*)) = (*A* ;; *B* ;; (*A* ;; *C*))

lemma *Comp-commute-aux*:
assumes [*simp*]: *length* (*Out A*) = *1*
and [*simp*]: *length* (*Out B*) = *1*
and [*simp*]: *io-diagram A*
and [*simp*]: *io-diagram B*
and [*simp*]: *io-diagram C*
and [*simp*]: *out B* ∉ *set* (*In A*)
and [*simp*]: *out A* ∉ *set* (*In B*)
and [*simp*]: *out A* ∈ *set* (*In C*)
and [*simp*]: *out B* ∈ *set* (*In C*)
and *Diff*: *out A* ≠ *out B*

shows *Trs* (*A* ;; (*B* ;; *C*)) =
[*In A* ⊕ *In B* ⊕ (*In C* ⊖ [*out A*] ⊖ [*out B*]) ⇝ *In A* @ *In B* @ (*In C* ⊖ [*out A*] ⊖ [*out B*])]
oo Trs A ‖ *Trs B* ‖ [*In C* ⊖ [*out A*] ⊖ [*out B*] ⇝ *In C* ⊖ [*out A*] ⊖ [*out B*]]
oo [*out A* # *out B* # (*In C* ⊖ [*out A*] ⊖ [*out B*]) ⇝ *In C*]
oo Trs C
and *In* (*A* ;; (*B* ;; *C*)) = *In A* ⊕ *In B* ⊕ (*In C* ⊖ [*out A*] ⊖ [*out B*])
and *Out* (*A* ;; (*B* ;; *C*)) = *Out C*

lemma *Comp-commute*:
assumes [*simp*]: *length* (*Out A*) = *1*

and [*simp*]: *length* (*Out B*) = *1*
and [*simp*]: *io-diagram A*
and [*simp*]: *io-diagram B*
and [*simp*]: *io-diagram C*
and [*simp*]: *out B* ∉ *set* (*In A*)
and [*simp*]: *out A* ∉ *set* (*In B*)
and [*simp*]: *out A* ∈ *set* (*In C*)
and [*simp*]: *out B* ∈ *set* (*In C*)
and *Diff*: *out A* ≠ *out B*
shows *in-equiv* (*A* ;; (*B* ;; *C*)) (*B* ;; (*A* ;; *C*))

lemma *CompA-commute-aux-a*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹ *length* (*Out A*) = *1* ⟹ *length* (*Out B*) = *1*
⟹ *out A* ∉ *set* (*Out C*) ⟹ *out B* ∉ *set* (*Out C*)
⟹ *out A* ≠ *out B* ⟹ *out A* ∈ *set* (*In B*) ⟹ *out B* ∉ *set* (*In A*)
⟹ *deterministic* (*Trs A*)
⟹ (*CompA* (*CompA B A*) (*CompA B C*)) = (*CompA* (*CompA A B*) (*CompA A C*))

lemma *CompA-commute-aux-b*: *io-diagram A* ⟹ *io-diagram B* ⟹ *io-diagram C* ⟹ *length* (*Out A*) = *1* ⟹ *length* (*Out B*) = *1*
⟹ *out A* ∉ *set* (*Out C*) ⟹ *out B* ∉ *set* (*Out C*)
⟹ *out A* ≠ *out B* ⟹ *out A* ∉ *set* (*In B*) ⟹ *out B* ∉ *set* (*In A*)
⟹ *in-equiv* (*CompA* (*CompA B A*) (*CompA B C*)) (*CompA* (*CompA A B*) (*CompA A C*))

fun *In-Equiv* :: ((*'var*, *'a*) *Dgr*) *list* ⇒ ((*'var*, *'a*) *Dgr*) *list* ⇒ *bool* **where**
In-Equiv [] [] = *True* |
In-Equiv (*A* # *As*) (*B*# *Bs*) = (*in-equiv A B* ∧ *In-Equiv As Bs*) |
In-Equiv - - = *False*

thm *internal-def*

thm *fb-out-less-step-def*
thm *fb-less-step-def*

thm *CompA-commute-aux-b*
thm *CompA-commute-aux-a*

lemma *CompA-commute*:
assumes [*simp*]: *io-diagram A*
and [*simp*]: *io-diagram B*
and [*simp*]: *io-diagram C*
and [*simp*]: *length* (*Out A*) = *1*
and [*simp*]: *length* (*Out B*) = *1*
and [*simp*]: *out A* ∉ *set* (*Out C*)
and [*simp*]: *out B* ∉ *set* (*Out C*)

and [*simp*]: *out A* ≠ *out B*
and [*simp*]: *deterministic* (*Trs A*)
and [*simp*]: *deterministic* (*Trs B*)
and *A*: (*out A* ∈ *set* (*In B*) ⟹ *out B* ∉ *set* (*In A*))
shows *in-equiv* (*CompA* (*CompA B A*) (*CompA B C*)) (*CompA* (*CompA A B*) (*CompA A C*))

lemma *In-Equiv-CompA-twice*: (⋀ *C* . *C* ∈ *set As* ⟹ *io-diagram C* ∧ *out A* ∉ *set* (*Out C*) ∧ *out B* ∉ *set* (*Out C*)) ⟹ *io-diagram A* ⟹ *io-diagram B*
⟹ *length* (*Out A*) = *1* ⟹ *length* (*Out B*) = *1* ⟹ *out A* ≠ *out B*
⟹ *deterministic* (*Trs A*) ⟹ *deterministic* (*Trs B*)
⟹ (*out A* ∈ *set* (*In B*) ⟹ *out B* ∉ *set* (*In A*))
⟹ *In-Equiv* (*map* (*CompA* (*CompA B A*)) (*map* (*CompA B*) *As*)) (*map* (*CompA* (*CompA A B*)) (*map* (*CompA A*) *As*))

thm *Type-OK-def*
thm *Deterministic-def*
thm *internal-def*
thm *fb-out-less-step-def*

thm *mem-get-other-out*

thm *mem-get-comp-out*

thm *comp-out-in*

lemma *map-diff*: (⋀ *b* . *b* ∈ *set x* ⟹ *b* ≠ *a* ⟹ *f b* ≠ *f a*) ⟹ *map f x* ⊖ [*f a*] = *map f* (*x* ⊖ [*a*])

lemma *In-Equiv-fb-out-less-step-commute*: *Type-OK As* ⟹ *Deterministic As* ⟹ *x* ∈ *internal As* ⟹ *y* ∈ *internal As* ⟹ *x* ≠ *y* ⟹ *loop-free As*
⟹ *In-Equiv* (*fb-out-less-step x* (*fb-out-less-step y As*)) (*fb-out-less-step y* (*fb-out-less-step x As*))

lemma [*simp*]: *Type-OK As* ⟹ *In-Equiv As As*

lemma *fb-less-append*: ⋀ *As* . *fb-less* (*x* @ *y*) *As* = *fb-less y* (*fb-less x As*)

thm *in-equiv-tran*

lemma *In-Equiv-trans*: ⋀ *Bs Cs* . *Type-OK Cs* ⟹ *In-Equiv As Bs* ⟹ *In-Equiv Bs Cs* ⟹ *In-Equiv As Cs*

lemma *In-Equiv-exists*: $\bigwedge$ *Bs* . *In-Equiv As Bs* $\Longrightarrow$ *A* $\in$ *set As* $\Longrightarrow$ $\exists$ *B* $\in$ *set Bs* . *in-equiv A B*

lemma *In-Equiv-Type-OK*: $\bigwedge$*Bs* . *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *Type-OK As*

lemma *In-Equiv-internal-aux*: *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *internal As* $\subseteq$ *internal Bs*

lemma *In-Equiv-sym*: $\bigwedge$ *Bs* . *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *In-Equiv Bs As*

lemma *In-Equiv-internal*: *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *internal As* = *internal Bs*

lemma *in-equiv-CompA*: *in-equiv A A'* $\Longrightarrow$ *in-equiv B B'* $\Longrightarrow$ *io-diagram A'* $\Longrightarrow$ *io-diagram B'* $\Longrightarrow$ *in-equiv* (*CompA A B*) (*CompA A' B'*)

lemma *In-Equiv-fb-less-step-cong*: $\bigwedge$ *Bs* . *Type-OK Bs* $\Longrightarrow$ *in-equiv A B* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *In-Equiv As Bs*
$\Longrightarrow$ *In-Equiv* (*fb-less-step A As*) (*fb-less-step B Bs*)

lemma *In-Equiv-append*: $\bigwedge$ *As'* . *In-Equiv As As'* $\Longrightarrow$ *In-Equiv Bs Bs'* $\Longrightarrow$ *In-Equiv* (*As* @ *Bs*) (*As'* @ *Bs'*)

lemma *In-Equiv-split*: $\bigwedge$ *Bs* . *In-Equiv As Bs* $\Longrightarrow$ *A* $\in$ *set As*
$\Longrightarrow$ $\exists$ *B As' As'' Bs' Bs''* . *As* = *As'* @ *A* # *As''* $\wedge$ *Bs* = *Bs'* @ *B* # *Bs''* $\wedge$ *in-equiv A B* $\wedge$ *In-Equiv As' Bs'* $\wedge$ *In-Equiv As'' Bs''*

lemma *In-Equiv-fb-out-less-step-cong*:
assumes [*simp*]: *Type-OK Bs*
and *In-Equiv As Bs*
and *internal*: *a* $\in$ *internal As*
shows *In-Equiv* (*fb-out-less-step a As*) (*fb-out-less-step a Bs*)

lemma *In-Equiv-IO-Rel*: $\bigwedge$ *Bs* . *In-Equiv As Bs* $\Longrightarrow$ *IO-Rel Bs* = *IO-Rel As*

lemma *In-Equiv-loop-free*: *In-Equiv As Bs* $\Longrightarrow$ *loop-free Bs* $\Longrightarrow$ *loop-free As*

lemma *loop-free-fb-out-less-step-internal*:
assumes [*simp*]: *loop-free As*
and [*simp*]: *Type-OK As*
and *a* $\in$ *internal As*
shows *loop-free* (*fb-out-less-step a As*)

lemma *loop-free-fb-less-internal*:
$\bigwedge$ *As* . *loop-free As* $\Longrightarrow$ *Type-OK As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *loop-free* (*fb-less x As*)

lemma *In-Equiv-fb-less-cong*: $\bigwedge$ *As Bs* . *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *loop-free Bs* $\Longrightarrow$ *In-Equiv* (*fb-less x As*) (*fb-less x Bs*)

thm *Type-OK-fb-out-less-step-new*

thm *Type-OK-fb-less*

lemma *Type-OK-fb-less-delete*: $\bigwedge$ *As* . *Type-OK As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *Type-OK* (*fb-less x As*)

thm *Deterministic-fb-out-less-step*

thm *internal-fb-out-less-step*

lemma *internal-fb-less*:
$\bigwedge$ *As* . *loop-free As* $\Longrightarrow$ *Type-OK As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *internal* (*fb-less x As*) = *internal As* $-$ *set x*

thm *Deterministic-fb-out-less-step*

lemma *Deterministic-fb-out-less-step-internal*:
assumes [*simp*]: *Type-OK As*
and *Deterministic As*
and *internal*: *a* $\in$ *internal As*
shows *Deterministic* (*fb-out-less-step a As*)

lemma *Deterministic-fb-less-internal*: $\bigwedge$ *As* . *Type-OK As* $\Longrightarrow$ *Deterministic As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *Deterministic* (*fb-less x As*)

lemma *In-Equiv-fb-less-Cons*: $\bigwedge$ *As* . *Type-OK As* $\Longrightarrow$ *Deterministic As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *a* $\in$ *internal As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct* (*a* # *x*)

$\Longrightarrow$ *In-Equiv* (*fb-less* (*a* # *x*) *As*) (*fb-less* (*x* @ [*a*]) *As*)

theorem *In-Equiv-fb-less*: $\bigwedge$ *y As* . *Type-OK As* $\Longrightarrow$ *Deterministic As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *set x* $\subseteq$ *internal As* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *perm x y*
$\Longrightarrow$ *In-Equiv* (*fb-less x As*) (*fb-less y As*)

lemma [*simp*]: *in-equiv* □ □

lemma *in-equiv-Parallel-list*: $\bigwedge$ *Bs* . *Type-OK Bs* $\Longrightarrow$ *In-Equiv As Bs* $\Longrightarrow$ *in-equiv* (*Parallel-list As*) (*Parallel-list Bs*)

thm *FB-fb-less*

lemma [*simp*]: *io-diagram A* $\Longrightarrow$ *distinct* (*VarFB A*)

lemma [*simp*]: *io-diagram A* $\Longrightarrow$ *distinct* (*InFB A*)

theorem *fb-perm-eq-Parallel-list*:
assumes [*simp*]: *Type-OK As*
and [*simp*]: *Deterministic As*
and [*simp*]: *loop-free As*
shows *fb-perm-eq* (*Parallel-list As*)

theorem *FeedbackSerial-Feedbackless*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*In A*) $\cap$ *set* (*In B*) = {} (∗*required*∗)
$\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$ *fb-perm-eq* (*A* ||| *B*) $\Longrightarrow$ *FB* (*A* ||| *B*) = *FB* (*FB* (*A*) ;; *FB* (*B*))

declare *io-diagram-distinct* [*simp del*]

lemma *in-out-equiv-FB-less*: *io-diagram B* $\Longrightarrow$ *in-out-equiv A B* $\Longrightarrow$ *fb-perm-eq A* $\Longrightarrow$ *in-out-equiv* (*FB A*) (*FB B*)

lemma [*simp*]: *io-diagram A* $\Longrightarrow$ *distinct* (*OutFB A*)

end

end

8 Refinement Calculus and Monotonic Predicate Transformers

theory *Refinement* **imports** *Main*
begin

In this section we introduce the basics of refinement calculus [**?**]. Part of this theory is a reformulation of some definitions from [**?**], but here they are given for predicates, while [**?**] uses sets.

notation
bot ($\bot$) **and**
top ($\top$) **and**
inf (**infixl** $\sqcap$ *70*)
and *sup* (**infixl** $\sqcup$ *65*)

8.1 Basic predicate transformers

definition
demonic :: (*'a* => *'b*::*lattice*) => *'b* => *'a* ⇒ *bool* ([: - :] [*0*] *1000*) **where**
[:*Q*:] *p s* = (*Q s* ≤ *p*)

definition
assert::*'a*::*semilattice-inf* => *'a* => *'a* ({. - .} [*0*] *1000*) **where**
{.*p*.} *q* ≡ *p* ⊓ *q*

definition
assume::(*'a*::*boolean-algebra*) => *'a* => *'a* ([. - .] [*0*] *1000*) **where**
[.*p*.] *q* ≡ (−*p* ⊔ *q*)

definition
angelic :: (*'a* ⇒ *'b*::{*semilattice-inf*,*order-bot*}) ⇒ *'b* ⇒ *'a* ⇒ *bool* ({: - :} [*0*] *1000*) **where**
{:*Q*:} *p s* = (*Q s* ⊓ *p* ≠ ⊥)

syntax
-*assert* :: *patterns* => *logic* => *logic* ((*1*{.-.-.}))
translations
-*assert x P* == *CONST assert* (-*abs x P*)

syntax
-*demonic* :: *patterns* => *patterns* => *logic* => *logic* (([:-⇝-.-:]))
translations
-*demonic x y t* == (*CONST demonic* (-*abs x* (-*abs y t*)))

syntax
-*angelic* :: *patterns* => *patterns* => *logic* => *logic* (({:- ⇝ -.-:}))
translations
-*angelic x y t* == (*CONST angelic* (-*abs x* (-*abs y t*)))

lemma *assert-o-def*: $\{.f\ o\ g.\} = \{.(\lambda\ x\ .\ f\ (g\ x)).\}$

lemma *demonic-demonic*: $[:r:]\ o\ [:r':] = [:r\ OO\ r':]$

lemma *assert-demonic-comp*: $\{.p.\}\ o\ [:r:]\ o\ \{.p'.\}\ o\ [:r':] =$
$\{.x\ .\ p\ x \wedge (\forall\ y\ .\ r\ x\ y \longrightarrow p'\ y).\}\ o\ [:r\ OO\ r':]$

lemma *demonic-assert-comp*: $[:r:]\ o\ \{.p.\} = \{.x.(\forall\ y\ .\ r\ x\ y \longrightarrow p\ y).\}\ o\ [:r:]$

lemma *assert-assert-comp*: $\{.p::'a::lattice.\}\ o\ \{.p'.\} = \{.p \sqcap p'.\}$

lemma *assert-assert-comp-pred*: $\{.p.\}\ o\ \{.p'.\} = \{.x\ .\ p\ x \wedge p'\ x.\}$

lemma *demonic-refinement*: $r' \leq r \Longrightarrow [:r:] \leq [:r':]$

definition *inpt r x* $= (\exists\ y\ .\ r\ x\ y)$

definition *trs* :: $('a => 'b \Rightarrow bool) => ('b \Rightarrow bool) => 'a \Rightarrow bool$ ($\{: - :]$ [0] *1000*) **where**
trs r $= \{.\ inpt\ r.\}\ o\ [:r:]$

syntax
-trs :: *patterns => patterns => logic => logic* (($\{:\text{-}\rightsquigarrow\text{-}.\text{-}:]$))
translations
-trs x y t == (*CONST trs* (*-abs x* (*-abs y t*)))

lemma *assert-demonic-prop*: $\{.p.\}\ o\ [:r:] = \{.p.\}\ o\ [:(\lambda\ x\ y\ .\ p\ x) \sqcap r:]$

lemma *trs-trs*: (*trs r*) *o* (*trs r'*)
= *trs* $((\lambda\ s\ t.\ (\forall\ s'\ .\ r\ s\ s' \longrightarrow (inpt\ r'\ s')))) \sqcap (r\ OO\ r'))$ (**is** *?S* = *?T*)

lemma *prec-inpt-equiv*: $p \leq inpt\ r \Longrightarrow r' = (\lambda\ x\ y\ .\ p\ x \wedge r\ x\ y) \Longrightarrow \{.p.\}\ o\ [:r:]$ $= \{:r':]$

lemma *assert-demonic-refinement*: $(\{.p.\}\ o\ [:r:] \leq \{.p'.\}\ o\ [:r':]) = (p \leq p' \wedge (\forall$ $x\ .\ p\ x \longrightarrow r'\ x \leq r\ x))$

lemma *spec-demonic-refinement*: $(\{.p.\}\ o\ [:r:] \leq [:r':]) = (\forall\ x\ .\ p\ x \longrightarrow r'\ x \leq r$ $x)$

lemma *trs-refinement*: $(trs\ r \leq trs\ r') = ((\forall\ x\ .\ inpt\ r\ x \longrightarrow inpt\ r'\ x) \wedge (\forall\ x\ .$ $inpt\ r\ x \longrightarrow r'\ x \leq r\ x))$

lemma *demonic-choice*: $[:r:] \sqcap [:r':] = [:r \sqcup r':]$

lemma *spec-demonic-choice*: $(\{.p.\}\ o\ [:r:]) \sqcap (\{.p'.\}\ o\ [:r':]) = (\{.p \sqcap p'.\}\ o\ [:r \sqcup r':])$

lemma *trs-demonic-choice*: $trs\ r \sqcap trs\ r' = trs\ ((\lambda\ x\ y\ .\ inpt\ r\ x \wedge inpt\ r'\ x) \sqcap (r \sqcup r'))$

lemma *spec-angelic*: $p \sqcap p' = \bot \Longrightarrow (\{.p.\}\ o\ [:r:]) \sqcup (\{.p'.\}\ o\ [:r':])$
$= \{.p \sqcup p'.\}\ o\ [:(\lambda\ x\ y\ .\ p\ x \longrightarrow r\ x\ y) \sqcap ((\lambda\ x\ y\ .\ p'\ x \longrightarrow r'\ x\ y)):]$

8.2 Conjunctive predicate transformers

definition *conjunctive* $(S::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice) = (\forall\ Q\ .\ S\ (Inf\ Q) = INFIMUM\ Q\ S)$

definition *sconjunctive* $(S::'a::complete\text{-}lattice \Rightarrow 'b::complete\text{-}lattice) = (\forall\ Q\ .\ (\exists\ x\ .\ x \in Q) \longrightarrow S\ (Inf\ Q) = INFIMUM\ Q\ S)$

lemma *conjunctive-sconjunctive*[*simp*]: $conjunctive\ S \Longrightarrow sconjunctive\ S$

lemma [*simp*]: $conjunctive\ \top$

lemma *conjuncive-demonic* [*simp*]: $conjunctive\ [:r:]$

lemma *sconjunctive-assert* [*simp*]: $sconjunctive\ \{.p.\}$

lemma *sconjunctive-simp*: $x \in Q \Longrightarrow sconjunctive\ S \Longrightarrow S\ (Inf\ Q) = INFIMUM\ Q\ S$

lemma *sconjunctive-INF-simp*: $x \in X \Longrightarrow sconjunctive\ S \Longrightarrow S\ (INFIMUM\ X\ Q) = INFIMUM\ (Q`X)\ S$

lemma *demonic-comp* [*simp*]: $sconjunctive\ S \Longrightarrow sconjunctive\ S' \Longrightarrow sconjunctive\ (S\ o\ S')$

lemma *conjunctive-INF*[*simp*]:$conjunctive\ S \Longrightarrow S\ (INFIMUM\ X\ Q) = (INFIMUM\ X\ (S\ o\ Q))$

lemma *conjunctive-simp*: $conjunctive\ S \Longrightarrow\ S\ (Inf\ Q) = INFIMUM\ Q\ S$

lemma *conjunctive-monotonic* [*simp*]: $sconjunctive\ S \Longrightarrow mono\ S$

definition $grd\ S = -\ S\ \bot$

lemma *grd-demonic*: $grd\ [:r:] = inpt\ r$

lemma $(S::'a::bot \Rightarrow 'b::boolean\text{-}algebra) \leq S' \Longrightarrow grd\ S' \leq grd\ S$

lemma [*simp*]: *inpt* ($\lambda x\ y.\ p\ x \wedge r\ x\ y$) = $p \sqcap$ *inpt* r

lemma [*simp*]: $p \leq$ *inpt* $r \Longrightarrow p \sqcap$ *inpt* $r = p$

lemma *grd-spec*: *grd* ({.p.} o [:r:]) = $-p \sqcup$ *inpt* r

definition *fail* $S = -(S\ \top)$
definition *term* $S = (S\ \top)$
definition *prec* $S = -$ (*fail* S)
definition *rel* $S = (\lambda\ x\ y\ .\ \neg\ S\ (\lambda\ z\ .\ y \neq z)\ x)$

lemma *rel-spec*: *rel* ({.p.} o [:r:]) $x\ y = (p\ x \longrightarrow r\ x\ y)$

lemma *prec-spec*: *prec* ({.p.} o [:r::'a⇒'b⇒*bool*:]) = p

lemma *fail-spec*: *fail* ({.p.} o [:(r::'a⇒'b::*boolean-algebra*):]) = $-p$

lemma [*simp*]: *prec* ({.p.} o [:(r::'a⇒'b::*boolean-algebra*):]) = p

lemma [*simp*]: *prec* (T::('a::*boolean-algebra* ⇒ 'b::*boolean-algebra*)) = $\top \Longrightarrow$ *prec* (S o T) = *prec* S

lemma [*simp*]: *prec* [:r::'a ⇒ 'b::*boolean-algebra*:] = $\top$

lemma *prec-rel*: {. p .} ∘ [: $\lambda x\ y.\ p\ x \wedge r\ x\ y$:] = {.p.} o [:r:]

definition *Fail* = $\bot$

lemma *Fail-assert-demonic*: *Fail* = {.$\bot$.} o [:r:]

lemma *Fail-assert*: *Fail* = {.$\bot$.} o [:$\bot$:]

lemma *fail-comp*[*simp*]: $\bot\ o\ S = \bot$

lemma *Fail-fail*: *mono* (S::'a::*boolean-algebra* ⇒ 'b::*boolean-algebra*) $\Longrightarrow$ (S = *Fail*) = (*fail* $S = \top$)

lemma *sconjunctive-spec*: *sconjunctive* $S \Longrightarrow S$ = {.*prec* S.} o [:*rel* S:]

definition *non-magic* $S = (S\ \bot = \bot)$

lemma *non-magic-spec*: *non-magic* ({.p.} o [:r:]) = ($p \leq$ *inpt* r)

lemma *sconjunctive-non-magic*: *sconjunctive* $S \Longrightarrow$ *non-magic* S = (*prec* $S \leq$ *inpt*

(*rel S*))

definition *implementable S* = (*sconjunctive S* ∧ *non-magic S*)

lemma *implementable-spec*: *implementable S* ⟹ ∃ *p r* . *S* = {.*p*.} *o* [:*r*:] ∧ *p* ≤ *inpt r*

definition *Skip* = (*id*:: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*))

lemma *assert-true-skip*: {.⊤::*'a* ⇒ *bool*.} = *Skip*

lemma *skip-comp* [*simp*]: *Skip o S* = *S*

lemma *comp-skip*[*simp*]:*S o Skip* = *S*

lemma *assert-rel-skip*[*simp*]: {. λ (*x*, *y*) . *True* .} = *Skip*

lemma [*simp*]: *mono S* ⟹ *mono S'* ⟹ *mono* (*S o S'*)

lemma [*simp*]: *mono* {.*p*::(*'a* ⇒ *bool*).}

lemma [*simp*]: *mono* [:*r*::(*'a* ⇒ *'b* ⇒ *bool*):]

lemma *assert-true-skip-a*: {. *x* . *True* .} = *Skip*

lemma *assert-false-fail*: {.⊥::*'a*::*boolean-algebra*.} = ⊥

lemma *magoc-comp*[*simp*]: ⊤ *o S* = ⊤

lemma *left-comp*: *T o U* = *T' o U'* ⟹ *S o T o U* = *S o T' o U'*

lemma *assert-demonic*: {.*p*.} *o* [:*r*:] = {.*p*.} *o* [:*x* ⇝ *y* . *p x* ∧ *r x y*:]

lemma *trs r* ⊓ *trs r'* = *trs* (λ *x y* . *inpt r x* ∧ *inpt r' x* ∧ (*r x y* ∨ *r' x y*))

lemma *mono-assert*[*simp*]: *mono* {.*p*.}

lemma *mono-assume*[*simp*]: *mono* [.*p*.]

lemma *mono-demonic*[*simp*]: *mono* [:*r*:]

lemma *mono-comp-a*[*simp*]: *mono S* ⟹ *mono T* ⟹ *mono* (*S o T*)

lemma *mono-demonic-choice*[*simp*]: *mono S* ⟹ *mono T* ⟹ *mono* (*S* ⊓ *T*)

lemma *mono-Skip*[*simp*]: *mono Skip*

lemma *mono-comp*: $mono\ S \Longrightarrow S \leq S' \Longrightarrow T \leq T' \Longrightarrow S\ o\ T \leq S'\ o\ T'$

lemma *sconjunctive-simp-a*: $sconjunctive\ S \Longrightarrow prec\ S = p \Longrightarrow rel\ S = r \Longrightarrow S = \{.p.\}\ o\ [:r:]$

lemma *sconjunctive-simp-b*: $sconjunctive\ S \Longrightarrow prec\ S = \top \Longrightarrow rel\ S = r \Longrightarrow S = [:r:]$

lemma *sconj-Fail*[*simp*]: *sconjunctive Fail*

lemma *sconjunctive-simp-c*: $sconjunctive\ (S::('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool) \Longrightarrow prec\ S = \bot \Longrightarrow S = Fail$

lemma *demonic-eq-skip*: $[:\ op = :] = Skip$

definition $Havoc = [:\top:]$

definition $Magic = [:\bot::'a \Rightarrow 'b::boolean\text{-}algebra:]$

lemma *Magic-top*: $Magic = \top$

lemma [*simp*]: $Magic \neq Fail$

lemma *Havoc-Fail*[*simp*]: $Havoc\ o\ (Fail::'a \Rightarrow 'b \Rightarrow bool) = Fail$

lemma *demonic-havoc*: $[:\ \lambda x\ (x',\ y).\ True\ :] = Havoc$

lemma [*simp*]: *mono Magic*

lemma *demonic-false-magic*: $[:\ \lambda(x,\ y)\ (u,\ v).\ False\ :] = Magic$

lemma *demonic-magic*[*simp*]: $[:r:]\ o\ Magic = Magic$

lemma *magic-comp*[*simp*]: $Magic\ o\ S = Magic$

lemma *hvoc-magic*[*simp*]: $Havoc \circ Magic = Magic$

lemma $Havoc\ \top = \top$

lemma *Skip-id*[*simp*]: $Skip\ p = p$

lemma *demonic-pair-skip*: $[:\ x,\ y \rightsquigarrow u,\ v.\ x = u \wedge y = v\ :] = Skip$

lemma *comp-demonic-demonic*: $S\ o\ [:r:]\ o\ [:r':] = S\ o\ [:r\ OO\ r':]$

lemma *comp-demonic-assert*: $S\ o\ [:r:]\ o\ \{.p.\} = S\ o\ \{.\ x.\ \forall y\ .\ r\ x\ y \longrightarrow p\ y\ .\}\ o\ [:r:]$

lemma *assert-demonic-eq-demonic*: $(\{.p.\}\ o\ [:r::'a \Rightarrow 'b \Rightarrow bool:] = [:r:]) = (\forall\ x\ .\ p\ x)$

lemma *trs-inpt-top*: $inpt\ r = \top \Longrightarrow trs\ r = [:r:]$

8.3 Product and Fusion of predicate transformers

In this section we define the fusion and product operators from [**?**]. The fusion of two programs S and T is intuitively equivalent with the parallel execution of the two programs. If S and T assign nondeterministically some value to some program variable x, then the fusion of S and T will assign a value to x which can be assigned by both S and T.

definition *fusion* :: $((('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)) \Rightarrow (('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)) \Rightarrow (('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool))$ (**infixl** $\parallel$ *70*) **where**
$(S \parallel S')\ q\ x = (\exists\ (p::'a \Rightarrow bool)\ p'\ .\ p \sqcap p' \leq q \wedge S\ p\ x \wedge S'\ p'\ x)$

lemma *fusion-demonic*: $[:r:] \parallel [:r':] = [:r \sqcap r':]$

lemma *fusion-spec*: $(\{.p.\} \circ [:r:]) \parallel (\{.p'.\} \circ [:r':]) = (\{.p \sqcap p'.\} \circ [:r \sqcap r':])$

lemma *fusion-assoc*: $S \parallel (T \parallel U) = (S \parallel T) \parallel U$

lemma *fusion-refinement*: $S \leq T \Longrightarrow S' \leq T' \Longrightarrow S \parallel S' \leq T \parallel T'$

lemma *conjunctive* $S \Longrightarrow S \parallel \top = \top$

lemma *fusion-spec-local*: $a \in init \Longrightarrow ([:\ x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y\ :] \circ \{.p.\} \circ [:r:]) \parallel (\{.p'.\} \circ [:r':])$
$= [:\ x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y\ :] \circ \{.u,x\ .\ p\ (u,\ x) \wedge p'\ x.\} \circ [:u,\ x \rightsquigarrow y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ y:]$ (**is** $?p \Longrightarrow ?S = ?T$)

lemma *fusion-demonic-idemp* [*simp*]: $[:r:] \parallel [:r:] = [:r:]$

lemma *fusion-spec-local-a*: $a \in init \Longrightarrow ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel [:r':]$
$= ([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:u,\ x \rightsquigarrow y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ y:])$

lemma *fusion-local-refinement*:
$a \in init \Longrightarrow (\bigwedge x\ u\ y\ .\ u \in init \Longrightarrow p'\ x \Longrightarrow r\ (u,\ x)\ y \Longrightarrow r'\ x\ y) \Longrightarrow$
$\{.p'.\}\ o\ (([:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]) \parallel [:r':]) \leq [:x \rightsquigarrow u,\ y\ .\ u \in init \wedge x = y:] \circ \{.p.\} \circ [:r:]$

lemma *fusion-spec-demonic*: $(\{.p.\}\ o\ [:r:]) \parallel [:r':] = \{.p.\}\ o\ [:r \sqcap r':]$

definition *Fusion* :: $('c \Rightarrow (('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool))) \Rightarrow (('a \Rightarrow bool) \Rightarrow ('b$

$\Rightarrow$ *bool*)) **where**
Fusion S q x = (∃ (*p*::*'c* ⇒ *'a* ⇒ *bool*) . (*INF c* . *p c*) ≤ *q* ∧ (∀ *c* . (*S c*) (*p c*) *x*))

lemma *Fusion-spec*: *Fusion* (λ *n* . {.*p n*.} ∘ [:*r n*:]) = ({.*INFIMUM UNIV p*.} ∘ [:*INFIMUM UNIV r*:])

lemma *Fusion-demonic*: *Fusion* (λ *n* . [:*r n*:]) = [:*INF n* . *r n*:]

lemma *Fusion-refinement*: (⋀ *i* . *S i* ≤ *T i*) ⟹ *Fusion S* ≤ *Fusion T*

lemma *mono-fusion*[*simp*]: *mono* (*S* ‖ *T*)

lemma *mono-Fusion*: *mono* (*Fusion S*)

definition *prod-pred A B* = (λ(*a*, *b*). *A a* ∧ *B b*)
definition *Prod* :: ((*'a* ⇒ *bool*) ⇒ (*'b* ⇒ *bool*)) ⇒ ((*'c* ⇒ *bool*) ⇒ (*'d* ⇒ *bool*)) ⇒ ((*'a* × *'c* ⇒ *bool*) ⇒ (*'b* × *'d* ⇒ *bool*))
(**infixr** ⋆⋆ *70*)
where
(*S* ⋆⋆ *T*) *q* = (λ (*x*, *y*) . ∃ *p p'* . *prod-pred p p'* ≤ *q* ∧ *S p x* ∧ *T p' y*)

lemma *mono-prod*[*simp*]: *mono* (*S* ⋆⋆ *T*)

lemma *Prod-spec*: ({.*p*.} *o* [:*r*:]) ⋆⋆ ({.*p'*.} *o* [:*r'*:]) = {.*x*,*y* . *p x* ∧ *p' y*.} *o* [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *r' y v*:]

lemma *Prod-demonic*: [:*r*:] ⋆⋆ [:*r'*:] = [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *r' y v*:]

lemma *Prod-spec-Skip*: ({.*p*.} *o* [:*r*:]) ⋆⋆ *Skip* = {.*x*,*y* . *p x*.} *o* [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *v* = *y*:]

lemma *Prod-Skip-spec*: *Skip* ⋆⋆ ({.*p*.} *o* [:*r*:]) = {.*x*,*y* . *p y*.} *o* [:*x*, *y* ⇝ *u*, *v* . *x* = *u* ∧ *r y v*:]

lemma *Prod-skip-demonic*: *Skip* ⋆⋆ [:*r*:] = [:*x*, *y* ⇝ *u*, *v* . *x* = *u* ∧ *r y v*:]

lemma *Prod-demonic-skip*: [:*r*:] ⋆⋆ *Skip* = [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *y* = *v*:]

lemma *Prod-spec-demonic*: ({.*p*.} *o* [:*r*:]) ⋆⋆ [:*r'*:] = {.*x*, *y* . *p x*.} *o* [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *r' y v*:]

lemma *Prod-demonic-spec*: [:*r*:] ⋆⋆ ({.*p*.} *o* [:*r'*:]) = {.*x*, *y* . *p y*.} *o* [:*x*, *y* ⇝ *u*, *v* . *r x u* ∧ *r' y v*:]

lemma *pair-eq-demonic-skip*: [: λ(*x*, *y*) (*u*, *v*). *x* = *u* ∧ *v* = *y* :] = *Skip*

lemma *Prod-assert-skip*: {.*p*.} ⋆⋆ *Skip* = {.*x*,*y* . *p x*.}

lemma *Prod-skip-assert*: $Skip \mathbin{**} \{.p.\} = \{.x,y\ .\ p\ y.\}$

lemma *fusion-comute*: $S \parallel T = T \parallel S$

lemma *fusion-mono1*: $S \leq S' \Longrightarrow S \parallel T \leq S' \parallel T$

lemma *prod-mono1*: $S \leq S' \Longrightarrow S \mathbin{**} T \leq S' \mathbin{**} T$

lemma *prod-mono2*: $S \leq S' \Longrightarrow T \mathbin{**} S \leq T \mathbin{**} S'$

lemma *Prod-fusion*: $S \mathbin{**} T = ([:x,y \rightsquigarrow x'\ .\ x = x':]\ o\ S\ o\ [:x \rightsquigarrow x',\ y\ .\ x = x':]) \parallel ([:x,\ y \rightsquigarrow y'\ .\ y = y':]\ o\ T\ o\ [:y \rightsquigarrow x,\ y'\ .\ y = y':])$

lemma *refin-comp-right*: $(S::'a \Rightarrow 'b::order) \leq T \Longrightarrow S\ o\ X \leq T\ o\ X$

lemma *refin-comp-left*: $mono\ X \Longrightarrow (S::'a \Rightarrow 'b::order) \leq T \Longrightarrow X\ o\ S\ \leq X\ o\ T$

lemma *mono-angelic*[*simp*]: $mono\ \{:r:\}$

lemma [*simp*]: $Skip \mathbin{**} Magic = Magic$

lemma [*simp*]: $S \mathbin{**} Fail = Fail$

lemma [*simp*]: $Fail \mathbin{**} S = Fail$

lemma *demonic-conj*: $[:(r::'a \Rightarrow 'b \Rightarrow bool):]\ o\ (S \sqcap S') = ([:r:]\ o\ S) \sqcap ([:r:]\ o\ S')$

lemma *demonic-assume*: $[:r:]\ o\ [.p.] = [:x \rightsquigarrow y\ .\ r\ x\ y \wedge p\ y:]$

lemma *assume-demonic*: $[.p.]\ o\ [:r:] = [:x \rightsquigarrow y\ .\ p\ x \wedge r\ x\ y:]$

lemma [*simp*]: $(Fail::'a::boolean\text{-}algebra) \leq S$

lemma *prod-skip-skip*[*simp*]: $Skip \mathbin{**} Skip = Skip$

lemma *fusion-prod*: $S \parallel T = [:x \rightsquigarrow y,\ z\ .\ x = y \wedge x = z:]\ o\ Prod\ S\ T\ o\ [:y\ ,\ z \rightsquigarrow x\ .\ y = x \wedge z = x:]$

lemma [*simp*]: $prec\ S = \top \Longrightarrow prec\ T = \top \Longrightarrow prec\ (S \mathbin{**} T) = \top$

lemma *prec-skip*[*simp*]: $prec\ Skip = (\top::'a \Rightarrow bool)$

lemma [*simp*]: $prec\ S = \top \Longrightarrow prec\ T = \top \Longrightarrow prec\ (S \parallel T) = \top$

8.4 Functional Update

definition *update* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ ($[-\text{-}-]$) **where**
$[-f-] = [:x \rightsquigarrow y \;.\; y = f\,x:]$
syntax
-update :: *patterns* $\Rightarrow$ *tuple-args* $\Rightarrow$ *logic* (($1[- \text{ - } \rightsquigarrow \text{ - } -]$))
translations
-update x (*-tuple-args* f F) == *CONST update* ((*-abs* x (*-tuple* f F)))
-update x (*-tuple-arg* F) == *CONST update* (*-abs* x F)

lemma *update-o-def*: $[-f\ o\ g-] = [-x \rightsquigarrow f\ (g\ x)-]$

lemma *update-simp*: $[-f-]\ q = (\lambda\ x\ .\ q\ (f\,x))$

lemma *update-assert-comp*: $[-f-]\ o\ \{.p.\} = \{.p\ o\ f.\}\ o\ [-f-]$

lemma *update-comp*: $[-f-]\ o\ [-g-] = [-g\ o\ f-]$

lemma *update-demonic-comp*: $[-f-]\ o\ [:r:] = [:x \rightsquigarrow y\ .\ r\ (f\,x)\ y:]$

lemma *demonic-update-comp*: $[:r:]\ o\ [-f-] = [:x \rightsquigarrow y\ .\ \exists\ z\ .\ r\,x\,z \wedge y = f\,z:]$

lemma *comp-update-demonic*: $S\ o\ [-f-]\ o\ [:r:] = S\ o\ [:x \rightsquigarrow y\ .\ r\ (f\,x)\ y:]$

lemma *comp-demonic-update*: $S\ o\ [:r:]\ o\ [-f-] = S\ o\ [:x \rightsquigarrow y\ .\ \exists\ z\ .\ r\,x\,z \wedge y = f\,z:]$

lemma *convert*: $(\lambda\ x\ y\ .\ (S{::}('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool))\ x\ (f\,y)) = [-f-]\ o\ S$

lemma *prod-update*: $[-f-] ** [-g-] = [-x,\ y \rightsquigarrow f\,x,\ g\,y\ -]$

lemma *prod-update-skip*: $[-f-] ** Skip = [-\ x,\ y \rightsquigarrow f\,x,\ y-]$

lemma *prod-skip-update*: $Skip ** [-f-] = [-\ x,\ y \rightsquigarrow x,\ f\,y-]$

lemma *prod-assert-update-skip*: $(\{.p.\}\ o\ [-f-]) ** Skip = \{.x,y\ .\ p\ x.\}\ o\ [-\ x,\ y \rightsquigarrow f\,x,\ y-]$

lemma *prod-skip-assert-update*: $Skip ** (\{.p.\}\ o\ [-f-]) = \{.x,y\ .\ p\ y.\}\ o\ [-\lambda\ (x,\ y)\ .\ (x,\ f\,y)-]$

lemma *prod-assert-update*: $(\{.p.\}\ o\ [-f-]) ** (\{.p'.\}\ o\ [-f'-]) = \{.x,y\ .\ p\ x \wedge p'\ y.\}\ o\ [-\lambda\ (x,\ y)\ .\ (f\,x,\ f'\ y)-]$

lemma *update-id-Skip*: $[-id-] = Skip$

lemma *prod-assert-assert-update*: $\{.p.\} ** (\{.p'.\}\ o\ [-f-]) = \{.x,y\ .\ p\ x \wedge p'\ y.\}\ o\ [-\ x,\ y \rightsquigarrow x,\ f\,y-]$

lemma *prod-assert-update-assert*: $(\{.p.\}\ o\ [-f-])** \{.p'.\} = \{.x,y\ .\ p\ x \wedge p'\ y.\}\ o\ [-\ x,\ y \rightsquigarrow f\ x,\ y-]$

lemma *prod-update-assert-update*: $[-f-]\ **\ (\{.p.\}\ o\ [-f'-]) = \{.x,y\ .\ p\ y.\}\ o\ [-x,\ y \rightsquigarrow f\ x,\ f'\ y-]$

lemma *prod-assert-update-update*: $(\{.p.\}\ o\ [-f-])** [-f'-] = \{.x,y\ .\ p\ x\ .\}\ o\ [-\ x,\ y \rightsquigarrow f\ x,\ f'\ y-]$

lemma *Fail-assert-update*: $Fail = \{.\bot.\}\ o\ [-\ (Eps\ \top)\ -]$

lemma *fail-assert-update*: $\bot = \{.\bot.\}\ o\ [-\ (Eps\ \top)\ -]$

lemma *update-fail*: $[-f-]\ o\ \bot = \bot$

lemma *fail-assert-demonic*: $\bot = \{.\bot.\}\ o\ [:\bot:]$

lemma *false-update-fail*: $\{.\lambda x.\ False.\}\ o\ [-f-] = \bot$

lemma *comp-update-update*: $S \circ [-f-] \circ [-f'-] = S \circ [-\ f'\ o\ f\ -]$

lemma *comp-update-assert*: $S \circ [-f-] \circ \{.p.\} = S \circ \{.p\ o\ f.\}\ o\ [-f-]$

lemma *prod-fail*: $\bot\ **\ S = \bot$

lemma *fail-prod*: $S\ **\ \bot = \bot$

lemma *assert-fail*: $\{.p::'a::boolean\text{-}algebra.\}\ o\ \bot = \bot$

lemma *angelic-assert*: $\{:r:\}\ o\ \{.p.\} = \{:x \rightsquigarrow y\ .\ r\ x\ y \wedge p\ y:\}$

lemma *Prod-Skip-angelic-demonic*: $Skip\ **\ (\{:r:\}\ o\ [:r':]) = \{:s,x \rightsquigarrow s',y\ .\ r\ x\ y \wedge s' = s:\}\ o\ [:s,x \rightsquigarrow s',y\ .\ r'\ x\ y \wedge s' = s:]$

lemma *Prod-angelic-demonic-Skip*: $(\{:r:\}\ o\ [:r':])\ **\ Skip = \{:x,\ u \rightsquigarrow y,\ u'\ .\ r\ x\ y \wedge u = u':\}\ o\ \ [:x,\ u \rightsquigarrow y,\ u'\ .\ r'\ x\ y \wedge u = u':]$

lemma *prec-rel-eq*: $p = p' \Longrightarrow r = r' \Longrightarrow \{.p.\}\ o\ [:r:] = \{.p'.\}\ o\ [:r':]$

lemma *prec-rel-le*: $p \leq p' \Longrightarrow (\bigwedge x\ .\ p\ x \Longrightarrow r'\ x \leq r\ x) \Longrightarrow \{.p.\}\ o\ [:r:] \leq \{.p'.\}\ o\ [:r':]$

lemma *assert-update-eq*: $(\{.p.\}\ o\ [-f-] = \{.p'.\}\ o\ [-f'-]) = (p = p' \wedge (\forall\ x.\ p\ x \longrightarrow f\ x = f'\ x))$

lemma *update-eq*: $([-f-] = [-f'-]) = (f = f')$

lemma *spec-eq-iff*:

shows *spec-eq-iff-1*: $p = p' \Longrightarrow f = f' \Longrightarrow \{.p.\}\ o\ [-f-] = \{.p'.\}\ o\ [-f'-]$
and *spec-eq-iff-2*: $f = f' \Longrightarrow [-f-] = [-f'-]$
and *spec-eq-iff-3*: $p = (\lambda\ x\ .\ True) \Longrightarrow f = f' \Longrightarrow \{.p.\}\ o\ [-f-] = [-f'-]$
and *spec-eq-iff-4*: $p = (\lambda\ x\ .\ True) \Longrightarrow f = f' \Longrightarrow [-f-] = \{.p.\}\ o\ [-f'-]$

lemma *spec-eq-iff-a*:
shows$(\bigwedge\ x\ .\ p\ x = p'\ x) \Longrightarrow (\bigwedge\ x\ .\ f\ x = f'\ x) \Longrightarrow \{.p.\}\ o\ [-f-] = \{.p'.\}\ o\ [-f'-]$
and $(\bigwedge\ x\ .\ f\ x = f'\ x) \Longrightarrow [-f-] = [-f'-]$
and $(\bigwedge\ x\ .\ p\ x) \Longrightarrow (\bigwedge\ x\ .\ f\ x = f'\ x) \Longrightarrow \{.p.\}\ o\ [-f-] = [-f'-]$
and $(\bigwedge\ x\ .\ p\ x) \Longrightarrow (\bigwedge\ x\ .\ f\ x = f'\ x) \Longrightarrow [-f-] = \{.p.\}\ o\ [-f'-]$

lemma *spec-eq-iff-prec*: $p = p' \Longrightarrow (\bigwedge\ x\ .\ p\ x \Longrightarrow f\ x = f'\ x) \Longrightarrow \{.p.\}\ o\ [-f-] = \{.p'.\}\ o\ [-f'-]$

lemma *trs-prod*: $trs\ r\ **\ trs\ r' = trs\ (\lambda\ (x,x')\ (y,y')\ .\ r\ x\ y \wedge r'\ x'\ y')$

lemma *sconjunctiveE*: $sconjunctive\ S \Longrightarrow (\exists\ p\ r\ .\ S = \{.\ p\ .\}\ o\ [:\ r :: 'a \Rightarrow 'b \Rightarrow bool:])$

lemma *sconjunctive-prod* [*simp*]: $sconjunctive\ S \Longrightarrow sconjunctive\ S' \Longrightarrow sconjunctive\ (S\ **\ S')$

lemma *nonmagic-prod* [*simp*]: $non\text{-}magic\ S \Longrightarrow non\text{-}magic\ S' \Longrightarrow non\text{-}magic\ (S\ **\ S')$

lemma *non-magic-comp* [*simp*]: $non\text{-}magic\ S \Longrightarrow non\text{-}magic\ S' \Longrightarrow non\text{-}magic\ (S\ o\ S')$

lemma *implementable-pred* [*simp*]: $implementable\ S \Longrightarrow implementable\ S' \Longrightarrow implementable\ (S\ **\ S')$

lemma *implementable-comp*[*simp*]: $implementable\ S \Longrightarrow implementable\ S' \Longrightarrow implementable\ (S\ o\ S')$

lemma *nonmagic-assert*: $non\text{-}magic\ \{.p::'a::boolean\text{-}algebra.\}$

8.5 Control Statements

definition *if-stm* $p\ S\ T = ([.p.]\ o\ S) \sqcap ([.-p.]\ o\ T)$

definition *while-stm* $p\ S = lfp\ (\lambda\ X\ .\ if\text{-}stm\ p\ (S\ o\ X)\ Skip)$

definition *Sup-less* $x\ (w::'b::wellorder) = Sup\ \{(x\ v)::'a::complete\text{-}lattice \mid v\ .\ v < w\}$

lemma *Sup-less-upper*: $v < w \Longrightarrow P\ v \leq Sup\text{-}less\ P\ w$

lemma *Sup-less-least*: $(\bigwedge\ v\ .\ v < w \Longrightarrow P\ v \leq Q) \Longrightarrow Sup\text{-}less\ P\ w \leq Q$

theorem *fp-wf-induction*:
$f\ x\ = x \Longrightarrow mono\ f \Longrightarrow (\forall\ w\ .\ (y\ w) \leq f\ (Sup\text{-}less\ y\ w)) \Longrightarrow Sup\ (range\ y) \leq x$

theorem *lfp-wf-induction*: $mono\ f \Longrightarrow (\forall\ w\ .\ (p\ w) \leq f\ (Sup\text{-}less\ p\ w)) \Longrightarrow Sup\ (range\ p) \leq lfp\ f$

theorem *lfp-wf-induction-a*: $mono\ f \Longrightarrow (\forall\ w\ .\ (p\ w) \leq f\ (Sup\text{-}less\ p\ w)) \Longrightarrow (SUP\ a.\ p\ a) \leq lfp\ f$

theorem *lfp-wf-induction-b*: $mono\ f \Longrightarrow (\forall\ w\ .\ (p\ w) \leq f\ (Sup\text{-}less\ p\ w)) \Longrightarrow S \leq (SUP\ a.\ p\ a) \Longrightarrow S \leq lfp\ f$

lemma [*simp*]: $mono\ S \Longrightarrow mono\ (\lambda X.\ if\text{-}stm\ b\ (S \circ X)\ T)$

definition $mono\text{-}mono\ F = (mono\ F \wedge (\forall\ f\ .\ mono\ f \longrightarrow mono\ (F\ f)))$

theorem *lfp-mono* [*simp*]:
$mono\text{-}mono\ F \Longrightarrow mono\ (lfp\ F)$

lemma *if-mono*[*simp*]: $mono\ S \Longrightarrow mono\ T \Longrightarrow mono\ (if\text{-}stm\ b\ S\ T)$

8.6 Hoare Total Correctness Rules

definition $Hoare\ p\ S\ q = (p \leq S\ q)$

definition $post\text{-}fun\ (p{::}'a{::}order)\ q = (if\ p \leq q\ then\ \top\ else\ \bot)$

lemma *post-mono* [*simp*]: $mono\ (post\text{-}fun\ p :: (\text{-}{::}\{order\text{-}bot,order\text{-}top\}))$

lemma *post-refin* [*simp*]: $mono\ S \Longrightarrow ((S\ p){::}'a{::}bounded\text{-}lattice) \sqcap (post\text{-}fun\ p)\ x \leq S\ x$

lemma *post-top* [*simp*]: $post\text{-}fun\ p\ p = \top$

theorem *hoare-refinement-post*:
$mono\ f \Longrightarrow\ (Hoare\ x\ \ f\ y) = (\{.x{::}'a{::}boolean\text{-}algebra.\}\ o\ (post\text{-}fun\ y) \leq f)$

lemma *assert-Sup-range*: $\{.Sup\ (range\ (p{::}'W \Rightarrow 'a{::}complete\text{-}distrib\text{-}lattice)).\} = Sup(range\ (assert\ o\ p))$

lemma *Sup-range-comp*: $(Sup\ (range\ p))\ o\ S = Sup\ (range\ (\lambda\ w\ .\ ((p\ w)\ o\ S)))$

lemma *Sup-less-comp*: $(Sup\text{-}less\ P)\ w\ o\ S = Sup\text{-}less\ (\lambda\ w\ .\ ((P\ w)\ o\ S))\ w$

lemma *assert-Sup*: $\{.Sup\ (X::'a::complete\text{-}distrib\text{-}lattice\ set).\} = Sup\ (assert\ `X)$

lemma *Sup-less-assert*: $Sup\text{-}less\ (\lambda w.\ \{.\ (p\ w)::'a::complete\text{-}distrib\text{-}lattice\ .\})\ w = \{.Sup\text{-}less\ p\ w.\}$

lemma [*simp*]: $Sup\text{-}less\ (\lambda n\ x.\ t\ x = n)\ n = (\lambda\ x\ .\ (t\ x < n))$

lemma [*simp*]: $Sup\text{-}less\ (\lambda n.\ \{.x.\ t\ x = n.\} \circ S)\ n = \{.x.\ t\ x < n.\} \circ S$

lemma [*simp*]: $(SUP\ a.\ \{.x\ .t\ x = a.\} \circ S) = S$

theorem *hoare-fixpoint*:
$mono\text{-}mono\ F \Longrightarrow$
$(\forall\ f\ w\ .\ mono\ f \longrightarrow (Hoare\ (Sup\text{-}less\ p\ w)\ f\ y \longrightarrow Hoare\ ((p\ w)::'a \Rightarrow bool)\ (F\ f)\ y)) \Longrightarrow Hoare(Sup\ (range\ p))\ (lfp\ F)\ y$

theorem *hoare-sequential*:
$mono\ S \Longrightarrow (Hoare\ p\ (S\ o\ T)\ r) = (\ (\exists\ q.\ Hoare\ p\ S\ q \wedge Hoare\ q\ T\ r))$

theorem *hoare-choice*:
$Hoare\ \ p\ (S \sqcap T)\ q = (Hoare\ p\ S\ q \wedge Hoare\ p\ T\ q)$

theorem *hoare-assume*:
$(Hoare\ P\ [.R.]\ Q) = (P \sqcap R \leq Q)$

lemma *hoare-if*: $mono\ S \Longrightarrow mono\ T \Longrightarrow Hoare\ (p \sqcap b)\ S\ q \Longrightarrow Hoare\ (p \sqcap -b)\ T\ q \Longrightarrow Hoare\ p\ (if\text{-}stm\ b\ S\ T)\ q$

lemma [*simp*]: $mono\ x \Longrightarrow mono\text{-}mono\ (\lambda X\ .\ if\text{-}stm\ b\ (x \circ X)\ Skip)$

lemma *hoare-while*:
$mono\ x \Longrightarrow (\forall\ w\ .\ Hoare\ ((p\ w) \sqcap b)\ x\ (Sup\text{-}less\ p\ w)) \Longrightarrow\ Hoare\ (Sup\ (range\ p))\ (while\text{-}stm\ b\ x)\ ((Sup\ (range\ p)) \sqcap -b)$

lemma *hoare-prec-post*: $mono\ S \Longrightarrow p \leq p' \Longrightarrow q' \leq q \Longrightarrow Hoare\ p'\ S\ q' \Longrightarrow Hoare\ p\ S\ q$

lemma [*simp*]: $mono\ x \Longrightarrow\ mono\ (while\text{-}stm\ b\ x)$

lemma *hoare-while-a*:
$mono\ x \Longrightarrow (\forall\ w\ .\ Hoare\ ((p\ w) \sqcap b)\ x\ (Sup\text{-}less\ p\ w)) \Longrightarrow p' \leq\ (Sup\ (range\ p)) \Longrightarrow ((Sup\ (range\ p)) \sqcap -b) \leq q$

$\Longrightarrow$ *Hoare* p' (*while-stm* b x) q

lemma *hoare-update*: $p \leq q$ *o* $f \Longrightarrow$ *Hoare* p $[-f-]$ q

lemma *hoare-demonic*: $(\bigwedge x\ y\ .\ p\ x \Longrightarrow r\ x\ y \Longrightarrow q\ y) \Longrightarrow$ *Hoare* p $[:r:]$ q

lemma *refinement-hoare*: $S \leq T \Longrightarrow$ *Hoare* $(p::'a::order)$ S $(q) \Longrightarrow$ *Hoare* p T q

lemma *refinement-hoare-iff*: $(S \leq T) = (\forall\ p\ q\ .$ *Hoare* $(p::'a::order)$ S $(q) \longrightarrow$ *Hoare* p T $q)$

8.7 Data Refinement

lemma *data-refinement*: *mono* $S' \Longrightarrow (\forall\ x\ a\ .\ \exists\ u\ .\ R\ x\ a\ u) \Longrightarrow$
$\{:x,\ a \rightsquigarrow x',\ u\ .\ x = x' \wedge R\ x\ a\ u:\}$ *o* $S \leq S'$ *o* $\{:y,\ b \rightsquigarrow y',\ v\ .\ y = y' \wedge R'\ y\ b\ v:\} \Longrightarrow$
$[:x \rightsquigarrow x',\ u\ .\ x = x':]$ *o* S *o* $[:y,\ v \rightsquigarrow y'\ .\ y = y':]$
$\leq [:x \rightsquigarrow x',\ a\ .\ x = x':]$ *o* S' *o* $[:y,\ b \rightsquigarrow y'\ .\ y = y':]$

lemma *mono-update*[*simp*]: *mono* $[-\ f\ -]$

end

9 Feedbackless HBD Translation

theory *FeedbacklessHBDTranslation* **imports** *Diagrams Refinement*
begin
context *BaseOperationFeedbacklessVars*
begin
definition *WhileFeedbackless* =
while-stm $(\lambda\ As\ .$ *internal* $As \neq \{\})$
$[:As \rightsquigarrow As'\ .\ \exists\ A\ .\ A \in$ *set* $As \wedge$ (*out* A) $\in$ *internal* $As \wedge\ As' =$ *map* (*CompA* A) $(As \ominus [A]):]$

definition *TranslateHBDFeedbackless* = *WhileFeedbackless* *o* $[-(\lambda\ As\ .$ *Parallel-list* $As)-]$

definition *ok-fbless* As = (*Deterministic* $As \wedge$ *loop-free* $As \wedge$ *Type-OK* As)

definition *TranslateHBDRec* = $\{.$ *ok-fbless* $.\}$
o $[:As \rightsquigarrow As'\ .\ \exists\ L\ .$ *perm* (*VarFB* (*Parallel-list* As)) $L \wedge As' =$ *fb-less* L $As:]$

lemma [*simp*]:$\{.As.$ *length* (*VarFB* (*Parallel-list* As)) $= w.\}$ (*TranslateHBDRec* x) $y \Longrightarrow [.\ -\ (\lambda As.$ *internal* $As \neq \{\})\ .]\ x\ y$

lemma *internal-fb-less-step*: *loop-free* $As \Longrightarrow$ *Type-OK* $As \Longrightarrow A \in$ *set* $As \Longrightarrow$ *out* $A \in$ *internal* $As \Longrightarrow$ *internal* (*fb-less-step* A $(As \ominus [A])) =$ *internal* As $-$

$\{out\ A\}$

lemma *ok-fbless-fb-less-step*: *ok-fbless As* $\Longrightarrow$ $A \in$ *set As* $\Longrightarrow$ *out A* $\in$ *internal As* $\Longrightarrow$ *ok-fbless* (*fb-less-step A* (*As* $\ominus$ [*A*]))

lemma *map-CompA-fb-out-less-step*: *Deterministic As* $\Longrightarrow$
loop-free As $\Longrightarrow$
Type-OK As $\Longrightarrow$ *A* $\in$ *set As* $\Longrightarrow$ *out A* $\in$ *internal As* $\Longrightarrow$ *map* (*CompA A*) (*As* $\ominus$ [*A*]) = *fb-out-less-step* (*out A*) *As*

lemma *length-diff*: *a* $\in$ *set x* $\Longrightarrow$ *length* (*x* $\ominus$ [*a*]) < *length x*

thm *perm-cons*

lemma *perm-cons-a*: $\bigwedge$ *y* . *a* $\in$ *set x* $\Longrightarrow$ *distinct x* $\Longrightarrow$ *perm* (*x* $\ominus$ [*a*]) *y* $\Longrightarrow$ *perm x* (*a* # *y*)

lemma [*simp*]: {.*As*. *length* (*VarFB* (*Parallel-list As*)) = *w*.} (*TranslateHBDRec x*) *y* $\Longrightarrow$
[. λ*As*. *internal As* $\neq$ {} .]
([:*As*$\rightsquigarrow$*As'*.$\exists$ *A*. *A* $\in$ *set As* $\wedge$ *out A* $\in$ *internal As* $\wedge$ *As'* = *map* (*CompA A*) (*As* $\ominus$ [*A*]):]
({.*As*. *length* (*VarFB* (*Parallel-list As*)) < *w*.} (*TranslateHBDRec x*))) *y*

lemma *Feedbackless-Rec-While-refinement*: *TranslateHBDRec* $\leq$ *WhileFeedbackless*

lemma [*simp*]: *TranslateHBDRec o* [$-$(λ *As* . *Parallel-list As*)$-$] $\leq$ *TranslateHBDFeedbackless*

thm *FB-fb-less*(*1*)

lemma *Out-Parallel-fb-less*: $\bigwedge$ *As* . *Type-OK As* $\Longrightarrow$ *loop-free As* $\Longrightarrow$ *distinct L* $\Longrightarrow$ *set L* $\subseteq$ *internal As* $\Longrightarrow$
Out (*Parallel-list* (*fb-less L As*)) = *concat* (*map Out As*) $\ominus$ *L*

lemma *io-diagram-distinct-VarFB*: *io-diagram A* $\Longrightarrow$ *distinct* (*VarFB A*)

theorem *fbless-correctness*: *ok-fbless As* $\Longrightarrow$ *perm* (*VarFB* (*Parallel-list As*)) *L* $\Longrightarrow$
in-equiv (*FB* (*Parallel-list As*)) (*Parallel-list* (*fb-less L As*))

lemma *Hoare-TranslateHBDRec*: *Hoare* (λ *As* . *As* = *As-init* $\wedge$ *ok-fbless As*)
(*TranslateHBDRec o* [$-$(λ *As* . *Parallel-list As*)$-$])
(λ *A* . *in-equiv* (*FB* (*Parallel-list As-init*)) *A*)

theorem *TranslateHBDFeedbacklessCorrectness*: *Hoare* (λ *As* . *As* = *As-init* $\wedge$ *ok-fbless As*)
TranslateHBDFeedbackless
(λ *A* . *in-equiv* (*FB* (*Parallel-list As-init*)) *A*)

end

end

10 Properties for Proving the Abstract Translation Algorithm

theory *HBDTranslationProperties* **imports** *ExtendedHBDAlgebra Diagrams*
begin
context *BaseOperationVars*
begin

lemma *io-diagram-fb-perm-eq*: *io-diagram A* $\Longrightarrow$ *fb-perm-eq A*

theorem *FeedbackSerial*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*In A*) $\cap$ *set* (*In B*) = {} (*required*)
$\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$ *FB* (*A* ||| *B*) = *FB* (*FB* (*A*) ;; *FB* (*B*))

lemmas *fb-perm-sym* = *fb-perm* [*THEN sym*]

declare *length-TVs* [*simp del*]

declare [[*simp-trace-depth-limit=40*]]

lemma *in-out-equiv-FB*: *io-diagram B* $\Longrightarrow$ *in-out-equiv A B* $\Longrightarrow$ *in-out-equiv* (*FB A*) (*FB B*)

end

end

11 HBD Translation Algorithms that use Feedback Composition

theory *HBDTranslationsUsingFeedback* **imports** *HBDTranslationProperties Refinement*
begin

context *BaseOperationVars*
begin

definition *TranslateHBD* =
while-stm (λ *As* . *length As* > *1*)(
[:*As* ⇝ *As'* . ∃ *Bs Cs* . *1* < *length Bs* ∧ *perm As* (*Bs* @ *Cs*) ∧ *As'* = *FB* (*Parallel-list Bs*) # *Cs*:]
⊓
[:*As* ⇝ *As'* . ∃ *A B Bs* . *perm As* (*A* # *B* # *Bs*) ∧ *As'* = (*FB* (*FB A* ;; *FB B*)) # *Bs*:]
)
o [−(λ *As* . *FB*(*As* ! *0*))−]

lemma [*simp*]:*Suc 0* ≤ *length As-init* ⟹
Hoare (λ*As*. *in-out-equiv* (*FB* (*As* ! *0*)) (*FB* (*Parallel-list As-init*))) [−λ*As*. *FB* (*As* ! *0*)−] (λ*S*. *in-out-equiv S* (*FB* (*Parallel-list As-init*)))

definition *invariant As-init n As* = (*length As* = *n* ∧ *io-distinct As* ∧ *in-out-equiv* (*FB* (*Parallel-list As*)) (*FB* (*Parallel-list As-init*)) ∧ *n* ≥ *1*)

lemma *io-diagram-Parallel-list*: ∀ *A* ∈ *set As* . *io-diagram A* ⟹ *distinct* (*concat* (*map Out As*)) ⟹ *io-diagram* (*Parallel-list As*)

lemma *io-diagram-Parallel-list-a*: *io-distinct As* ⟹ *io-diagram* (*Parallel-list As*)

thm *Parallel-list-cons*

thm *Parallel-assoc-gen*

thm *ParallelId-left*
thm *io-diagram-Parallel-list*

lemma *Parallel-list-append*: ∀ *A* ∈ *set As* . *io-diagram A* ⟹ *distinct* (*concat* (*map Out As*)) ⟹ ∀ *A* ∈ *set Bs* . *io-diagram A*
⟹ *distinct* (*concat* (*map Out Bs*))⟹
Parallel-list (*As* @ *Bs*) = *Parallel-list As* ||| *Parallel-list Bs*

primrec *sequence* :: *nat* ⇒ *nat list* **where**
sequence 0 = [] |
sequence (*Suc n*) = *sequence n* @ [*n*]

lemma *sequence* (*Suc* (*Suc 0*)) = [*0*,*1*]

lemma *in-out-equiv-io-diagram*[*simp*]: *in-out-equiv A B* ⟹ *io-diagram B* ⟹ *io-diagram A*

thm *comp-parallel-distrib*

lemma *in-out-equiv-Parallel-cong-right*: *io-diagram A* $\Longrightarrow$ *io-diagram C* $\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$ *in-out-equiv B C*
$\Longrightarrow$ *in-out-equiv* (*A* ||| *B*) (*A* ||| *C*)

lemma *perm-map*: *perm x y* $\Longrightarrow$ *perm* (*map f x*) (*map f y*)

lemma *distinct-concat-perm*: $\bigwedge$ *Y* . *distinct* (*concat X*) $\Longrightarrow$ *perm X Y* $\Longrightarrow$ *distinct* (*concat Y*)

lemma *distinct-Par-equiv-a*: $\bigwedge$ *Bs* . $\forall$ *A* $\in$ *set As* . *io-diagram A* $\Longrightarrow$ *distinct* (*concat* (*map Out As*)) $\Longrightarrow$ *perm As Bs* $\Longrightarrow$
in-out-equiv (*Parallel-list As*) (*Parallel-list Bs*)

thm *distinct-concat-perm*
thm *perm-map*

lemma *distinct-FB*: *distinct* (*In A*) $\Longrightarrow$ *distinct* (*In* (*FB A*))

lemma *io-distinct-FB-cat*: *io-distinct* (*A* # *Cs*) $\Longrightarrow$ *io-distinct* (*FB A* # *Cs*)

lemma *io-distinct-perm*: *io-distinct As* $\Longrightarrow$ *perm As Bs* $\Longrightarrow$ *io-distinct Bs*

lemma [*simp*]: *distinct* (*concat X*) $\Longrightarrow$ *op-list* [] *op* $\oplus$ (*X*) = *concat X*

lemma [*simp*]: *io-distinct As* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$ *io-distinct* (*FB* (*Parallel-list Bs*) # *Cs*)

lemma *io-distinct-append-a*: *io-distinct As* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$ *io-distinct Bs*

lemma *io-distinct-append-b*: *io-distinct As* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$ *io-distinct Cs*

lemma [*simp*]: *io-distinct As* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$ *io-diagram* (*FB* (*FB* (*Parallel-list Bs*) ||| *Parallel-list Cs*))

lemma [*simp*]: *io-distinct As* $\Longrightarrow$ *io-diagram* (*FB* (*Parallel-list As*))

lemma *io-distinct-set-In*[*simp*]: *io-distinct x* $\Longrightarrow$ *perm x* (*A* # *B* # *Bs*) $\Longrightarrow$ *set* (*In A*) $\cap$ *set* (*In B*) = {}

lemma *io-distinct-set-Out*[*simp*]: *io-distinct x* $\Longrightarrow$ *perm x* (*A* # *B* # *Bs*) $\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {}

lemma *distinct-Par-equiv-b*: *io-distinct As* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$ *in-out-equiv* (*FB* (*FB* (*Parallel-list Bs*) ||| *Parallel-list Cs*)) (*FB* (*Parallel-list As*))

lemma *distinct-Par-equiv*: *io-distinct As-init* $\Longrightarrow$ *Suc 0* $\leq$ *length As-init* $\Longrightarrow$
length As = *w* $\Longrightarrow$ *io-distinct As* $\Longrightarrow$ *in-out-equiv* (*FB* (*Parallel-list As*)) (*FB* (*Parallel-list As-init*)) $\Longrightarrow$
Suc 0 < *w* $\Longrightarrow$ *Suc 0* < *length Bs* $\Longrightarrow$ *perm As* (*Bs* @ *Cs*) $\Longrightarrow$
io-distinct (*FB* (*Parallel-list Bs*) # *Cs*) $\wedge$ *in-out-equiv* (*FB* (*FB* (*Parallel-list Bs*) ||| *Parallel-list Cs*)) (*FB* (*Parallel-list As-init*))

lemma *AAAA-x*[*simp*]: *io-distinct As-init* $\Longrightarrow$ *Suc 0* $\leq$ *length As-init* $\Longrightarrow$ *invariant As-init w x* $\Longrightarrow$ *Suc 0* < *length x* $\Longrightarrow$ *Suc 0* < *length Bs*
$\Longrightarrow$ *perm x* (*Bs* @ *Cs*)
$\Longrightarrow$ *invariant As-init* (*Suc* (*length Cs*)) (*FB* (*Parallel-list Bs*) # *Cs*)

term {*1*,*2*,*3*} − {*2*,*3*}

thm *ParallelId-right*

lemma [*simp*]: *io-distinct As-init* $\Longrightarrow$
Suc 0 $\leq$ *length As-init* $\Longrightarrow$ *invariant As-init w x* $\Longrightarrow$ *Suc 0* < *length x* $\Longrightarrow$ *perm x* (*A* # *B* # *Bs*)
$\Longrightarrow$ *invariant As-init* (*Suc* (*length Bs*)) (*FB* (*FB A* ;; *FB B*) # *Bs*)

lemma [*simp*]: *io-distinct As-init* $\Longrightarrow$ *Suc 0* $\leq$ *length As-init* $\Longrightarrow$
Hoare (*invariant As-init w* $\sqcap$ (λ*As*. *Suc 0* < *length As*))
[:*As*$\rightsquigarrow$*As'*.$\exists$ *Bs*. *Suc 0* < *length Bs* $\wedge$ ($\exists$ *Cs*. *perm As* (*Bs* @ *Cs*) $\wedge$ *As'* = *FB* (*Parallel-list Bs*) # *Cs*):] (*Sup-less* (*invariant As-init*) *w*)

lemma [*simp*]: *io-distinct As-init* $\Longrightarrow$ *Suc 0* $\leq$ *length As-init* $\Longrightarrow$
Hoare (*invariant As-init w* $\sqcap$ (λ*As*. *Suc 0* < *length As*))
[:*As*$\rightsquigarrow$*As'*.$\exists$ *A B Bs*. *perm As* (*A* # *B* # *Bs*) $\wedge$ *As'* = *FB* (*FB A* ;; *FB B*) # *Bs*:] (*Sup-less* (*invariant As-init*) *w*)

theorem *CorrectnessTranslateHBD*: *io-distinct As-init* $\Longrightarrow$ *length As-init* $\geq$ *1* $\Longrightarrow$
Hoare (*io-distinct* $\sqcap$ (λ *As* . *As* = *As-init*)) *TranslateHBD* (λ *S* . *in-out-equiv S* (*FB* (*Parallel-list As-init*)))
end

end