Refinement Calculus of Reactive Systems: Isabelle Theories

Viorel Preoteasa Iulia Dragomir Stavros Tripakis

January 11, 2018

Abstract

This document contains the Isabelle theories of the Refinement Calculus of Reactive Systems (RCRS). It has been automatically generated by Isabelle from the corresponding theories. For an overview of RCRS, the reader is referred primarily to [1, 2]. Additional papers about RCRS are [3, 4, 5, 6, 7, 8]. A precursor of RCRS is the theory of relational interfaces [9].

- Section 1 formalizes the Refinement Calculus [10] and auxiliary concepts needed for RCRS.
- Section 2 formalizes complete distributive lattices.
- Section 3 formalizes linear temporal logic.
- Section 4 formalizes monotonic property transformers, which form the semantic foundation of RCRS.
- Section 5 gives an overview of RCRS following closely the paper [1]. The section numbers in the subsections/subsubsections of Section 5 in the table of contents below refer to the sections of paper [1].
- Section 6 formalizes instantaneous feedback as presented in [4].
- Section 7 formalizes Simulink in RCRS [6, 3].
- Section 8 formalizes list operations and proves properties used in Section 9.
- Section 9 formalizes the hierarchical block diagram translation algorithms presented in [6] and proves that these algorithms yield semantically equivalent results, as presented in [5].

Contents

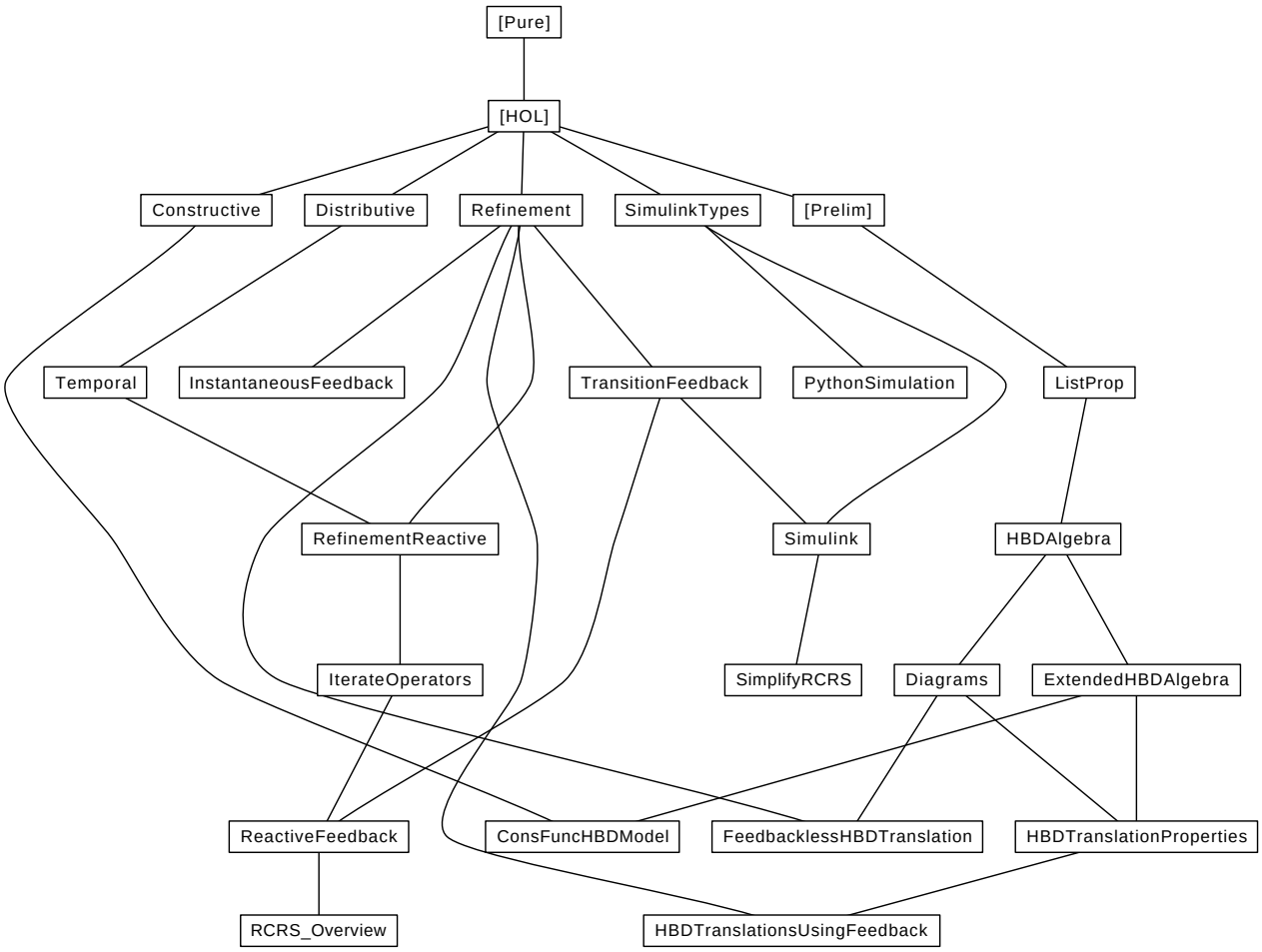

Figure 1: Dependency graph of RCRS Isabelle theories.

1 Refinement Calculus and Monotonic Predicate Transformers

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

In this section we introduce the basics of refinement calculus [10]. Part of this theory is a reformulation of some definitions from [11], but here they are given for predicates, while [11] uses sets.

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

1.1 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*.} = {.(λ *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* ∧ (∀ *y* . *r x y* ⟶ *p' y*).} *o* [:*r OO r'*:]

lemma *demonic-assert-comp*: [:*r*:] *o* {.*p*.} = {.*x*.(∀ *y* . *r x y* ⟶ *p y*).} *o* [:*r*:]

lemma *assert-assert-comp*: {.*p*::'*a*::*lattice*.} *o* {.*p'*.} = {.*p* ⊓ *p'*.}

lemma *assert-assert-comp-pred*: {.*p*.} *o* {.*p'*.} = {.*x* . *p x* ∧ *p' x*.}

lemma *demonic-refinement*: *r'* ≤ *r* ⟹ [:*r*:] ≤ [:*r'*:]

definition *inpt r x* = (∃ *y* . *r x y*)

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

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

lemma *assert-demonic-prop*: {.*p*.} *o* [:*r*:] = {.*p*.} *o* [:(λ *x y* . *p x*) ⊓ *r*:]

lemma *trs-trs*: (*trs r*) *o* (*trs r'*)
= *trs* ((λ *s t*. (∀ *s'* . *r s s'* ⟶ (*inpt r' s'*))) ⊓ (*r OO r'*)) (**is** *?S* = *?T*)

lemma *prec-inpt-equiv*: *p* ≤ *inpt r* ⟹ *r'* = (λ *x y* . *p x* ∧ *r x y*) ⟹ {.*p*.} *o* [:*r*:] = {:*r'*:]

lemma *assert-demonic-refinement*: ({.*p*.} *o* [:*r*:] ≤ {.*p'*.} *o* [:*r'*:]) = (*p* ≤ *p'* ∧ (∀ *x* . *p x* ⟶ *r' x* ≤ *r x*))

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

1.2 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* $S = -\ (\mathit{fail}\ S)$
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* $(\{.p.\}\ o\ [:r::'a \Rightarrow 'b \Rightarrow \mathit{bool}:]) = p$

lemma *fail-spec*: *fail* $(\{.p.\}\ o\ [:(r::'a \Rightarrow 'b::\mathit{boolean\text{-}algebra):]) = -p$

lemma [*simp*]: *prec* $(\{.p.\}\ o\ [:(r::'a \Rightarrow 'b::\mathit{boolean\text{-}algebra):]) = p$

lemma [*simp*]: *prec* $(T::('a::\mathit{boolean\text{-}algebra} \Rightarrow 'b::\mathit{boolean\text{-}algebra})) = \top \Longrightarrow \mathit{prec}\ (S\ o\ T) = \mathit{prec}\ S$

lemma [*simp*]: *prec* $[:r::'a \Rightarrow 'b::\mathit{boolean\text{-}algebra}:] = \top$

lemma *prec-rel*: $\{.\ p\ .\} \circ [:\ \lambda x\ y.\ p\ x \wedge r\ x\ y\ :] = \{.p.\}\ o\ [:r:]$

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* $(S::'a::\mathit{boolean\text{-}algebra} \Rightarrow 'b::\mathit{boolean\text{-}algebra}) \Longrightarrow (S = \mathit{Fail}) = (\mathit{fail}\ S = \top)$

lemma *sconjunctive-spec*: *sconjunctive* $S \Longrightarrow S = \{.\mathit{prec}\ S.\}\ o\ [:\mathit{rel}\ S:]$

definition *non-magic* $S = (S\ \bot = \bot)$

lemma *non-magic-spec*: *non-magic* $(\{.p.\}\ o\ [:r:]) = (p \leq \mathit{inpt}\ r)$

lemma *sconjunctive-non-magic*: *sconjunctive* $S \Longrightarrow$ *non-magic* $S = (\mathit{prec}\ S \leq \mathit{inpt}\ (\mathit{rel}\ S))$

definition *implementable* $S = (\mathit{sconjunctive}\ S \wedge \mathit{non\text{-}magic}\ S)$

lemma *implementable-spec*: *implementable* $S \Longrightarrow \exists\ p\ r\ .\ S = \{.p.\}\ o\ [:r:] \wedge p \leq \mathit{inpt}\ r$

definition *Skip* $= (\mathit{id}::('a \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow \mathit{bool}))$

lemma *assert-true-skip*: $\{.\top::'a \Rightarrow \mathit{bool}.\} = \mathit{Skip}$

lemma *skip-comp* [*simp*]: *Skip o S* $= S$

lemma *comp-skip*[*simp*]:*S o Skip* $= S$

lemma *assert-rel-skip*[*simp*]: $\{.\ \lambda\ (x,\ y)\ .\ \mathit{True}\ .\} = \mathit{Skip}$

lemma [*simp*]: $mono\ S \Longrightarrow mono\ S' \Longrightarrow mono\ (S\ o\ S')$

lemma [*simp*]: $mono\ \{.p::('a \Rightarrow bool).\}$

lemma [*simp*]: $mono\ [:r::('a \Rightarrow 'b \Rightarrow bool):]$

lemma *assert-true-skip-a*: $\{.\ x\ .\ True\ .\} = Skip$

lemma *assert-false-fail*: $\{.\bot::'a::boolean\text{-}algebra.\} = \bot$

lemma *magoc-comp*[*simp*]: $\top\ o\ S = \top$

lemma *left-comp*: $T\ o\ U = T'\ o\ U' \Longrightarrow S\ o\ T\ o\ U = S\ o\ T'\ o\ U'$

lemma *assert-demonic*: $\{.p.\}\ o\ [:r:] = \{.p.\}\ o\ [:x \rightsquigarrow y\ .\ p\ x \wedge r\ x\ y:]$

lemma $trs\ r \sqcap trs\ r' = trs\ (\lambda\ x\ y\ .\ inpt\ r\ x \wedge inpt\ r'\ x \wedge (r\ x\ y \vee r'\ x\ y))$

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

lemma *mono-assume*[*simp*]: $mono\ [.p.]$

lemma *mono-demonic*[*simp*]: $mono\ [:r:]$

lemma *mono-comp-a*[*simp*]: $mono\ S \Longrightarrow mono\ T \Longrightarrow mono\ (S\ o\ T)$

lemma *mono-demonic-choice*[*simp*]: $mono\ S \Longrightarrow mono\ T \Longrightarrow mono\ (S \sqcap T)$

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:]

1.3 Product and Fusion of predicate transformers

In this section we define the fusion and product operators from [12]. 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**
$Fusion\ S\ q\ x = (\exists\ (p::'c \Rightarrow 'a \Rightarrow bool)\ .\ (INF\ c\ .\ p\ c) \leq q \wedge (\forall\ c\ .\ (S\ c)\ (p\ c)\ x))$

lemma *Fusion-spec*: $Fusion\ (\lambda\ n\ .\ \{.p\ n.\} \circ [:r\ n:]) = (\{.INFIMUM\ UNIV\ p.\} \circ [:INFIMUM\ UNIV\ r:])$

lemma *Fusion-demonic*: $Fusion\ (\lambda\ n\ .\ [:r\ n:]) = [:INF\ n\ .\ r\ n:]$

lemma *Fusion-refinement*: $(\bigwedge i\ .\ S\ i \leq T\ i) \Longrightarrow Fusion\ S \leq Fusion\ T$

lemma *mono-fusion*[*simp*]: $mono\ (S \parallel T)$

lemma *mono-Fusion*: $mono\ (Fusion\ S)$

definition *prod-pred* $A\ B = (\lambda(a,\ b).\ A\ a \wedge B\ b)$
definition *Prod* :: $((('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)) \Rightarrow ((('c \Rightarrow bool) \Rightarrow ('d \Rightarrow bool)) \Rightarrow (('a \times 'c \Rightarrow bool)$
$\Rightarrow ('b \times 'd \Rightarrow bool))$
(**infixr** $**$ *70*)
where
$(S ** T)\ q = (\lambda\ (x,\ y)\ .\ \exists\ p\ p'\ .\ prod\text{-}pred\ p\ p' \leq q \wedge S\ p\ x \wedge T\ p'\ y)$

lemma *mono-prod*[*simp*]: $mono\ (S ** T)$

lemma *Prod-spec*: $(\{.p.\}\ o\ [:r:]) ** (\{.p'.\}\ o\ [:r':]) = \{.x,y\ .\ p\ x \wedge p'\ y.\}\ o\ [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge r'$
$y\ v:]$

lemma *Prod-demonic*: $[:r:] ** [:r':] = [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge r'\ y\ v:]$

lemma *Prod-spec-Skip*: $(\{.p.\}\ o\ [:r:]) ** Skip = \{.x,y\ .\ p\ x.\}\ o\ [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge v = y:]$

lemma *Prod-Skip-spec*: $Skip ** (\{.p.\}\ o\ [:r:]) = \{.x,y\ .\ p\ y.\}\ o\ [:x,\ y \rightsquigarrow u,\ v\ .\ x = u \wedge r\ y\ v:]$

lemma *Prod-skip-demonic*: $Skip ** [:r:] = [:x,\ y \rightsquigarrow u,\ v\ .\ x = u \wedge r\ y\ v:]$

lemma *Prod-demonic-skip*: $[:r:] ** Skip = [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge\ \ y = v:]$

lemma *Prod-spec-demonic*: $(\{.p.\}\ o\ [:r:]) **\ \ [:r':] = \{.x,\ y\ .\ p\ x.\}\ o\ [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge r'\ y\ v:]$

lemma *Prod-demonic-spec*: $[:r:] ** (\{.p.\}\ o\ [:r':]) = \{.x,\ y\ .\ p\ y.\}\ o\ [:x,\ y \rightsquigarrow u,\ v\ .\ r\ x\ u \wedge r'\ y\ v:]$

lemma *pair-eq-demonic-skip*: $[:\ \lambda(x,\ y)\ (u,\ v).\ x = u \wedge v = y\ :] = Skip$

lemma *Prod-assert-skip*: $\{.p.\} ** Skip = \{.x,y\ .\ p\ x.\}$

lemma *Prod-skip-assert*: $Skip \ast\ast \{.p.\} = \{.x,y\ .\ p\ y.\}$

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*: $S \leq S' \Longrightarrow S \ast\ast T \leq S' \ast\ast T$

lemma *prod-mono2*: $S \leq S' \Longrightarrow T \ast\ast S \leq T \ast\ast S'$

lemma *Prod-fusion*: $S \ast\ast T = ([:x,y \rightsquigarrow x'\ .\ x = x':]\ o\ S\ o\ [:x \rightsquigarrow x',\ y\ .\ x = x':]) \parallel ([:x,\ y \rightsquigarrow y'\ .\ y = y':]\ o\ T\ o\ [:y \rightsquigarrow x,\ y'\ .\ y = y':])$

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*]: $Skip \ast\ast Magic = Magic$

lemma [*simp*]: $S \ast\ast Fail = Fail$

lemma [*simp*]: $Fail \ast\ast S = Fail$

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*]: $Skip \ast\ast Skip = Skip$

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*]: $prec\ S = \top \Longrightarrow prec\ T = \top \Longrightarrow prec\ (S \ast\ast T) = \top$

lemma *prec-skip*[*simp*]: $prec\ Skip = (\top::'a \Rightarrow bool)$

lemma [*simp*]: $prec\ S = \top \Longrightarrow prec\ T = \top \Longrightarrow prec\ (S \parallel T) = \top$

1.4 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* ⇒ *tuple-args* ⇒ *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*: $S \ast\ast \bot = \bot$

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*: $Skip \ast\ast (\{:r:\}\ o\ [:r':]) = \{:s,x \rightsquigarrow s',y\ .\ r\ x\ y \wedge s' = s:\}\ o\ [:s,x \rightsquigarrow s',y\ .\ r'\ x\ y \wedge s' = s:]$

lemma *Prod-angelic-demonic-Skip*: $(\{:r:\}\ o\ [:r':]) \ast\ast Skip = \{:x,\ u \rightsquigarrow y,\ u'\ .\ r\ x\ y \wedge u = u':\}\ o\ [:x,\ u \rightsquigarrow y,\ u'\ .\ r'\ x\ y \wedge u = u':]$

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*: $trs\ r \ast\ast trs\ r' = trs\ (\lambda\ (x,x')\ (y,y')\ .\ r\ x\ y \wedge r'\ x'\ y')$

lemma *sconjunctiveE*: $sconjunctive\ S \Longrightarrow (\exists\ p\ r\ .\ S = \{.\ p\ .\}\ o\ [:\ r\ ::'a \Rightarrow 'b \Rightarrow bool:])$

lemma *sconjunctive-prod* [*simp*]: $sconjunctive\ S \Longrightarrow sconjunctive\ S' \Longrightarrow sconjunctive\ (S \ast\ast S')$

lemma *nonmagic-prod* [*simp*]: $non\text{-}magic\ S \Longrightarrow non\text{-}magic\ S' \Longrightarrow non\text{-}magic\ (S \ast\ast S')$

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*]: $implementable\ S \Longrightarrow implementable\ S' \Longrightarrow implementable\ (S \ast\ast S')$

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

lemma *nonmagic-assert*: $non\text{-}magic\ \{.p::'a::boolean\text{-}algebra.\}$

1.5 Control Statements

definition *if-stm p S T* = ([.*p*.] *o S*) ⊓ ([.−*p*.] *o T*)

definition *while-stm p S* = *lfp* (λ *X* . *if-stm p* (*S o X*) *Skip*)

definition *Sup-less x* (*w*::′*b*::*wellorder*) = *Sup* {(*x v*)::′*a*::*complete-lattice* | *v* . *v* < *w*}

lemma *Sup-less-upper*: *v* < *w* ⟹ *P v* ≤ *Sup-less P w*

lemma *Sup-less-least*: (⋀ *v* . *v* < *w* ⟹ *P v* ≤ *Q*) ⟹ *Sup-less P w* ≤ *Q*

theorem *fp-wf-induction*:
f x = *x* ⟹ *mono f* ⟹ (∀ *w* . (*y w*) ≤ *f* (*Sup-less y w*)) ⟹ *Sup* (*range y*) ≤ *x*

theorem *lfp-wf-induction*: *mono f* ⟹ (∀ *w* . (*p w*) ≤ *f* (*Sup-less p w*)) ⟹ *Sup* (*range p*) ≤ *lfp f*

theorem *lfp-wf-induction-a*: *mono f* ⟹ (∀ *w* . (*p w*) ≤ *f* (*Sup-less p w*)) ⟹ (*SUP a*. *p a*) ≤ *lfp f*

theorem *lfp-wf-induction-b*: *mono f* ⟹ (∀ *w* . (*p w*) ≤ *f* (*Sup-less p w*)) ⟹ *S* ≤ (*SUP a*. *p a*) ⟹ *S* ≤ *lfp f*

lemma [*simp*]: *mono S* ⟹ *mono* (λ*X*. *if-stm b* (*S* ∘ *X*) *T*)

definition *mono-mono F* = (*mono F* ∧ (∀ *f* . *mono f* ⟶ *mono* (*F f*)))

theorem *lfp-mono* [*simp*]:
mono-mono F ⟹ *mono* (*lfp F*)

lemma *if-mono*[*simp*]: *mono S* ⟹ *mono T* ⟹ *mono* (*if-stm b S T*)

1.6 Hoare Total Correctness Rules

definition *Hoare p S q* = (*p* ≤ *S q*)

definition *post-fun* (*p*::′*a*::*order*) *q* = (*if p* ≤ *q then* ⊤ *else* ⊥)

lemma *post-mono* [*simp*]: *mono* (*post-fun p* :: (-::{*order-bot*,*order-top*}))

lemma *post-refin* [*simp*]: *mono S* ⟹ ((*S p*)::′*a*::*bounded-lattice*) ⊓ (*post-fun p*) *x* ≤ *S x*

lemma *post-top* [*simp*]: *post-fun p p* = ⊤

theorem *hoare-refinement-post*:
mono f ⟹ (*Hoare x f y*) = ({.*x*::′*a*::*boolean-algebra*.} *o* (*post-fun y*) ≤ *f*)

lemma *assert-Sup-range*: {.*Sup* (*range* (*p*::′*W* ⇒ ′*a*::*complete-distrib-lattice*)).} = *Sup*(*range* (*assert o p*))

lemma *Sup-range-comp*: (*Sup* (*range p*)) *o S* = *Sup* (*range* (λ *w* . ((*p w*) *o S*)))

lemma *Sup-less-comp*: (*Sup-less P*) *w o S* = *Sup-less* (λ *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*:
$mono\ x \Longrightarrow (\forall\ w\ .\ Hoare\ ((p\ w) \sqcap b)\ x\ (Sup\text{-}less\ p\ w)) \Longrightarrow p' \leq\ (Sup\ (range\ p)) \Longrightarrow ((Sup\ (range\ p)) \sqcap -b) \leq q$
$\Longrightarrow\ \ Hoare\ p'\ (while\text{-}stm\ b\ x)\ q$

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*: $(S \leq T) = (\forall\ p\ q\ .\ Hoare\ (p::'a::order)\ S\ (q) \longrightarrow Hoare\ p\ T\ q)$

1.7 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

1.8 Feedback Operator on Predicate Transformers

theory *TransitionFeedback*
imports *../RefinementReactive/Refinement Complex*

begin

definition *grd-update* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ ($[-(-)\rightarrow(-)-]$) **where**
$[-p\rightarrow f-] = [:x \rightsquigarrow y\ .\ p\ x \wedge y = f\ x:]$

lemma $[-p\rightarrow f-] = [.p.]\ o\ [-f-]$

lemma *assert-grd-update*: $(\bigwedge\ x\ .\ p\ x \Longrightarrow p'\ x) \Longrightarrow \{.p.\}\ o\ [-p'\rightarrow f-] = \{.p.\}\ o\ [-f-]$

lemma *grd-update-comp*: $[-p\rightarrow f-]\ o\ [-q\rightarrow g-] = [-p \sqcap (q\ o\ f) \rightarrow g\ o\ f-]$

lemma *grd-update-assert-comp*: $[-p\rightarrow f-]\ o\ \{.q.\} = \{.\ x\ .\ p\ x \longrightarrow q\ (f\ x).\}\ o\ [-p \rightarrow f-]$

lemma *grd-update-update-comp*: $[-p\rightarrow f-]\ o\ [-g-] = [-p \rightarrow g\ o\ f-]$

lemma *update-grd-update-comp*: $[-g-]\ o\ [-p\rightarrow f-] = [-p\ o\ g \rightarrow f\ o\ g-]$

lemma *grd-update-update* [*simp*]: $[-\top\rightarrow f-] = [-f-]$

lemma [*simp*]: $(\exists\, y.\ (a,\ y) = f\ (u,\ x)) = (a = fst\ (f\ (u,\ x)))$

lemma *pair-eq*: $((a,\ b) = x) = (a = fst\ x \wedge b = snd\ x)$

lemma *comp-exists*: $(r\ OO\ r')\ x\ y = (\exists\ z\ .\ r\ x\ z \wedge r'\ z\ y)$

lemma *comp-existsa*: $(r\ OO\ r') = (\lambda\ x\ y\ .\ \exists\ z\ .\ r\ x\ z \wedge r'\ z\ y)$

lemma *drop-assumption*: $p \Longrightarrow True$

lemma *fun-comp-simp*: $((\lambda(x,\ y).\ (f\ x,\ y)) \circ (\lambda(a,\ b).\ (c\ b,\ d\ (a,\ b)))) = (\lambda\ (a,\ b)\ .\ (((f\ o\ c)\ b),\ d\ (a,\ b)))$

lemma *fun-comp-simp-b*: $((\lambda(a::'c,\ b::'d).\ (c\ b,\ d\ (a,\ b))) \circ (\lambda(x::'a,\ y::'d).\ (f\ x,\ y))) = (\lambda\ (x,\ y)\ .\ (c\ y,\ d\ (f\ x,\ y)))$

lemma *fun-comp-simp-c*: $((\lambda((c,\ d),\ a).\ (a,\ c,\ d)) \circ (\lambda(x,\ y).\ (case\ x\ of\ (a,\ b) \Rightarrow (c\ b,\ d\ (a,\ b)),\ f\ y)) \circ (\lambda(a,\ c,\ b).\ ((a,\ b),\ c))) = (\lambda\ (u,v,w)\ .\ (f\ v,\ c\ w,\ d\ (u,\ w)))$

lemma *fun-comp-simp-d*: $(\lambda x.\ case\ case\ x\ of\ (c,\ b) \Rightarrow ((case\ x\ of\ (v,\ w) \Rightarrow f\ v,\ b),\ c)\ of\ (x,\ y) \Rightarrow p\ x \wedge p'\ y) = (\lambda\ (u,v)\ .\ p\ (f\ u,\ v) \wedge p'\ u)$

lemma *fun-comp-simp-e*: $(\lambda x.\ case\ x\ of\ (v,\ w) \Rightarrow (c\ w,\ d\ (case\ x\ of\ (v,\ w) \Rightarrow f\ v,\ w))) = (\lambda\ (u,\ v)\ .\ (c\ v,\ d\ (f\ u,\ v)))$

definition *select* $S = \{.\ x\ .\ (\exists\ u\ .\ prec\ S\ (u,\ x)).\}\ o\ [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge prec\ S\ (u,\ x)\ :]\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

lemma *selectc-spec*: *select* $(\{.\ p\ .\}\ o\ [:r:]) = \{.\ x\ .\ (\exists\ u\ .\ p\ (u,\ x)).\}\ o\ [:x \rightsquigarrow v\ .\ \exists\ u\ y\ .\ p\ (u,\ x) \wedge r\ (u,x)\ (v,y)\ :]$

lemma *select-sconjunctive*[*simp*]: *sconjunctive* $S \Longrightarrow$ *sconjunctive* $(select\ S)$

lemma *sconjunctive-fusion*[*simp*]: *sconjunctive* $S \Longrightarrow$ *sconjunctive* $S' \Longrightarrow$ *sconjunctive* $(S \parallel S')$

lemma *sconjunctive-Skip*[*simp*]: *sconjunctive Skip*

lemma [*simp*]: *prec* $S = \top \Longrightarrow$ *prec* $(select\ S) = \top$

definition *selectA* $S = \{.\ x\ .\ (\exists\ u\ .\ prec\ S\ (u,\ x)).\}\ o\ [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge prec\ S\ (u,\ x)\ :]\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *selectB* $S = \{:x \rightsquigarrow u,\ x'\ .\ x = x':\}\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *selectC* $S = \{:x \rightsquigarrow u,\ x'\ .\ x = x':\}\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *feedback* $S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((select\ S) **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

definition *feedbackA* $S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((selectA\ S) **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

definition *feedbackB* $S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((selectB\ S) **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

definition *feedbackC* $S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((selectC\ S) **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

lemma *selectA-spec*: *selectA* $(\{.\ p\ .\}\ o\ [:r:]) = \{.\ x\ .\ (\exists\ u\ .\ p\ (u,\ x)).\}\ o\ [:x \rightsquigarrow u\ .\ \exists\ \ y\ .\ p\ (u,\ x) \wedge r\ (u,x)\ (u,y)\ :]$

thm *Prod-angelic-demonic-Skip*

lemma *feedbackB-spec*: *feedbackB* $(\{.p.\}\ o\ [:r:]) = \{:x \rightsquigarrow u,\ x'\ .\ p\ (u,x) \wedge (\forall\ v\ y\ .\ r\ (u,x)\ (v,\ y) \longrightarrow p\ (v,x)) \wedge x = x':\}\ o\ [:u,x \rightsquigarrow y\ .\ \exists\ v\ y'\ .\ r\ (u,x)\ (v,\ y') \wedge r\ (v,\ x)\ (v,\ y):]$

lemma *feedbackC-spec*: *feedbackC* $(\{.p.\}\ o\ [:r:]) = \{:x \rightsquigarrow u,\ x'\ .\ p\ (u,x) \wedge (\forall\ y\ .\ r\ (u,x)\ (u,\ y) \longrightarrow p\ (u,x)) \wedge x = x':\}\ o\ [:u,x \rightsquigarrow y\ .\ r\ (u,\ x)\ (u,\ y):]$

lemma *feedbackB-decomp*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
$feedbackB\ (\{.\ u,\ x\ .\ p\ (u,\ x) \wedge p'\ x.\}\ o\ [:u,\ x \rightsquigarrow v,\ y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ v:])$
$= \{.\ x\ .\ p'\ x \wedge (\forall\ b.\ r'\ x\ b \longrightarrow p\ (b,x)).\}\ o\ [:x \rightsquigarrow y\ .\ \exists\ v\ .\ r'\ x\ v \wedge r\ (v,\ x)\ y:]$

lemma [*simp*]: *prec* $S = \top \Longrightarrow$ *prec* (*feedback* S) $= \top$

lemma *feedback-simp-a*: *feedback* ({.*p*.} *o* [:*r*:]) =
{. λx. ($\exists\, u$. *p* (u, x)) $\wedge$ ($\forall\, a$. ($\exists\, u$. *p* (u, x) $\wedge$ ($\exists\, y$. *r* (u, x) (a, y))) $\longrightarrow$ *p* (a, x)) .} $\circ$
[:$x \rightsquigarrow y$. ($\exists$ v .($\exists\, u$. *p* (u, x) $\wedge$ ($\exists\, y$. *r* (u, x) (v, y))) $\wedge$ *r* (v, x) (v, y)):]

lemma *feedbackA-simp-a*: *feedbackA* ({.*p*.} *o* [:*r*:]) =
{. x. $\exists\, u$. *p* (u, x) .} $\circ$ [:$x \rightsquigarrow z$. $\exists\, a$. *p* (a, x) $\wedge$ *r* (a, x) (a, z):]

lemma *feedback-simp-b*: *feedback* ({.*p*.} *o* [$-q \rightarrow f-$]) =
{. λx. ($\exists\, u$. *p* (u, x)) $\wedge$ ($\forall\, u$. *p* (u, x) $\wedge$ *q* (u, x) $\longrightarrow$ *p* (*fst* (*f* (u, x)), x)) .} $\circ$
[:$x \rightsquigarrow y$. ($\exists$ u . *p* (u, x) $\wedge$ *q* (u, x) $\wedge$ *q* (*fst* (*f* (u, x)), x) $\wedge$ *fst* (*f* (u, x)) = *fst* (*f* (*fst* (*f* (u, x)), x)) $\wedge$ y = *snd* (*f* (*fst* (*f* (u, x)), x))):]

lemma *feedback-simp-c*: *feedback* ({.*p*.} *o* [$-f-$]) =
{. x. ($\exists\, u$. *p* (u, x)) $\wedge$ ($\forall\, u$. *p* (u, x) $\longrightarrow$ *p* (*fst* (*f* (u, x)), x)) .} $\circ$
[:$x \rightsquigarrow y$. ($\exists$ u . *p* (u, x) $\wedge$ *fst* (*f* (u, x)) = *fst* (*f* (*fst* (*f* (u, x)), x)) $\wedge$ y = *snd* (*f* (*fst* (*f* (u, x)), x))):]

lemma *feedback-simp-cc*: *feedback* ([$-f-$]) =
[:$x \rightsquigarrow y$. ($\exists$ u . *fst* (*f* (u, x)) = *fst* (*f* (*fst* (*f* (u, x)), x)) $\wedge$ y = *snd* (*f* (*fst* (*f* (u, x)), x))):]

lemma *feedback-test*: *feedback* ([$-(\lambda$ (u,x) . (u , u))$-$]) = [:$\top$:]

lemma *feedback-simp-d*: *feedback* [:*r*:] = [: $x \rightsquigarrow y$. $\exists$ v . *r* (v, x) (v, y):]

lemma *feedback-update-simp*: *feedback* ({.*p*.} *o* [$-$ λ (u, x) . (*f* x, *g* (u, x))$-$])
= {. x . *p* (*f* x, x).} *o* [$-\lambda$ x . *g* (*f* x, x)$-$]

lemma *feedback-update-simp-x*: *feedback* ({. *p*.} *o* [$-$ λ *ux* . (*f* (*snd* *ux*), *g* *ux*)$-$])
= {. x . *p* (*f* x, x).} *o* [$-\lambda$ x . *g* (*f* x, x)$-$]

lemma *feedback-update-simp-a*: *feedback* ({.*p*.} *o* [$-$ λ (u, s, x) . (*f* (s, x), *g* (u, s, x), *h* (u, s, x)) $-$])
= {. s, x . *p* ((*f* (s, x)), s, x).} *o* [$-\lambda$ (s, x) . (*g* ((*f* (s, x)), s, x), *h* ((*f* (s, x)), s, x))$-$]

lemma *feedback-update-simp-b*: *feedback* ({.*p*.} *o* [$-$ λ (u, s, x) . (*f* (s, x), *g* (u, s, x), *h* (u, s, x)) $-$])
= {. s, x . *p* ((*f* (s, x)), s, x).} *o* [$-\lambda$ (s, x) . (*g* ((*f* (s, x)), s, x), *h* ((*f* (s, x)), s, x))$-$]

lemma *feedback-update-simp-c*: *feedback* ({. (u, s, x) . *p* u s x .} *o* [$-$ λ (u, s, x) . (*f* s x, *g* u s x, *h* u s x) $-$])
= {. s, x . *p* (*f* s x) s x.} *o* [$-\lambda$ (s, x) . (*g* (*f* s x) s x, *h* (*f* s x) s x)$-$]

lemma *feedback-simp-bot*: *feedback* ($\bot$::(('a $\times$ 'b) $\Rightarrow$ *bool*) $\Rightarrow$ ('a $\times$ 'c) $\Rightarrow$ *bool*) = $\bot$

lemma A = {.*p*.} *o* [$-\lambda$ (a, b) . (c b, d (a, b))$-$] $\Longrightarrow$ B = {.*p*'.} *o* [$-f-$] $\Longrightarrow$ *feedback* (A *o* (B ** *Skip*))
= {. x . *p* (*f* (c x), x) $\wedge$ *p*' (c x) .} $\circ$ [$-\lambda x$. d (*f* (c x), x)$-$]

lemma *AAA*: $p = p' \Longrightarrow$ ($\bigwedge$ x . *p* x $\Longrightarrow$ *r* x = *r*' x) $\Longrightarrow$ {.*p*.} *o* [:*r*:] = {.*p*'.} *o* [:*r*':]

thm *feedback-simp-a*

lemma $A = \{.p.\}\ o\ [-\lambda\ (a,\ b)\ .\ (c\ b,\ d\ (a,\ b))-] \Longrightarrow B = \{.p'.\}\ o\ [-f-] \Longrightarrow$ *feedback* $((B\ **\ Skip)\ o\ A)$
$= \{.\ x\ .\ p\ (f\ (c\ x),\ x) \wedge p'\ (c\ x)\ .\} \circ [-\lambda x\ .\ d\ (f\ (c\ x),\ x)-]$

lemma $A = \{.p.\}\ o\ [-\lambda\ (a,\ b)\ .\ (c\ b,\ d\ (a,\ b))-] \Longrightarrow B = \{.p'.\}\ o\ [-f-] \Longrightarrow$
feedback (*feedback* $([-\lambda\ (a,c,b)\ .\ ((a,b),\ c)-]\ o\ (A\ **\ B)\ o\ [-\lambda\ ((c,\ d),\ a)\ .\ (a,c,d)-]\ )) = \{.\ x\ .\ p\ (f\ (c\ x),\ x) \wedge p'\ (c\ x)\ .\} \circ [-\lambda x\ .\ d\ (f\ (c\ x),\ x)-]$
lemma *feedback-simp-aa*: *feedback* $(\{.inpt\ r.\}\ o\ [:r:]) =$
$\{.\ \lambda x.\ (\exists\, u.\ inpt\ r\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u.\ inpt\ r\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow inpt\ r\ (a,\ x)).\} \circ$
$[:x \rightsquigarrow y\ .\ (\exists\ v\ .(\exists\, u.\ \ (\exists\, y.\ r\ (u,\ x)\ (v,\ y))) \wedge\ \ r\ (v,\ x)\ (v,\ y)):]$

lemma *feedback-in-simp-aux*: $((\exists\, u.\ inpt\ r\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u.\ inpt\ r\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow inpt\ r\ (a,\ x)))$
$= ((\exists\, u.\ inpt\ r\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u\ y.\ r\ (u,\ x)\ (a,\ y)) \longrightarrow inpt\ r\ (a,\ x)))$

lemma *feedback-simp-aaa*: *feedback* $(\{.inpt\ r.\}\ o\ [:r:]) =$
$\{.\ \lambda x.\ (\exists\, u.\ inpt\ r\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u.\ inpt\ r\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow inpt\ r\ (a,\ x)).\} \circ$
$[:x \rightsquigarrow y\ .\ (\exists\ v\ .\ \ r\ (v,\ x)\ (v,\ y)):]$

lemma *feedbackB-simp-aaaa*: *feedbackB* $(\{.inpt\ r.\}\ o\ [:r:]) =$
$\{:x \rightsquigarrow (u,\ x').inpt\ r\ (u,\ x) \wedge (\forall\, v.\ (\exists\, y.\ r\ (u,\ x)\ (v,\ y)) \longrightarrow inpt\ r\ (v,\ x)) \wedge x = x':\} \circ [:(u,\ x) \rightsquigarrow y.\exists\, v.\ (\exists\, y'.\ r\ (u,\ x)\ (v,\ y')) \wedge r\ (v,\ x)\ (v,\ y):]$

lemma *feedbackB-simp-aaaaa*: $p \leq inpt\ r \Longrightarrow$ *feedbackB* $(\{.p.\}\ o\ [:r:]) =$
$\{:x \rightsquigarrow (u,\ x').\ p\ (u,\ x) \wedge (\forall\, v.\ (\exists\, y.\ r\ (u,\ x)\ (v,\ y)) \longrightarrow p\ (v,\ x)) \wedge x = x':\} \circ [:(u,\ x) \rightsquigarrow y.\exists\, v.\ (\exists\, y'.\ r\ (u,\ x)\ (v,\ y')) \wedge r\ (v,\ x)\ (v,\ y):]$

lemma *feedback-simp-aaaa*: *feedback* $(\{.inpt\ r.\}\ o\ [:r:]) =$
$\{.\ \lambda x.((\exists\, u.\ inpt\ r\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u\ y.\ r\ (u,\ x)\ (a,\ y)) \longrightarrow inpt\ r\ (a,\ x))).\} \circ$
$[:x \rightsquigarrow y\ .\ (\exists\ v\ .\ \ r\ (v,\ x)\ (v,\ y)):]$

lemma *feedback-simp-aaaaa*: $p \leq inpt\ r \Longrightarrow$ *feedback* $(\{.p.\}\ o\ [:r:]) =$
$\{.\ \lambda x.((\exists\, u.\ p\ (u,\ x)) \wedge (\forall\, a.\ (\exists\, u\ y.\ p\ (u,\ x) \wedge r\ (u,\ x)\ (a,\ y)) \longrightarrow p\ (a,\ x))).\} \circ$
$[:x \rightsquigarrow y\ .\ (\exists\ v\ .\ \ p\ (v,\ x) \wedge r\ (v,\ x)\ (v,\ y)):]$

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

lemma *feedback-in-simp-a*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
feedback $(\ \{.\ u,\ x\ .\ p\ (u,\ x) \wedge p'\ x.\}\ o\ [:u,\ x \rightsquigarrow v,\ y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ v:])$
$= \{.\ x\ .\ p'\ x \wedge (\forall\, b.\ r'\ x\ b \longrightarrow p\ (b,x)).\}\ o\ [:x \rightsquigarrow y\ .\ \exists\ v\ .\ r'\ x\ v \wedge r\ (v,\ x)\ y:]$

lemma *feedback-in-simp-b*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
feedback $(\ \{.\ u,\ x\ .\ p\ (u,\ x) \wedge p'\ x.\}\ o\ [:u,\ x \rightsquigarrow v,\ y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ v:])$
$= \{.\ x\ .\ p'\ x \wedge (\forall\, b.\ r'\ x\ b \longrightarrow p\ (b,x)).\}\ o\ [:x \rightsquigarrow y\ .\ \exists\ v\ .\ r'\ x\ v \wedge r\ (v,\ x)\ y:]$
lemma $p \leq inpt\ r \Longrightarrow p'' \leq inpt\ r'' \Longrightarrow$ *feedback* $(\ (Skip\ **\ (\{.p.\}\ o\ [:r:]))\ o\ ([-\lambda(x,y)\ .\ (y,x)-])\ o\ (Skip\ **\ (\{.p''.\}\ o\ [:r'':])))\ )$

$= (\{.p.\}\ o\ [:r:])\ o\ (\{.p''.\}\ o\ [:r'':])$

lemma *feedback-update-simp-aaa*: $(\bigwedge u\ x.\ fst(f(u,x)) = fst(f(undefined,x))) \Longrightarrow$
$feedback(\{.p.\}\ o\ [-f-]) = \{.x.\ p(fst(f(undefined,\ x)),\ x).\} \circ [-\ \lambda x.\ snd(f(fst(f(undefined,x)),x))-]$

lemma *feedback-update-simp-bbb*: $(\bigwedge u\ x.\ fst(f(u,x)) = fst(f(undefined,x))) \Longrightarrow$
$feedback([-f-]) = [-\ \lambda x.\ snd(f(fst(f(undefined,x)),x))-]$

thm *feedback-def*

thm *feedback-in-simp-a*

definition *feedbackless* $S = (SOME\ T\ .\ \exists\ p\ f\ .\ S = \{.p.\}\ o\ [-f-] \wedge T = \{.x.\ p(fst(f(Eps\ (\lambda\ u\ .\ p\ (u,\ x)),\ x)),\ x).\} \circ [-\ \lambda x.\ snd(f(fst(f(Eps\ (\lambda\ u\ .\ p\ (u,\ x)),x)),x))-])$

definition *fstsome* $p\ x = Eps\ (\lambda\ u\ .\ p\ (u,\ x))$

definition *fbv* $p\ f\ x = fst(f(fstsome\ p\ x,\ x))$

definition *fb-prec* $p\ f\ x = p(fbv\ p\ f\ x,\ x)$
definition *fb-func* $p\ f\ x = snd(f(fbv\ p\ f\ x,\ x))$

lemma *fb-prec-simp*: *fb-prec* $p\ f = (\lambda\ x\ .\ p(fbv\ p\ f\ x,\ x))$

lemma *fb-func-simp*: *fb-func* $p\ f = (\lambda\ x\ .\ snd(f(fbv\ p\ f\ x,\ x)))$

lemma *feedbackless-update-simp-aaa*: $feedbackless(\{.p.\}\ o\ [-f-]) = \{.fb\text{-}prec\ p\ f.\} \circ [-\ fb\text{-}func\ p\ f\ -]$

lemma $(\bigwedge u\ x.\ fst(f(u,x)) = fst(f(undefined,x))) \Longrightarrow feedback(\{.p.\}\ o\ [-f-]) = feedbackless(\{.p.\}\ o\ [-f-])$

lemma *feedbackless-update-simp-bbb*: $feedbackless([-f-]) = [-\ fb\text{-}func\ \top\ f\ -]$

lemma *feedback-update-simp-ccc*: $feedback(\ \{.\bot.\}\ o\ [-f-]) = \bot$

1.8.1 Different Feedback Attempts

definition *select''* $S = [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge prec\ S\ (u,\ x)\ :]\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *selectb* $S = \{:\ x \rightsquigarrow u,\ x'.\ x = x' \wedge prec\ S\ (u,\ x):\}\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *selectd* $S = [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge prec\ S\ (u,\ x)\ :]\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *selecte* $S = [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge grd\ S\ (u,\ x)\ :]\ o\ S\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

definition *feedbackf* $S = \{.\ x\ .\ (\exists\ u\ .\ prec\ S\ (u,\ x)).\}\ o\ [:x \rightsquigarrow (u,\ x'),\ u'\ .\ x' = x \wedge u' = u \wedge prec\ S\ (u,\ x)\ :]$
$o\ (S\ **\ Skip)\ o\ [:(v,\ y),\ u \rightsquigarrow (v',\ y')\ .\ v = u \wedge v' = v \wedge y' = y:]$

definition *feedbackg* $S = [:x \rightsquigarrow (u,\ x'),\ u'\ .\ x' = x \wedge u' = u \wedge grd\ S\ (u,\ x)\ :]\ o\ (S\ **\ Skip)\ o\ [:(v,\ y),\ u \rightsquigarrow v',\ y'\ .\ v = u \wedge y' = y \wedge v' = v:]$

lemma *selectc''-spec*: $select''\ (\{.\ p\ .\}\ o\ [:r:]) = [:x \rightsquigarrow v\ .\ \exists\ u\ y\ .\ p\ (u,\ x) \wedge r\ (u,x)\ (v,y)\ :]$

lemma *selectcb-spec*: $selectb\ (\{.\ p\ .\}\ o\ [:r:]) = \{:\ x \rightsquigarrow u,x'.\ x = x' \wedge\ p\ (u,\ x):\}\ o\ [:u,\ x \rightsquigarrow v\ .\ \exists\ y\ .\ p\ (u,\ x) \wedge r\ (u,x)\ (v,y)\ :]$

lemma *feedbackf-spec*: $feedbackf\ (\{.\ p\ .\}\ o\ [:r:]) = \{.\ x\ .\ (\exists\ u\ .\ p\ (u,\ x)).\}\ o\ [:x \rightsquigarrow u,\ y\ .\ p\ (u,\ x) \wedge r\ (u,x)\ (u,y)\ :]$

lemma *feedbackg-spec*: $feedbackg\ (\{.\ p\ .\}\ o\ [:r:]) = \{.\ x\ .\ (\forall\ u\ .\ p\ (u,\ x)).\}\ o\ [:x \rightsquigarrow u,\ y\ .r\ (u,x)\ (u,y)\ :]$

lemma *selectd-spec*: $selectd\ (\{.\ p\ .\}\ o\ [:r:]) = [:x \rightsquigarrow u,\ x'\ .\ x' = x \wedge p\ (u,\ x)\ :]\ o\ [:r:]\ o\ [:v,\ y \rightsquigarrow v'\ .\ v' = v:]$

lemma *selecte-spec*: $selecte\ (\{.\ p\ .\}\ o\ [:r:]) = \{.x\ .\ \forall\ u\ .\ p\ (u,\ x).\}\ o\ [:x \rightsquigarrow v\ .\ \exists\ u\ y\ .\ r\ (u,\ x)\ (v,\ y):]$

definition $feedback'\ S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((select\ S)\ **\ Skip)\ o\ S\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$
definition $feedback''\ S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((select''\ S)\ **\ Skip)\ o\ S\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

definition $feedbacka\ S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((select\ S)\ **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])$
definition $feedbackb\ S = [:x \rightsquigarrow x',\ x''\ .\ x' = x \wedge x'' = x:]\ o\ ((selectb\ S)\ **\ Skip)\ o\ (S \parallel [:u,\ x \rightsquigarrow v,\ y\ .\ u = v:])\ o\ [:u,y \rightsquigarrow y'\ .\ y' = y:]$

lemma *feedback-simp-a-a*: $feedback'\ (\{.p.\}\ o\ [:r:]) =$
$\{.\ x.\ (\exists\, u.\ p\ (u,\ x))\ \wedge\ \ (\forall\, a.\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow p\ (a,\ x))\ .\} \circ$
$[:\ \lambda x\ y.\ \exists\, a\ aa.\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (aa,\ y))) \wedge r\ (aa,\ x)\ (a,\ y)\ :]$

lemma *feedback-simp-a-b*: $feedback''\ (\{.p.\}\ o\ [:r:]) =$
$\{.\ \lambda x.\ \forall\, a.\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow p\ (a,\ x)\ .\} \circ [:\ \lambda x\ y.\ \exists\, a\ aa.\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (aa,\ y))) \wedge r\ (aa,\ x)\ (a,\ y)\ :]$

lemma *feedbackb-simp-a*: $feedbackb\ (\{.p.\}\ o\ [:r:]) =$
$\{:x \rightsquigarrow u,\ x'\ .\ x = x' \wedge p\ (u,\ x) \wedge (\forall\, a.\ ((\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow p\ (a,\ x))\ :\} \circ$
$[:u,\ x \rightsquigarrow y\ \ .\ (\exists\ \ v\ .(\ (\exists\, y.\ r\ (u,\ x)\ (v,\ y))) \wedge\ \ r\ (v,\ x)\ (v,\ y)):]$

lemma *feedbackb-simp-aa*: $feedbackb\ (\{.inpt\ r.\}\ o\ [:r:]) =$
$\{:x \rightsquigarrow u,\ x'\ .\ x = x' \wedge inpt\ r\ (u,\ x) \wedge (\forall\, a.\ ((\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow inpt\ r\ (a,\ x))\ :\} \circ$
$[:u,\ x \rightsquigarrow y\ \ .\ (\exists\ \ v\ .(\ (\exists\, y.\ r\ (u,\ x)\ (v,\ y))) \wedge\ \ r\ (v,\ x)\ (v,\ y)):]$

lemma *feedbacka-simp-a*: $feedbacka\ (\{.p.\}\ o\ [:r:]) =$
$\{.\ \lambda x.\ (\exists\, u.\ p\ (u,\ x))\ \wedge\ (\forall\, a.\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (a,\ y))) \longrightarrow p\ (a,\ x))\ .\} \circ$
$[:\ \lambda x\ (v,\ y)\ .\ (\exists\, u.\ p\ (u,\ x) \wedge (\exists\, y.\ r\ (u,\ x)\ (v,\ y))) \wedge r\ (v,\ x)\ (v,\ y):]$

lemma *feedback-in-simp-a-a*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
$feedback'\ (\ \{.\ u,\ x\ .\ p\ (u,\ x) \wedge p'\ x.\}\ o\ [:u,\ x \rightsquigarrow v,\ y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ v:])$
$= \{.\ x\ .\ p'\ x \wedge (\forall\, b.\ r'\ x\ b \longrightarrow p\ (b,x)).\}\ o\ [:x \rightsquigarrow y\ .\ \exists\ \ v\ .\ r'\ x\ v \wedge r\ (v,\ x)\ y:]$

lemma *feedbacka-in-simp-a*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
$feedbacka\ (\ \{.\ u,\ x\ .\ p\ (u,\ x) \wedge p'\ x.\}\ o\ [:u,\ x \rightsquigarrow v,\ y\ .\ r\ (u,\ x)\ y \wedge r'\ x\ v:])$

$= \{.\ x\ .\ p'\ x \wedge (\forall\, b.\ r'\ x\ b \longrightarrow p\ (b,x)).\}\ o\ [:x \rightsquigarrow v,\ y\ .\ r'\ x\ v \wedge r\ (v,\ x)\ y:]$

lemma *feedbacka-simp-b*: *feedbacka* $[:r:] = [:\ x \rightsquigarrow v,\ y\ .\ r\ (v,\ x)\ (v,\ y):]$

1.8.2 Feedback of Decomposable Components

definition *decomposable* $r\ r'\ r'' = (\forall\ u\ x\ v\ y\ .\ r\ (u,\ x)\ (v,\ y) = ((r'\ x\ v) \wedge r''\ (u,\ x)\ y))$

lemma *decomposabel-iff*: $(\exists\ r'\ r''\ .\ decomposable\ r\ r'\ r'') = ((\forall\ u\ x\ v\ y\ .\ r\ (u,\ x)\ (v,\ y) = ((\exists\ u\ y\ .\ r\ (u,\ x)\ (v,\ y)) \wedge (\exists\ v\ .\ r\ (u,\ x)\ (v,\ y)))\ ))$

lemma *decomposable-calc*: $(\exists\ r'\ r''\ .\ decomposable\ r\ r'\ r'') \Longrightarrow decomposable\ r\ (\lambda\ x\ v\ .\ (\exists\ y\ u'\ .\ r\ (u',\ x)\ (v,\ y)))\ (\lambda\ (u,x)\ y\ .\ (\exists\ v\ .\ r\ (u,x)\ (v,\ y)))$

lemma *decomposable-inpt*: $decomposable\ r\ r'\ r'' \Longrightarrow inpt\ r\ (u,\ x) = (inpt\ r'\ x \wedge inpt\ r''\ (u,\ x))$

lemma *decomposable-feedback-trs*: $decomposable\ r\ r'\ r'' \Longrightarrow feedback\ (trs\ r)$
$= \{.\ x\ .\ inpt\ r'\ x \wedge (\forall\, b.\ r'\ x\ b \longrightarrow inpt\ r''\ (b,\ x)).\} \circ [:x \rightsquigarrow y.\exists\, v.\ r'\ x\ v \wedge r''\ (v,\ x)\ y:]$

lemma *spec-eq*: $(\bigwedge\ x\ .\ p\ x = p'\ x) \Longrightarrow (\bigwedge\ x\ y\ .\ p\ x \Longrightarrow r\ x\ y = r'\ x\ y) \Longrightarrow \{.p.\}\ o\ [:r:] = \{.p'.\}\ o\ [:r':]$

theorem $decomposable\ r\ r'\ r'' \Longrightarrow\ feedback\ (trs\ r)$
$= trs\ (\lambda\ x\ y\ .\ (\forall\, v.\ r'\ x\ v \longrightarrow inpt\ r''\ (v,\ x)) \wedge (\exists\, v.\ r'\ x\ v \wedge r''\ (v,\ x)\ y))$

lemma $[simp]$: $((\exists\, u.\ p\ u\ x) \wedge (\exists\, v.\ Ex\ (r\ v)) \wedge (\forall\, a.\ (\exists\, u.\ p\ u\ x) \wedge (\exists\, v.\ Ex\ (r\ v)) \wedge Ex\ (r\ a) \longrightarrow p\ a\ x))$
$= (((\exists\, v.\ Ex\ (r\ v)) \wedge (\forall\ a\ .\ Ex\ (r\ a) \longrightarrow p\ a\ x)))$

definition $Decomposable\ r = (\exists\ r'\ r''\ .\ decomposable\ r\ r'\ r'')$

definition *fst-dec* $r = (\lambda\ x\ v\ .\ \exists\ u\ y\ .\ r\ (u,\ x)\ (v,\ y))$
definition *snd-dec* $r = (\lambda\ (u,\ x)\ y\ .\ \exists\ v\ .\ r\ (u,\ x)\ (v,\ y))$

lemma *decomosable-fst-snd*: $Decomposable\ r = (decomposable\ r\ (fst\text{-}dec\ r)\ (snd\text{-}dec\ r))$

definition *state-determ* $r = (\forall\ x\ y\ y'\ s\ s'\ s''\ .\ r\ (s,\ x)\ (s',\ y) \wedge r\ (s,\ x)\ (s'',\ y') \longrightarrow s' = s'')$

lemma *decomposable-and*: $decomposable\ r\ r'\ r'' \Longrightarrow decomposable\ (\lambda\ (u,\ x)\ (v,\ y)\ .\ p(u,\ x) \wedge r\ (u,x)\ (v,\ y))\ r'\ (\lambda\ (u,x)\ y\ .\ p\ (u,\ x) \wedge r''\ (u,\ x)\ y)$

end

2 Complete Distributive Lattice

theory *Distributive* **imports** *Main*
begin

notation
bot ($\bot$) **and**

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

context *complete-lattice*
begin
lemma *Sup-Inf-le*: *Sup* (*Inf* ‘ {*f* ‘ *A* | *f* . (∀ *Y* ∈ *A* . *f Y* ∈ *Y*)}) ≤ *Inf* (*Sup* ‘ *A*)
end

class *complete-distributive-lattice* = *complete-lattice* +
assumes *Inf-Sup-le*: *Inf* (*Sup* ‘ *A*) ≤ *Sup* (*Inf* ‘ {*f* ‘ *A* | *f* . (∀ *Y* ∈ *A* . *f Y* ∈ *Y*)})
begin

lemma *Inf-Sup*: *Inf* (*Sup* ‘ *A*) = *Sup* (*Inf* ‘ {*f* ‘ *A* | *f* . (∀ *Y* ∈ *A* . *f Y* ∈ *Y*)})

lemma *Sup-Inf*: *Sup* (*Inf* ‘ *A*) = *Inf* (*Sup* ‘ {*f* ‘ *A* | *f* . (∀ *Y* ∈ *A* . *f Y* ∈ *Y*)})

lemma *dual-complete-distributive-lattice*:
class.complete-distributive-lattice Sup Inf sup (*op* ≥) (*op* >) *inf* ⊤ ⊥

lemma *sup-Inf*: *a* ⊔ *Inf B* = (*INF b*:*B*. *a* ⊔ *b*)

lemma *inf-Sup*: *a* ⊓ *Sup B* = (*SUP b*:*B*. *a* ⊓ *b*)

subclass *complete-distrib-lattice*

end

instantiation *bool* :: *complete-distributive-lattice*
begin
instance
end

instantiation *set* :: (*type*) *complete-distributive-lattice*
begin
instance
end

context *complete-distributive-lattice*
begin

lemma *INF-SUP*: (*INF y*. *SUP x*. ((*P x y*)::′*a*)) = (*SUP x*. *INF y*. *P* (*x y*) *y*)

end

instantiation *fun* :: (*type*, *complete-distributive-lattice*) *complete-distributive-lattice*
begin

instance

end

context *complete-linorder*

begin

subclass *complete-distributive-lattice*

end

end

3 Linear Temporal Logic

theory *Temporal* **imports** *Distributive*
begin

In this section we introduce an algebraic axiomatization of Linear Temporal Logic (LTL). We model LTL formulas semantically as predicates on traces. For example the LTL formula $\alpha = \Box \diamondsuit (x = 1)$ is modeled as a predicate $\alpha : (nat \Rightarrow nat) \Rightarrow bool$, where $\alpha\ x = True$ if $x\ i = 1$ for infinitely many $i : nat$. In this formula $\Box$ and $\diamondsuit$ denote the always and eventually operators, respectively. Formulas with multiple variables are modeled similarly. For example a formula α in two variables is modeled as $\alpha : (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow bool$, and for example $(\Box\ \alpha)\ x\ y$ is defined as $(\forall i.\alpha\ x[i..]\ y[i..])$, where $x[i..]\ j = x\ (i + j)$. We would like to construct an algebraic structure (Isabelle class) which has the temporal operators as operations, and which has instatiations to $(nat \Rightarrow 'a) \Rightarrow bool$, $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow bool$, and so on. Ideally our structure should be such that if we have this structure on a type $'a :: temporal$, then we could extend it to $(nat \Rightarrow 'b) \Rightarrow 'a$ in a way similar to the way Boolean algebras are extended from a type $'a :: boolean\_algebra$ to $'b \Rightarrow 'a$. Unfortunately, if we use for example $\Box$ as primitive operation on our temporal structure, then we cannot extend $\Box$ from $'a :: temporal$ to $(nat \Rightarrow 'b) \Rightarrow 'a$. A possible extension of $\Box$ could be

$$(\Box\ \alpha)\ x = \bigwedge_{i:nat} \Box(\alpha\ x[i..]) \text{ and } \Box\ b = b$$

where $\alpha : (nat \Rightarrow 'b) \Rightarrow 'a$ and $b : bool$. However, if we apply this definition to $\alpha : (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow bool$, then we get

$$(\Box\ \alpha)\ x\ y = (\forall i\ j.\alpha\ x[i..]\ y[j..])$$

which is not correct.

To overcome this problem we introduce as a primitive operation $!! : 'a \Rightarrow nat \Rightarrow 'a$, where $'a$ is the type of temporal formulas, and $\alpha!!i$ is the formula α at time point i. If α is a formula in two variables as before, then

$$(\alpha!!i)\ x\ y = \alpha\ x[i..]\ y[i..].$$

and we define for example the the operator always by

$$\Box\alpha = \bigwedge_{i:nat} \alpha!!i$$

class *temporal* = *complete-boolean-algebra* + *complete-distributive-lattice* +
  fixes *at* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixl** !! *150*)
  assumes [*simp*]: $a\ !!\ i\ !!\ j = a\ !!\ (i + j)$
  assumes [*simp*]: $a\ !!\ 0 = a$

assumes [*simp*]: $-(a \mathbin{!!} i) = (-a) \mathbin{!!} i$
assumes *Inf-at*[*simp*]: $(Inf\ X) \mathbin{!!} i = (INFIMUM\ X\ (\lambda\ x\ .\ at\ x\ i))$
begin
lemma [*simp*]: $\top \mathbin{!!} i = \top$

lemma *Sup-at*: $(Sup\ X) \mathbin{!!} i = (SUPREMUM\ X\ (\lambda\ x\ .\ x \mathbin{!!} i))$

lemma [*simp*]: $(a \sqcap b) \mathbin{!!} i = (a \mathbin{!!} i) \sqcap (b \mathbin{!!} i)$

lemma [*simp*]: $(INF\ x{:}X.\ f\ x) \mathbin{!!} i = (INF\ x{:}X.\ f\ x \mathbin{!!} i)$

definition *always* :: $'a \Rightarrow 'a$ $(\square\ (\text{-})\ [900]\ 900)$ **where**
$\square\ p = (INF\ i\ .\ p \mathbin{!!} i)$

definition *eventually-bounded* :: *nat set* $\Rightarrow 'a \Rightarrow 'a$ $(\diamondsuit b\ (\text{-})\ (\text{-})\ [900{,}900]\ 900)$ **where**
$\diamondsuit b\ b\ p = (SUP\ i{:}\ b\ .\ p \mathbin{!!} i)$

definition *always-bounded* :: *nat set* $\Rightarrow 'a \Rightarrow 'a$ $(\square b\ (\text{-})\ (\text{-})\ [900{,}900]\ 900)$ **where**
$\square b\ b\ p = (INF\ i{:}\ b\ .\ p \mathbin{!!} i)$

lemma $(\square b\ b\ p) \sqcap (\square b\ b'\ p) = (\square b\ (b \cup b')\ p)$

definition *eventually* :: $'a \Rightarrow 'a$ $(\diamondsuit\ (\text{-})\ [900]\ 900)$ **where**
$\diamondsuit\ p = (SUP\ i\ .\ p \mathbin{!!} i)$

definition *next* :: $'a \Rightarrow 'a$ $(\bigodot\ (\text{-})\ [900]\ 900)$ **where**
$\bigodot\ p = p \mathbin{!!} (Suc\ 0)$

definition *until* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** *until 65*) **where**
$(p\ until\ q) = (SUP\ n\ .\ (INFIMUM\ \{i\ .\ i < n\}\ \ (at\ p)) \sqcap (q \mathbin{!!} n))$

definition *leads* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** *leads 65*) **where**
$(p\ leads\ q) = -(p\ until\ -q)$

lemma *iterate-next*: $(next\ \hat{}\hat{}\ n)\ p = p \mathbin{!!} n$

lemma *always-next*: $\square\ p = p \sqcap (\square\ (\bigodot\ p))$
end

Next lemma, in the context of complete boolean algebras, will be used to prove $-(p\ until\ -p) = \square\ p$.

context *complete-boolean-algebra*
begin
lemma *until-always*: $(INF\ n.\ (SUP\ i : \{i.\ i < n\}\ .\ -\ p\ i) \sqcup ((p :: nat \Rightarrow 'a)\ n)) \leq p\ n$

end

We prove now a number of results of the temporal class.

context *temporal*
begin
lemma [*simp*]: $(a \sqcup b) \mathbin{!!} i = (a \mathbin{!!} i) \sqcup (b \mathbin{!!} i)$

lemma *always-less* [*simp*]: $\square\ p \leq p$

lemma *always-always*: $\square\ \square\ x = \square\ x$

lemma *always-and*: $\Box\ (p \sqcap q) = (\Box\ p) \sqcap (\Box\ q)$

lemma *eventually-or*: $\Diamond\ (p \sqcup q) = (\Diamond\ p) \sqcup (\Diamond\ q)$

lemma *neg-until-always*: $-(p\ until\ -p) = \Box\ p$

lemma *leads-always*: $p\ leads\ p = \Box\ p$

lemma *neg-always-eventually*: $\Box\ p = -\ \Diamond\ (-\ p)$

lemma *neg-true-until-always*: $-(\top\ until\ -p) = \Box\ p$

lemma *top-leads-always*: $\top\ leads\ p = \Box\ p$

lemma *neg-until-true*: $-(p\ until\ -\top) = \top$

lemma *leads-top*: $p\ leads\ \top = \top$

lemma *neg-until-false*: $-(p\ until\ -\bot) = \bot$

lemma *leads-bot*: $p\ leads\ \bot = \bot$

lemma *true-until-eventually*: $(\top\ until\ p) = \Diamond\ p$

end

Boolean algebras with $b!!i = b$ form a temporal class.

```
instantiation bool :: temporal
  begin
    definition at-bool-def [simp]: (p::bool) !! i = p
  instance
  end

type-synonym 'a trace = nat ⇒ 'a
```

Asumming that $'a :: temporal$ is a type of class *temporal*, and $'b$ is an arbitrary type, we would like to create the instantiation of $'b\ trace \Rightarrow 'a$ as a temporal class. However Isabelle allows only instatiations of functions from a class to another class. To solve this problem we introduce a new class called trace with an operation $suffix :: 'a \Rightarrow nat \Rightarrow 'a$ where $suffix\ a\ i\ j = (a[i..])\ j = a\ (i + j)$ when a is a trace with elements of some type $'b$ ($'a = nat \Rightarrow 'b$).

```
class trace =
  fixes suffix :: 'a ⇒ nat ⇒ 'a (-[- ..] [80, 15] 80)
  fixes eqtop :: nat ⇒ 'a ⇒ 'a ⇒ bool
  fixes cat :: nat ⇒ 'a ⇒ 'a ⇒ 'a
  fixes Cat :: (nat ⇒ 'a) ⇒ 'a
  assumes suffix-suffix[simp]: a[i..][j..] = a[i + j..]
  assumes [simp]: a[0..] = a
  assumes [simp]: eqtop 0 a b = True
  assumes [simp]: eqtop n a a = True
  assumes all-eqtop[simp]: ∀ n . eqtop n a b ⟹ a = b
  assumes eqtop-sym: eqtop n a b =  eqtop n b a
  assumes eqtop-tran: eqtop n a b ⟹  eqtop n b c ⟹ eqtop n a c
  assumes [simp]: eqtop n (cat n x y) z = eqtop n x z
  assumes cat-at-eq[simp]: (cat n x y)[n..] = y
```

assumes *eqtop-Suc*: $eqtop\ (Suc\ n)\ x\ y = (eqtop\ n\ x\ y \wedge eqtop\ (Suc\ 0)\ (x[n..])\ (y[n..]))$
assumes *Cat-Suc*: $Cat\ u = cat\ (Suc\ 0)\ (u\ 0)\ (Cat\ (\lambda\ i\ .\ u\ (Suc\ i)))$
assumes *cat-Suc*: $cat\ (Suc\ n)\ x\ y = cat\ (Suc\ 0)\ x\ (cat\ n\ (x[Suc\ 0..])\ y)$
assumes *cat-Zero*[*simp*]: $cat\ 0\ x\ y = y$

begin
definition *next-trace* :: $'a \Rightarrow 'a$ $(\odot\ (-)\ [900]\ 900)$ **where**
$\odot\ p = p[Suc\ 0..]$

lemma *eq-le*[*simp*]: $\bigwedge\ a\ b\ .\ n \leq m \Longrightarrow eqtop\ m\ a\ b \Longrightarrow eqtop\ n\ a\ b$

lemma *eqtop-Suc-Cat*: $\bigwedge\ u\ .\ eqtop\ (Suc\ 0)\ ((Cat\ u)[n..])\ (u\ n)$

lemma *eqtop-tail-eqtop*: $eqtop\ n\ x\ y \Longrightarrow x[n\ ..] = y[n\ ..] \Longrightarrow eqtop\ na\ x\ y$

lemma [*simp*]: $eqtop\ n\ z\ (cat\ n\ x\ y) = eqtop\ n\ z\ x$

lemma *eqtop-tail*: $eqtop\ n\ x\ y \Longrightarrow x[n..] = y[n..] \Longrightarrow x = y$

definition $cons\ x = cat\ (Suc\ 0)\ x\ x$

lemma [*simp*]: $(cons\ a)[Suc\ 0..] = a$

lemma [*simp*]: $eqtop\ 0 = \top$

lemma [*simp*]: $eqtop\ n\ x\ (cat\ n\ x\ y)$

lemma [*simp*]: $\exists\, y.\ x = y[Suc\ 0\ ..]$

lemma *eqtop-plus*: $\bigwedge\ x\ y\ .\ (eqtop\ n\ x\ y \wedge (eqtop\ m\ (x[n..])\ (y[n..]))) = eqtop\ (n + m)\ x\ y$

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

lemma [*simp*]: $cat\ n\ x\ (x[n..]) = x$

lemma *eqtop-Suc-a*: $eqtop\ (Suc\ n)\ x\ y = (eqtop\ (Suc\ 0)\ x\ y \wedge eqtop\ n\ (x[Suc\ 0\ ..])\ (y[Suc\ 0\ ..]))$

lemma *cat-Suc-b*: $\bigwedge\ x\ y\ .\ cat\ (Suc\ n)\ x\ y = cat\ n\ x\ (cat\ (Suc\ 0)\ (x[n..])\ y)$

lemma *cat-at*: $\bigwedge\ i\ x\ y\ .\ i \leq n \Longrightarrow (cat\ n\ x\ y[i..]) = cat\ (n - i)\ (x[i..])\ y$

lemma *eqtop-cat-le*: $\bigwedge\ m\ x\ y\ z\ .\ m \leq n \Longrightarrow eqtop\ m\ (cat\ n\ x\ y)\ z = eqtop\ m\ x\ z$

lemma *eqtop-cat-aux*: $i < n \Longrightarrow eqtop\ (Suc\ 0)\ (cat\ n\ x\ y[i..])\ (x[i..])$

end

instantiation *prod* :: (*trace*, *trace*) *trace*
begin

definition *at-prod-def*: $x[i..] \equiv ((fst\ x)[i..], (snd\ x)[i..])$
definition *eqtop-prod-def*: $eqtop\ n\ x\ y \equiv eqtop\ n\ (fst\ x)\ (fst\ y) \wedge eqtop\ n\ (snd\ x)\ (snd\ y)$
definition *cat-prod-def*: $cat\ n\ x\ y \equiv (cat\ n\ (fst\ x)\ (fst\ y), cat\ n\ (snd\ x)\ (snd\ y))$
definition *Cat-prod-def*: $Cat\ u \equiv (Cat\ (fst\ o\ u), Cat\ (snd\ o\ u))$

instance

end

instantiation *fun* :: (*trace*, *temporal*) *temporal*
begin
definition *at-fun-def*: $(P::'a \Rightarrow 'b)\ !!\ i = (\lambda\ x\ .\ (P\ (x[i..]))\ !!\ i)$
instance
end

lemma *SUP-Suc*: $(SUP\ x{:}\{i.\ i < Suc\ n\}.\ p\ x) = (SUP\ x{:}\{i.\ i < n\}.\ p\ x) \sqcup ((p\ n)::'a::complete\text{-}lattice)$

definition *top-dep* $p = (\forall\ x\ x'\ .\ eqtop\ (Suc\ 0)\ x\ x' \longrightarrow p\ x = p\ x')$

lemma *INF-distrib*: $(INF\ x\ y.\ p\ x \sqcup ((q\ y)::'a::complete\text{-}distrib\text{-}lattice)) = (INF\ x\ .\ p\ x) \sqcup (INF\ y\ .\ q\ y)$

lemma *top-dep-INF-SUP*: $top\text{-}dep\ p \Longrightarrow\ (INF\ x.\ (SUP\ xa{:}\{i.\ i < n\}.\ (-\ p\ (x[xa\ ..]))\ !!\ xa) \sqcup (-\ p\ (x[n\ ..]))\ !!\ n) =$
$(INF\ x\ y.\ (SUP\ xa{:}\{i.\ i < n\}.\ (-\ p\ (x[xa\ ..]))\ !!\ xa) \sqcup (-\ p\ y)\ !!\ n)$

lemma *top-dep-all-leadsto-aux*: $top\text{-}dep\ p \Longrightarrow (INF\ b.\ SUP\ x{:}\{i.\ i < n\}.\ (-\ p\ (b[x\ ..]))\ !!\ x) \leq (SUP\ x{:}\{i.\ i < n\}.\ INF\ xa.\ (-\ p\ xa)\ !!\ x)$

theorem *top-dep-all-leadsto*: $top\text{-}dep\ p \Longrightarrow INFIMUM\ UNIV\ (p\ leads\ (\lambda\ y\ .\ q)) = ((SUPREMUM\ UNIV\ p)\ leads\ q)$

theorem *SUP-Always*: $top\text{-}dep\ p \Longrightarrow SUPREMUM\ UNIV\ (\Box\ p) = \Box\ (SUPREMUM\ UNIV\ (p::('b::trace) \Rightarrow 'a::temporal))$

In the last part of our formalization, we need to instantiate the functions from *nat* to some arbitrary type $'a$ as a trace class. However, this again is not possible using the instatiation mechanism of Isabelle. We solve this problem by creating another class called *nat*, and then we instatiate the functions from $'a :: nat$ to $'b$ as traces. The class *nat* is defined such that if we have a type $'a :: nat$, then $'a$ is isomorphic to the type *nat*.

class *nat* = *zero* + *plus* + *minus* + *one* +
fixes *RepNat* :: $'a \Rightarrow nat$
fixes *AbsNat* :: $nat \Rightarrow 'a$
assumes *RepAbsNat*[*simp*]: $RepNat\ (AbsNat\ n) = n$
and *AbsRepNat*[*simp*]: $AbsNat\ (RepNat\ x) = x$
and *zero-Nat-def*: $0 = AbsNat\ 0$

and *one-Nat-def*: $1 = AbsNat\ 1$
and *plus-Nat-def*: $a + b = AbsNat\ (RepNat\ a + RepNat\ b)$
and *minus-Nat-def*: $a - b = AbsNat\ (RepNat\ a - RepNat\ b)$
begin
lemma *AbsNat-plus*: $AbsNat\ (i + j) = AbsNat\ i + AbsNat\ j$
lemma *AbsNat-minus*: $AbsNat\ (i - j) = AbsNat\ i - AbsNat\ j$
lemma *AbsNat-zero* [*simp*]: $AbsNat\ 0 + i = i$
lemma [*simp*]: $(AbsNat\ (Suc\ 0) + x = 0) = False$

subclass *comm-monoid-diff*
end

The type natural numbers is an instantiation of the class *nat*.

instantiation *nat* :: *nat*
begin
definition *RepNat-nat-def* [*simp*]: $(RepNat :: nat \Rightarrow nat) = id$
definition *AbsNat-nat-def* [*simp*]: $(AbsNat :: nat \Rightarrow nat) = id$
instance
end

Finally, functions from $'a :: nat$ to some arbitrary type $'b$ are instatiated as a trace class.

instantiation *fun* :: (*nat*, *type*) *trace*
begin
definition *at-trace-def* [*simp*]: $((t :: 'a \Rightarrow 'b)[i..])\ j = (t\ (AbsNat\ i + j))$
definition *eqtop-trace-def* [*simp*]: $eqtop\ n\ a\ b = (\forall\ i < n\ .\ a\ (AbsNat\ i) = b\ (AbsNat\ i))$
definition *cat-trace-def* [*simp*]: $cat\ n\ a\ b\ i = (if\ RepNat\ i < n\ then\ a\ i\ else\ b\ (i - AbsNat\ n))$
definition *Cat-trace-def* [*simp*]: $Cat\ y\ i = (y\ (RepNat\ i)\ 0)$
lemma *eqtop-trace-eq*: $\forall n\ i.\ i < n \longrightarrow (a::'a \Rightarrow 'b)\ (AbsNat\ i) = b\ (AbsNat\ i) \Longrightarrow a = b$

lemma [*simp*]: $(RepNat\ (AbsNat\ n + xa) < n) = False$

lemma [*simp*]: $AbsNat\ n + AbsNat\ 0 = AbsNat\ n$

lemma *trace-eqtop-tail*: $\forall i<n.\ x\ (AbsNat\ i) = y\ (AbsNat\ i) \Longrightarrow \forall xa.\ x\ (AbsNat\ n + xa) = y\ (AbsNat\ n + xa) \Longrightarrow x\ xa = y\ xa$

lemma *trace-eqtop-Suc*: $\forall i<n.\ x\ (AbsNat\ i) = y\ (AbsNat\ i) \Longrightarrow x\ (AbsNat\ n) = y\ (AbsNat\ n) \Longrightarrow i < Suc\ n \Longrightarrow x\ (AbsNat\ i) = y\ (AbsNat\ i)$

lemma *RepNat-is-zero*: $RepNat\ x = 0 \Longrightarrow x = 0$

lemma *RepNat-zero*: $RepNat\ x = 0 \Longrightarrow u\ 0\ x = u\ 0\ 0$

lemma [*simp*]: $0 < RepNat\ x \Longrightarrow (Suc\ (RepNat\ (x - AbsNat\ (Suc\ 0)))) = RepNat\ x$

instance
end

By putting together all class definitions and instatiations introduced so far, we obtain the temporal class structure for predicates on traces with arbitrary number of parameters.

For example in the next lemma r and r' are predicate relations, and the operator always is available for them as a consequence of the above construction.

lemma $(\Box\ r)\ OO\ (\Box\ r') \leq (\Box\ (r\ OO\ r'))$

lemma [*simp*]: $(next \ \hat{} \hat{} \ n) \ \top = \top$

lemma $r \ (u[1..]) = (\exists \ y \ . \ (\bigodot \ (\lambda \ v \ . \ v = y \wedge r \ y)) \ u)$

lemma $r \ (u[1..]) = (\ (\bigodot \ (\lambda \ v \ . \ \exists \ y \ . \ v = y \wedge r \ y)) \ u)$

lemma $(r \ (u[1..])::bool) = (\ (\bigodot \ r) \ u)$

lemma $((\Box \ r) \ u \ (u[1..]) \ x \ y \ ::bool) = (\ (\bigodot \ (\lambda \ u' \ . \ (\Box \ r) \ u \ u' \ x \ y)) \ u)$

lemma $r \ (u[1..]) = (\exists \ y \ . \ (\bigodot \ (\lambda \ v \ y \ . \ v = y \wedge r \ \ y)) \ u \ y)$

3.1 Propositional Temporal Logic

definition $prop \ P \ \sigma = (P \in \sigma \ (0::nat))$

definition $Exists \ P \ f \ \sigma = (\exists \ \sigma' \ . \ (\forall \ i \ . \ \sigma \ i - \{P\} = \sigma' \ i - \{P\}) \wedge f \ \sigma')$
definition $Forall \ P \ f \ \sigma = (\forall \ \sigma' \ . \ (\forall \ i \ . \ \sigma \ i - \{P\} = \sigma' \ i - \{P\}) \longrightarrow f \ \sigma')$
definition $impl:: \ 'a \Rightarrow 'a \Rightarrow ('a::boolean\text{-}algebra)$ (**infixl** $\rightarrow$ *60*)
where $x \rightarrow y = ((-x) \sqcup y)$

lemma $x \neq y \Longrightarrow (Exists \ y \ ((\Box \ (prop \ x \rightarrow (\diamondsuit \ prop \ y))) \sqcap \Box \diamondsuit \ prop \ y)) = \top$

lemma $x \neq y \Longrightarrow (Forall \ y \ ((\Box \ (prop \ x \rightarrow (\diamondsuit \ prop \ y))) \rightarrow \Box \diamondsuit \ prop \ y)) = (\Box \diamondsuit \ (prop \ x))$

end

4 Monotonic Property Transformers

theory *RefinementReactive*
imports *Temporal Refinement*
begin

In this section we introduce reactive systems which are modeled as monotonic property transformers where properties are predicates on traces. We start with introducing some examples that uses LTL to specify global behaviour on traces, and later we introduce property transformers based on symbolic transition systems.

definition $HAVOC = [:x \rightsquigarrow y \ . \ True:]$
definition $ASSERT\text{-}LIVE = \{. \Box \diamondsuit (\lambda \ x \ . \ x \ 0).\}$
definition $GUARANTY\text{-}LIVE = [:x \rightsquigarrow y \ . \ \Box \diamondsuit (\lambda \ y \ . \ y \ 0):]$
definition $AE = ASSERT\text{-}LIVE \ o \ HAVOC$
definition $SKIP = [:x \rightsquigarrow y \ . \ x = y:]$

lemma [*simp*]: $SKIP = id$

definition $REQ\text{-}RESP = [: \Box(\lambda \ xs \ ys \ . \ xs \ (0::nat) \longrightarrow (\diamondsuit \ (\lambda \ ys \ . \ ys \ (0::nat))) \ ys) :]$

definition *FAIL* = $\bot$

lemma *HAVOC o ASSERT-LIVE = FAIL*

lemma *HAVOC o AE = FAIL*

lemma *HAVOC o ASSERT-LIVE = FAIL*

lemma *SKIP o AE = AE*

lemma (*REQ-RESP o AE*) = *AE*

4.1 Symbolic transition systems

In this section we introduce property transformers basend on symbolic transition systems. These are systems with local state. The execution starts in some initial state, and with some input value the system computes a new state and an output value. Then using the current state, and a new input value the system computes a new state, and a new output, and so on. The system may fail if at some point the input and the current state do not statisfy a required predicate.

In the folowing definitions the variables u, x, y stand for the state of the system, the input, and the output respectively. The *init* is the property that the initial state should satisfy. The predicate p is the precondition of the input and the current state, and the relation r gives the next state and the output based on the input and the current state.

definition *illegal-sts init p r x* = ($\exists$ *n u y . init* (*u 0*) $\wedge$ ($\forall$ *i* < *n . r* (*u i*, *x i*) (*u* (*Suc i*), *y i*)) $\wedge$ ($\neg$ *p* (*u n*, *x n*)))
definition *run-sts r u x y* = ($\forall$ *i . r* (*u i*, *x i*) (*u* (*Suc i*), *y i*))

definition *LocalSystem init p r q x* = ($\neg$ *illegal-sts init p r x* $\wedge$ ($\forall$ *u y* . (*init* (*u 0*) $\wedge$ *run-sts r u x y*) $\longrightarrow$ *q y*))

lemma *LocalSystem-not-fail-run*: *LocalSystem init p r* = {.$-$ *illegal-sts init p r*.} *o* [:*x* $\rightsquigarrow$ *y* . $\exists$ *u* . *init* (*u 0*) $\wedge$ *run-sts r u x y*:]

definition *fail-sys-delete init p r x* = ($\exists$ *n u y . u* $\in$ *init* $\wedge$ ($\forall$ *i* < *n . r* (*u i*) (*u* (*Suc i*)) (*x i*) (*y i*)) $\wedge$ ($\neg$ *p* (*u n*) (*u* (*Suc n*)) (*x n*)))
definition *run-delete r u x y* = ($\forall$ *i . r* (*u i*) (*u* (*Suc i*)) (*x i*) (*y i*))

definition *LocalSystem-delete init p r q x* = ($\neg$ *fail-sys-delete init p r x* $\wedge$ ($\forall$ *u y* . (*u* $\in$ *init* $\wedge$ *run-delete r u x y*) $\longrightarrow$ *q y*))

lemma *fail* (*LocalSystem init p r*) = *illegal-sts init p r*

definition *lift-pre p* = (λ (*u*, *x*) (*u*′, *y*) . *p* (*u* (*0*::*nat*), *x* (*0*::*nat*)))
definition *lift-rel r* = (λ (*u*, *x*) (*u*′, *y*) . *r* (*u* (*0*::*nat*), *x* (*0*::*nat*)) (*u*′ *0*, *y* (*0*::*nat*)))

definition *prec-pre-sts init p r x* = ($\forall$ *u y . init* (*u 0*) $\longrightarrow$ (*lift-rel r leads lift-pre p*) (*u*, *x*) (*u*[*1*..], *y*))
definition *rel-pre-sts init r x y* = ($\exists$ *u . init* (*u 0*) $\wedge$ ($\Box$ *lift-rel r*) (*u*, *x*) (*u*[*1*..], *y*))

lemma *prec-pre-sts-simp*: *prec-pre-sts init p r x* = ($\forall$ *u y . init* (*u 0*) $\longrightarrow$ ($\forall$ *n* . ($\forall$ *i* < *n . r* (*u i*, *x*

$i)\ (u\ (Suc\ i),\ y\ i)) \longrightarrow p\ (u\ n,\ x\ n)))$

lemma *prec-stateless-sts-simp*: *prec-pre-sts* $\top$ $(\lambda\ (s{::}unit,\ x)\ .\ inpt\ r\ x)\ (\lambda\ (s{::}unit,\ x)\ (s'{::}unit,\ y)\ .\ r\ x\ y :: bool)$
$= (\Box\ (\lambda\ x\ .\ inpt\ r\ (x\ 0)))$

lemma *prec-pre-sts-top*[*simp*]: *prec-pre-sts init* $\top$ $r = \top$

lemma *prec-pre-sts-bot*[*simp*]: *init a* $\Longrightarrow$ *prec-pre-sts init* $\bot$ $r = \bot$

lemma *rel-pre-sts-simp*: *rel-pre-sts init r x y* $= (\exists\ u\ .\ init\ (u\ 0) \wedge (\forall\ i\ .\ r\ (u\ i,\ x\ i)\ (u\ (Suc\ i),\ y\ i)))$

lemma *LocalSystem-simp*: *LocalSystem init p r* $= \{.prec\text{-}pre\text{-}sts\ init\ p\ r.\}\ o\ [:rel\text{-}pre\text{-}sts\ init\ r:]$

definition *local-init init S* $=$ *INFIMUM init S*

definition *zip-set A B* $= \{u\ .\ ((fst\ o\ u) \in A) \wedge ((snd\ o\ u) \in B)\}$

definition *nzip*:: $('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow ('a \times 'b)$ **infixl** $||$ *65*) **where** $(xs\ ||\ ys)\ i = (xs\ i,\ ys\ i)$

lemma *nzip-def-abs*: $(a\ ||\ b) = (\lambda i.\ (a\ i,\ b\ i))$

lemma *nzip-split*: $(fst\ o\ u)\ ||\ (snd\ o\ u) = u$

lemma [*simp*]: $fst \circ x\ ||\ y = x$

lemma [*simp*]: $snd \circ x\ ||\ y = y$

lemma [*simp*]: $x \in A \Longrightarrow y \in B \Longrightarrow (x\ ||\ y) \in$ *zip-set A B*

lemma *local-demonic-init*: *local-init init* $(\lambda\ u\ .\ \{.\ x\ .\ p\ u\ x.\}\ o\ [:x \rightsquigarrow y\ .\ r\ u\ x\ y\ :]) =$
$[:z \rightsquigarrow u,\ x\ .\ u \in init \wedge z = x:]\ o\ \{.u,\ x\ .\ p\ u\ x.\}\ o\ [:u,\ x \rightsquigarrow y\ .\ r\ u\ x\ y\ :]$

lemma *local-init-comp*: $u' \in init' \Longrightarrow (\forall\ u.\ sconjunctive\ (S\ u)) \Longrightarrow$ *(local-init init S) o (local-init init' S')*
$=$ *local-init (zip-set init init') (*$\lambda\ u\ .\ (S\ (fst\ o\ u))\ o\ (S'\ (snd\ o\ u))$)

definition *rel-comp-sts r r'* $= (\lambda\ ((u,v),x)\ ((u',v'),\ z)\ .\ (\exists\ y\ .\ r\ (u,x)\ (u',y) \wedge r'\ (v,y)\ (v',z)))$
definition *prec-comp-sts p r p'* $= (\lambda\ ((u,v),x)\ .\ p\ (u,x) \wedge (\forall\ y\ u'\ .\ r\ (u,\ x)\ (u',y) \longrightarrow p'\ (v,y)))$

definition *sts-comp S S'* $= [-(u,v),x \rightsquigarrow (u,x),v\ -]\ o\ (S\ **\ Skip)\ o\ [-(u,y),v \rightsquigarrow (v,y),u-]\ o\ (S'\ **\ Skip)\ o\ [-(v,z),u \rightsquigarrow (u,v),z-]$

lemma *sts-comp-prec-rel*: *sts-comp* $(\{.p.\}\ o\ [:r:])\ (\{.p'.\}\ o\ [:r':]) = \{.prec\text{-}comp\text{-}sts\ p\ r\ p'.\}\ o\ [:rel\text{-}comp\text{-}sts\ r\ r':]$

We show next that the composition of two SymSystem S and S' is not equal to the SymSystem of the compostion of local transitions of S and S'

definition *initS u = True*
definition *precS = (λ (u, x) . True)*
definition *relS = (λ (u::nat, x::nat) (u'::nat, y::nat) . u = 0 ∧ u' = 1)*

definition *initS' v = True*
definition *precS' = (λ (u, x) . False)*
definition *relS' = (λ (v::nat, x) (v'::nat, y::nat) . True)*

definition *symbS = LocalSystem initS precS relS*
definition *symbS' = LocalSystem initS' precS' relS'*
definition *symbS''= LocalSystem (prod-pred initS initS') (prec-comp-sts precS relS precS') (rel-comp-sts relS relS')*

lemma [*simp*]: *symbS = Magic*

lemma [*simp*]: *symbS''= Fail*

theorem *symbS o symbS' ≠ symbS''*

lemma *rel-pre-sts-comp*: *rel-pre-sts init r OO rel-pre-sts init' r' = rel-pre-sts (prod-pred init init') (rel-comp-sts r r')*

theorem *LocalSystem-comp*: *init' a ⟹ LocalSystem init p r o LocalSystem init' p' r' =*
{.x.(∀ u. *init (u 0) ⟶ (∀ y n. (∀ i<n. r (u i, x i) (u (Suc i), y i)) ⟶ p (u n, x n))) ∧*
(∀ y. (∃ u. init (u 0) ∧ (∀ i. r (u i, x i) (u (Suc i), y i))) ⟶ (∀ u. init' (u 0) ⟶ (∀ ya n. (∀ i<n. r' (u i, y i) (u (Suc i), ya i)) ⟶ p' (u n, y n)))).} ∘
[: *rel-pre-sts init r OO rel-pre-sts init' r'* :]

lemma *sts-comp-prec-aux-a*: *p' ≤ inpt r' ⟹*
(⋀ v y n .v 0 = b ⟹ (∀ i<n. rel-comp-sts r r' ((u i, v i), x i) ((u (Suc i), v (Suc i)), y i)) ⟹ prec-comp-sts p r p' ((u n, v n), x n)) ⟹
∀ i < n. r (u i, x i) (u (Suc i), y i) ⟹ p (u n, x n) ∧ (∃ z v . v 0 = b ∧ (∀ i < n . r' (v i, y i) (v (Suc i), z i) ∧ p' (v i, y i)))

lemma *sts-comp-prec-b*: *p' ≤ inpt r' ⟹ init' b ⟹ prec-pre-sts (prod-pred init init') (prec-comp-sts p r p') (rel-comp-sts r r') x ⟹*
(prec-pre-sts init p r x ∧ (∀ y. rel-pre-sts init r x y ⟶ prec-pre-sts init' p' r' y))

primrec *u-y* :: *('a × 'b ⇒ 'a × 'c ⇒ bool) ⇒ 'a ⇒ (nat ⇒ 'b) ⇒ nat ⇒ 'a ×'c* **where**
u-y r a x 0 = (SOME (u,y) . r (a, x 0) (u, y)) |
u-y r a x (Suc n) = (SOME (u, y) . r (fst (u-y r a x n), x (Suc n)) (u, y))

definition *uu r a x i = (case i of 0 ⇒ a | Suc n ⇒ fst (u-y r a x n))*
definition *yy r a x = snd o (u-y r a x)*

lemma *sts-exists-aux*: *p ≤ inpt r ⟹ prec-pre-sts init p r x ⟹*
init a ⟹ (∀ i ≤ n . r (uu r a x i, x i) (uu r a x (Suc i), yy r a x i))

lemma *sts-exists*: *p ≤ inpt r ⟹ prec-pre-sts init p r x ⟹ init a ⟹ r (uu r a x n, x n) (uu r a x (Suc n), yy r a x n)*

lemma *sts-prec*: $p \leq inpt\ r \Longrightarrow$ *prec-pre-sts init p r x* $\Longrightarrow$ *init a* $\Longrightarrow$ $p\ (uu\ r\ a\ x\ n,\ x\ n)$

lemma *sts-exists-prec*: $p \leq inpt\ r \Longrightarrow$ *prec-pre-sts init p r x* $\Longrightarrow$ *init a* $\Longrightarrow$ $p\ (uu\ r\ a\ x\ n,\ x\ n) \wedge r\ (uu\ r\ a\ x\ n,\ x\ n)\ (uu\ r\ a\ x\ (Suc\ n),\ yy\ r\ a\ x\ n)$

lemma *sts-comp-prec-a*: $p \leq inpt\ r \Longrightarrow$ *prec-pre-sts init p r x* $\Longrightarrow (\bigwedge y.$ *rel-pre-sts init r x y* $\Longrightarrow$ *prec-pre-sts init' p' r' y)*
$\Longrightarrow$ *prec-pre-sts* (*prod-pred init init'*) (*prec-comp-sts p r p'*) (*rel-comp-sts r r'*) *x*

lemma *prec-pre-sts-comp*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow init'\ b \Longrightarrow$
(*prec-pre-sts init p r x* $\wedge$ ($\forall y.$ *rel-pre-sts init r x y* $\longrightarrow$ *prec-pre-sts init' p' r' y*))
= *prec-pre-sts* (*prod-pred init init'*) (*prec-comp-sts p r p'*) (*rel-comp-sts r r'*) *x*

lemma *sts-comp*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow init'\ b \Longrightarrow$
LocalSystem init p r o LocalSystem init' p' r' = *LocalSystem* (*prod-pred init init'*) (*prec-comp-sts p r p'*) (*rel-comp-sts r r'*)

4.2 Parallel Composition of STSs

definition *rel-prod-sts r r'* = (λ ((*u*,*v*), (*x*, *y*)) ((*u'*, *v'*), (*x'*, *y'*)) . *r* (*u*,*x*) (*u'*,*x'*) $\wedge$ *r'* (*v*, *y*) (*v'*, *y'*))
definition *prec-prod-sts p p'* = (λ ((*u*,*v*), (*x*, *y*)) . *p* (*u*,*x*) $\wedge$ *p'* (*v*,*y*))

lemma (*prec-prod-sts* (*inpt r*) (*inpt r'*)) $\leq$ *inpt* (*rel-prod-sts r r'*)

lemma (*prec-prod-sts* (*inpt r*) (*inpt r'*)) = *inpt* (*rel-prod-sts r r'*)

definition *distrib-state* = [:(*u*,*v*), (*x*, *y*) $\rightsquigarrow$ (*u'*, *x'*), (*v'*, *y'*). *u'*=*u* $\wedge$ *v'*=*v* $\wedge$ *x'*=*x* $\wedge$ *y'*=*y*:]
definition *merge-state* = [:(*u*, *x*), (*v*, *y*) $\rightsquigarrow$ (*u'*, *v'*), (*x'*, *y'*). *u'*=*u* $\wedge$ *v'*=*v* $\wedge$ *x'*=*x* $\wedge$ *y'*=*y*:]

lemma *distrib-state o merge-state* = *Skip*

lemma *merge-state o distrib-state* = *Skip*

definition *prod-sts S S'* = (*distrib-state o* (*S* $**$ *S'*) *o merge-state*)

lemma *prod-sts*: *prod-sts* ({.*p*.} *o* [:*r*:]) ({.*p'*.}*o*[:*r'*:]) = {.*prec-prod-sts p p'*.} *o* [:*rel-prod-sts r r'*:]

lemma *update-demonic-update*: [$-f-$] *o* [:*r*:] *o* [$-g-$] = [:*x* $\rightsquigarrow$ *y* . $\exists$ *z* . *r* (*f x*) *z* $\wedge$ *y* = *g z*:]

lemma *sts-prod-prec*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow init\ a \Longrightarrow init'\ b \Longrightarrow$
prec-pre-sts (*prod-pred init init'*) (*prec-prod-sts p p'*) (*rel-prod-sts r r'*) (*x* || *y*)
= (*prec-pre-sts init p r x* $\wedge$ *prec-pre-sts init' p' r' y*)

lemma *sts-prod-rel*: (λ *x y* . $\exists$ *z*. *rel-pre-sts* (*prod-pred init init'*) (*rel-prod-sts r r'*) (*case x of* (*x*, *xa*) $\Rightarrow$ *x* || *xa*) *z* $\wedge$ *y* = (*fst* $\circ$ *z*, *snd* $\circ$ *z*))
= (λ (*x*, *y*) (*u*, *v*) . *rel-pre-sts init r x u* $\wedge$ *rel-pre-sts init' r' y v*)

theorem *sts-prod*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow init\ a \Longrightarrow init'\ b \Longrightarrow$
LocalSystem init p r $**$ *LocalSystem init' p' r'* =
[$-x$, *x'* $\rightsquigarrow$ *x* || *x'*$-$] *o LocalSystem* (*prod-pred init init'*) (*prec-prod-sts p p'*) (*rel-prod-sts r r'*) *o* [$-y$ $\rightsquigarrow$ *fst o y*, *snd o y*$-$]

4.3 Example: COUNTER

In this section we introduce an example counter that counts how many times the input variable x is true. The input is a sequence of boolen values and the output is a sequence of natural numbers. The output at some moment in time is the number of true values seen so far in the input.

We defined the system counter in two different ways and we show that the two definitions are equivalent. The first definition takes the entire input sequence and it computes the corresponding output sequence. We introduce the second version of the counter as a reactive system based on a symbolic transition system. We use a local variable to record the number of true values seen so far, and initially the local variable is zero. At every step we increase the local variable if the input is true. The output of the system at every step is equal to the local variable.

primrec *count* :: *bool trace* ⇒ *nat trace* **where**
count x 0 = (*if x 0 then 1 else 0*) |
count x (*Suc n*) = (*if x* (*Suc n*) *then count x n* + *1 else count x n*)

definition *Counter-global n* = {.*x* . (∀ *k* . *count x k* ≤ *n*).} *o* [:*x* ⇝ *y* . *y* = *count x*:]

definition *prec-count M* = (λ (*u*, *x*) . *u* ≤ *M*)
definition *rel-count* = (λ (*u*,*x*) (*u'*, *y*) . (*x* ⟶ *u'* = *Suc u*) ∧ (¬ *x* ⟶ *u'* = *u*) ∧ *y* = *u'*)

lemma *counter-a-aux*: *u 0* = *0* ⟹ ∀ *i* < *n*. (*x i* ⟶ *u* (*Suc i*) = *Suc* (*u i*)) ∧ (¬ *x i* ⟶ *u* (*Suc i*) = *u i*) ⟹ (∀ *i* < *n* . *count x i* = *u* (*Suc i*))

lemma *counter-b-aux*: *u 0* = *0* ⟹ ∀ *n*. (*xa n* ⟶ *u* (*Suc n*) = *Suc* (*u n*)) ∧ (¬ *xa n* ⟶ *u* (*Suc n*) = *u n*) ∧ *xb n* = *u* (*Suc n*)
⟹ *count xa n* = *u* (*Suc n*)

definition *COUNTER M* = *LocalSystem* (λ *a* . *a* = *0*) (*prec-count M*) *rel-count*

lemma *COUNTER* = *Counter-global*

4.4 Example: LIVE

The last example of this formalization introduces a system which does some local computation, and ensures some global liveness property. We show that this example is the fusion of a symbolic transition system and a demonic choice which ensures the liveness property of the output sequence. We also show that asumming some liveness property for the input, we can refine the example into an executable system that does not ensure the liveness property of the output on its own, but relies on the liveness of the input.

definition *rel-ex* = (λ (*u*, *x*) (*u'*, *y*) . ((*x* ∧ *u'* = *u* + (*1*::*int*)) ∨ (¬ *x* ∧ *u'* = *u* − *1*) ∨ *u'* = *0*) ∧ (*y* = (*u'* = *0*)))
definition *prec-ex* = (λ (*u*, *x*) . −*1* ≤ *u* ∧ *u* ≤ *3*)

definition *LIVE* = {. *prec-pre-sts* (λ *a* . *a* = *0*) *prec-ex rel-ex*.}
o [:*x* ⇝ *y* . ∃ *u* . *u* (*0*::*nat*) = *0* ∧ (□(λ *u x y* . *rel-ex* (*u* (*0*::*nat*), *x* (*0*::*nat*)) (*u 1*, *y* (*0*::*nat*)))) *u x y* ∧ (□ (◇ (λ *y* . *y 0*))) *y* :]

thm *fusion-spec-local-a*

lemma *LIVE-fusion*: $LIVE = (LocalSystem\ (\lambda\ a\ .\ a = 0)\ prec\text{-}ex\ rel\text{-}ex) \parallel [:x \rightsquigarrow y\ .\ (\Box\ (\Diamond\ (\lambda\ y\ .\ y\ 0))))\ y:]$

definition *preca-ex* $x = (x\ 1 = (\neg x\ (0::nat)))$

lemma *monotonic-SymSystem*[*simp*]: *mono* $(LocalSystem\ init\ p\ r)$

lemma *event-ex-aux-a*: $a\ 0 = (0::int) \Longrightarrow \forall n.\ xa\ (Suc\ n) = (\neg\ xa\ n) \Longrightarrow$
$\forall n.\ (xa\ n \wedge a\ (Suc\ n) = a\ n + 1 \vee \neg\ xa\ n \wedge a\ (Suc\ n) = a\ n - 1 \vee a\ (Suc\ n) = 0) \Longrightarrow$
$(a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg\ xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq 1$

lemma *event-ex-aux*: $a\ 0 = (0::int) \Longrightarrow \forall n.\ xa\ (Suc\ n) = (\neg\ xa\ n) \Longrightarrow$
$\forall n.\ (xa\ n \wedge a\ (Suc\ n) = a\ n + 1 \vee \neg\ xa\ n \wedge a\ (Suc\ n) = a\ n - 1 \vee a\ (Suc\ n) = 0) \Longrightarrow$
$(\forall\ n\ .\ (a\ n = -1 \longrightarrow xa\ n) \wedge (a\ n = 1 \longrightarrow \neg\ xa\ n) \wedge -1 \leq a\ n \wedge a\ n \leq 1)$

thm *fusion-local-refinement*

lemma $\{.\Box\ preca\text{-}ex.\}\ o\ LIVE \leq LocalSystem\ (\lambda\ a\ .\ a = (0::int))\ prec\text{-}ex\ rel\text{-}ex$
end

4.5 Iterate Operators

theory *IterateOperators* **imports** *../RefinementReactive/RefinementReactive*
begin

definition *append-inf* :: $'a\ list \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a)$ (**infixr** .. 65) **where**
$(xs..s)\ i = (if\ i < length\ xs\ then\ xs\ !\ i\ else\ s\ (i - (length\ xs)))$

lemma [*simp*]: $[x\ 0]\ ..\ x[Suc\ 0..] = x$

lemma [*simp*]: $([a]\ ..\ x)[Suc\ 0\ ..] = x$

lemma [*simp*]: $([a]\ ..\ x)\ 0 = a$

definition *SkipNext* $S = [:\ x \rightsquigarrow a,\ b\ .\ a = x \wedge b = x[Suc\ 0..]\ :]\ o\ (Prod\ Skip\ S)\ o\ [:\ a,\ b \rightsquigarrow x\ .\ x = cat\ (Suc\ 0)\ a\ b\ :]$

definition *Next* $S = [:x \rightsquigarrow y\ .\ y = x[Suc\ 0..]:]\ o\ S\ o\ [:y \rightsquigarrow x\ .\ y = x[Suc\ 0..]:]$

definition *NextAngelic* $S = \{:\ x \rightsquigarrow y\ .\ y = x[Suc\ 0..]:\}\ o\ S\ o\ \{:y \rightsquigarrow x\ .\ y = x[Suc\ 0..]\ :\}$

definition *SkipTop* $n = [:eqtop\ n:]$

lemma *SkipNext-Next*: $SkipNext\ S = Next\ S \parallel SkipTop\ (Suc\ 0)$

lemma [*simp*]: $SkipTop\ 0 = Havoc$

lemma *proj-skip* [*simp*]: $[:\ y \rightsquigarrow x.\ y = x[Suc\ 0\ ..]\ :] \circ [:\ x \rightsquigarrow y.\ y = x[Suc\ 0\ ..]\ :] = Skip$

lemma *Next-comp*: $Next\ (S\ o\ T) = Next\ S\ o\ Next\ T$

lemma *transp-ref-comp*: $transp\ r \Longrightarrow [:r:] \leq [:r:]\ o\ [:r:]$

lemma *fusion-comp-demonic*: $transp\ r \Longrightarrow (S\ o\ T) \parallel [:r:] \leq (S \parallel [:r:])\ o\ (T \parallel [:r:])$

lemma *fusion-comp-eqtop*: $(S\ o\ T) \parallel [:eqtop\ n:] \leq (S \parallel [:eqtop\ n:])\ o\ (T \parallel [:\ eqtop\ n:])$

lemma *SkipNext-comp-a*[*simp*]: $SkipNext\ (S\ o\ T) \leq (SkipNext\ S)\ o\ (SkipNext\ T)$

definition $auxfun\ p'\ T\ x\ xa = (SUPREMUM\ \{b.\ p'\ b\}\ (\lambda\ b\ .\ (Sup\ \{p'\ .\ (\exists\ p\ .\ (\forall\ a\ b.\ p\ a \wedge p'\ b \longrightarrow x\ (cat\ (Suc\ 0)\ a\ b)) \wedge p\ xa \wedge T\ p'\ b)\})))$

lemma *SkipNext-comp-b*[*simp*]: $mono\ S \Longrightarrow mono\ T \Longrightarrow SkipNext\ (S\ o\ T) \geq (SkipNext\ S)\ o\ (SkipNext\ T)$

lemma *SkipNext-comp*: $mono\ S \Longrightarrow mono\ T \Longrightarrow SkipNext\ (S\ o\ T) = (SkipNext\ S)\ o\ (SkipNext\ T)$

lemma *Next-fusion*: $Next\ (S \parallel T) = (Next\ S) \parallel (Next\ T)$

lemma *fusion-SkipTop-idemp* [*simp*]: $SkipTop\ n \parallel SkipTop\ n = SkipTop\ n$

lemma *SkipNext-fusion*: $SkipNext\ (S \parallel T) = (SkipNext\ S) \parallel (SkipNext\ T)$

lemma *SkipNext-SkipTop*: $SkipNext\ (SkipTop\ n) = SkipTop\ (Suc\ n)$

lemma *SkipTop-SkipNext*: $SkipTop\ n = (SkipNext\ \hat{}\hat{}\ n)\ Havoc$

lemma *SkipNext-power*: $(SkipNext\ \hat{}\hat{}\ (Suc\ n))\ S = (Next\ \hat{}\hat{}\ (Suc\ n))\ S \parallel SkipTop\ (Suc\ n)$

lemma *Next-demonic*: $Next\ [:r:] = [:\bigodot\ r:]$

lemma *SkipNext-demonic*: $SkipNext\ \{.p.\} = \{.\bigodot\ p.\}$

lemma *NextAngelic-angelic*: $NextAngelic\ (\{:r::(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool:\}) = \{:\bigodot\ r:\}$

lemma *Next-assert-demonic*: $Next\ (\{.p.\}\ o\ [:r:]) = \{.\bigodot\ p.\}\ o\ [:\bigodot\ r:]$

lemma *Next-angelic-demonic*: $Next\ (\{:r:\}\ o\ [:r':]) = \{:\bigodot\ r:\}\ o\ [:\bigodot\ r':]$

lemma *eqtop-Suc-zero*: $eqtop\ (Suc\ 0) = (\lambda\ x\ y\ .\ x\ 0 = y\ 0)$

definition $idnext\ r = \bigodot\ r \sqcap eqtop\ (Suc\ 0)$

lemma *SkipNext-assert-demonic*: $SkipNext\ (\{.p.\}\ o\ [:r:]) = \{.\ \bigodot\ p\ .\} \circ [:\ idnext\ r\ :]$

lemma *Next-assert-demonic2*: $Next\ (\lambda\ q\ .\ \{.p.\}\ ([:r:]\ q)) = \{.\bigodot\ p.\}\ o\ [:\bigodot\ r:]$

lemma *Iterate-Next-assert-demonic*: $(Next\hat{}\hat{}n)\ (\{.p.\}\ o\ [:r:]) = \{.(next\hat{}\hat{}n)p.\}\ o\ [:(next\hat{}\hat{}n)\ r:]$

lemma *power-SkipNext-assert-demonic*: $(SkipNext\hat{}\hat{}n)\ (\{.p.\}\ o\ [:r:]) = \{.(next\hat{}\hat{}n)p.\}\ o\ [:(idnext\hat{}\hat{}n)\ r:]$

lemma *Iterate-Next-demonic*: $(Next\ \hat{}\hat{}\ n)\ [:r:] = [:(next\ \hat{}\hat{}\ n)\ r:]$

definition $Always\ S = Fusion\ (\lambda\ n\ .\ (Next\ \hat{}\hat{}\ n)\ S)$

lemma *Always-demonic*: $Always\ [:r:] = [:\Box\ r:]$

lemma *Always-assert-demonic*: $Always\ (\{.p.\}\ o\ [:r:]) = \{.\Box\ p.\}\ o\ [:\Box\ r:]$

lemma *SkipNext-simp*: *SkipNext S Q x* =
(∃ *p p'*. (∀ *a b*. *p a* ∧ *p' b* ⟶ *Q* (*cat* (*Suc 0*) *a b*)) ∧ *p x* ∧ *S p'* (*x*[*Suc 0*..]))

type-synonym ('*a*, '*b*) *trans* = ('*b* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*)

primrec *Iterate* :: (('*a*, '*a*) *trans* ⇒ ('*a*, '*a*) *trans*) ⇒ ('*a*, '*a*) *trans* ⇒ *nat* ⇒ ('*a*, '*a*) *trans* **where**
Iterate F S 0 = *Skip* |
Iterate F S (*Suc n*) = (*Iterate F S n*) *o* ((*F* ^^ *n*) *S*)

definition *Mask n S* = *S o* (*SkipTop n*)

definition *IterateNextMask S n* = *Mask n* (*Iterate Next S n*)

lemma *IterateNextMask-simp*: *IterateNextMask S* = (λ *n* . *Mask n* (*Iterate Next S n*))

definition *IterateSkipNextMask S n* = *Mask n* (*Iterate SkipNext S n*)

lemma *IterateSkipNextMask-simp*: *IterateSkipNextMask S* = (λ *n* . *Mask n* (*Iterate SkipNext S n*))

definition *IterateOmegaNextMask S* = *Fusion* (*IterateNextMask S*)

definition *IterateOmegaSkipNextMask S* = *Fusion* (*IterateSkipNextMask S*)

definition *AddUnitDelay S* = ([:*u*, *x*, *y* ⇝ *a*, *b* . *a* = *u* (*0*::*nat*) ∧ *b* = *x* (*0*::*nat*):] *o S o* [:*c*, *d* ⇝ *u'*, *x'*, *y'* . *u'* (*Suc 0*) = *c* ∧ *y'* (*0*::*nat*) = *d*:])
‖ [:*u*, *x*, (*y*::*nat* ⇒ '*a*) ⇝ *u'*, *x'*, (*y'*::*nat* ⇒'*a*) . *u'* (*0*::*nat*) = *u* (*0*::*nat*) ∧ *x'* = *x*:]

lemma *AddUnitDelay-spec*: *AddUnitDelay* ({.*u*, *x* . *p u x*.} *o* [:*u*, *x* ⇝ *u'*, *y* . *r u u' x y*:]) =
{.*u*, *x*, *y* . *p* (*u 0*) (*x 0*).} *o* [:*u*, *x*, *y* ⇝ *u'*, *x'*, *y'* . *r* (*u 0*) (*u'* (*Suc 0*)) (*x 0*) (*y' 0*) ∧ *x* = *x'* ∧ *u 0* = *u' 0*:]
(**is** *?L* = *?R*)

definition *DelayFeedback init S* = [:*x* ⇝ *u*, *x'*,*y* . *init* (*u* (*0*::*nat*)) ∧ *x* = *x'*:]
o IterateOmegaSkipNextMask (*AddUnitDelay S*) *o* [:*u*, *x*, *y* ⇝ *y'* . *y* = *y'*:]

lemma *SkipNext-refinement*: *S* ≤ *T* ⟹ *SkipNext S* ≤ *SkipNext T*

lemma *SkipNext-pow-refinement*: *S* ≤ *T* ⟹ (*SkipNext* ^^ *n*) *S* ≤ (*SkipNext* ^^ *n*) *T*

lemma *Mask-refinement*: *S* ≤ *T* ⟹ *Mask i S* ≤ *Mask i T*

lemma *mono-SkipNext*[*simp*]: *mono* (*SkipNext S*)

lemma *mono-SkipNext-pow* [*simp*]: *mono S* ⟹ *mono* ((*SkipNext* ^^ *n*) *S*)

lemma *mono-Iterate-SkipNext*[*simp*]: *mono S* ⟹ *mono* (*Iterate SkipNext S n*)

lemma *Iterate-SkipNext-refinement*: ⋀ *S T* . *mono S* ⟹ *S* ≤ *T* ⟹ *Iterate SkipNext S n* ≤ *Iterate SkipNext T n*

lemma *IterateSkipNextMask-refinemnt*: *mono S* ⟹ *S* ≤ *T* ⟹ *IterateSkipNextMask S i* ≤ *IterateSkipNextMask T i*

lemma *IterateOmegaSkipNextMask-refinement*: $mono\ S \Longrightarrow S \leq T \Longrightarrow IterateOmegaSkipNextMask\ S \leq IterateOmegaSkipNextMask\ T$

lemma *AddUnitDelay-refinement*: $S \leq T \Longrightarrow AddUnitDelay\ S \leq AddUnitDelay\ T$

lemma *mono-IterateOmegaSkipNextMask*: $mono\ (IterateOmegaSkipNextMask\ S)$

lemma *mono-AddUnitDelay*: $mono\ (AddUnitDelay\ S)$

theorem *DelayFeedback-refinement*: $init' \leq init \Longrightarrow S \leq T \Longrightarrow DelayFeedback\ init\ S \leq DelayFeedback\ init'\ T$

lemma [*simp*]: $mono\ (SkipTop\ n)$

lemma [*simp*]: $SkipNext\ Skip = Skip$

lemma *Iterate-SkipNextA*: $mono\ S \Longrightarrow S \circ (SkipNext\ (Iterate\ SkipNext\ S\ n)) = Iterate\ SkipNext\ S\ (Suc\ n)$

lemma *skiptop-simp*: $SkipTop\ n\ p = (\lambda\ x\ .\ \forall\ \ y\ .\ eqtop\ n\ x\ y \longrightarrow p\ y)$

definition $HavocTop\ n = [:x \rightsquigarrow y\ .\ x[n..] = y[n..]:]$

lemma *HavocTop-Next*: $HavocTop\ (Suc\ n) = Next\ (HavocTop\ n)$

lemma [*simp*]: $HavocTop\ 0 = Skip$

lemma $HavocTop\ n = (Next\ \hat{}\hat{}\ n)\ Skip$

lemma *Next-NextSkip-aux*: $[:\ \lambda y\ x.\ \forall\, xa.\ y\ xa = x\ (Suc\ xa)\ :]\ (\lambda a.\ \forall\, b.\ a[Suc\ 0\ ..] = b[Suc\ 0\ ..] \longrightarrow x\ b) = [:\ \lambda y\ x.\ \forall\, xa.\ y\ xa = x\ (Suc\ xa)\ :]\ x$

lemma *demonic-apply-pred*: $[:\ \lambda x\ y.\ r\ x\ y\ :]\ p\ \ = (\lambda x.\ \forall\ \ y\ .\ r\ x\ y \longrightarrow p\ y)$

lemma *Next-SkipNext-HavocTop*: $mono\ S \Longrightarrow Next\ S = SkipNext\ S\ o\ HavocTop\ (Suc\ 0)$

lemma *HavocTop-Next-power*: $HavocTop\ n \circ Next\ ((Next\ \hat{}\hat{}\ n)\ S) = \ \ Next\ ((Next\ \hat{}\hat{}\ n)\ S)$

lemma *Next-SkipNext*: $mono\ S \Longrightarrow (Next\ \hat{}\hat{}\ n)\ S = (SkipNext\ \hat{}\hat{}\ n)\ S \circ HavocTop\ n$ (**is** $?Q \Longrightarrow ?A\ n = ?B\ n$)

lemma *Iterate-Next-SkipNext-aux*: $mono\ S \Longrightarrow HavocTop\ n \circ (Next\ \hat{}\hat{}\ (Suc\ n))\ S = (SkipNext\ \hat{}\hat{}\ (Suc\ n))\ S \circ HavocTop\ (Suc\ n)$ (**is** $?P \Longrightarrow ?A = ?B$)

lemma *Iterate-Next-SkipNext-Suc*: $mono\ S \Longrightarrow Iterate\ Next\ S\ (Suc\ n) = (Iterate\ SkipNext\ S\ (Suc\ n))\ o\ (HavocTop\ n)$ (**is** $?P \Longrightarrow ?A\ n = ?B\ n$)

lemma *Iterate-Next-SkipNext*: $mono\ S \Longrightarrow Iterate\ Next\ S\ n = (Iterate\ SkipNext\ S\ n)\ o\ (HavocTop\ (n - 1))$

lemma $HavocTop\ n \leq Skip$

lemma *mono-Iterate-NextSkip*: *mono S* $\Longrightarrow$ *mono* (*Iterate SkipNext S n*)

lemma (*Havoc* (*X*::'*a*::*complete-lattice*) $\neq \bot$) = (*X* = $\top$)

type-synonym ('*a*, '*b*) *rel* = ('*a* $\Rightarrow$ '*b* $\Rightarrow$ *bool*)

primrec *IterateRel* :: (('*a*, '*a*) *rel* $\Rightarrow$ ('*a*, '*a*) *rel*) $\Rightarrow$ ('*a*, '*a*) *rel* $\Rightarrow$ *nat* $\Rightarrow$ ('*a*, '*a*) *rel* **where**
IterateRel F r 0 = (λ *a b* . *a* = *b*) |
IterateRel F r (*Suc n*) = *IterateRel F r n OO* ((*F* ^^ *n*) *r*)

lemma *IterateRel-init*: ($\forall$ *r r'* . *F* (*r OO r'*) = *F r OO F r'*) $\Longrightarrow$ *F* (*op* =) = (*op* =) $\Longrightarrow$ *IterateRel F r* (*Suc n*) = *r OO F* (*IterateRel F r n*) (**is** *?P* $\Longrightarrow$ *?Q* $\Longrightarrow$ *?R n*)

lemma [*simp*]: *idnext* (*op* =) = (*op* =)

lemma [*simp*]: *idnext* (*r OO r'*) = (*idnext r*) *OO idnext r'*

lemma *IterateRel-idnext-init*: *IterateRel idnext r* (*Suc n*) = *r OO idnext* (*IterateRel idnext r n*)

lemma [*simp*]: ($\bigwedge$ (*p*::'*a*$\Rightarrow$*bool*) (*r*::'*a*$\Rightarrow$'*b*$\Rightarrow$*bool*) . *F* ({.*p*.} *o* [:*r*:]) = {.*A p*.} *o* [:(*B*::('*a*$\Rightarrow$'*b*$\Rightarrow$*bool*) $\Rightarrow$ ('*a*$\Rightarrow$'*b*$\Rightarrow$*bool*)) *r*:])
$\Longrightarrow$ ((*F* ^^ *n*) ({.*p*.} *o* [:*r*:])) = {.(*A* ^^*n*) *p*.} *o* [:(*B* ^^*n*) *r*:]

lemma *Iterate-id*: *Iterate id S n* = *S* ^^ *n*

lemma *IterateRel-id*: *IterateRel id r n* = (*r* ^^ *n*)

lemma *Iterate-IterateRel*: ($\bigwedge$ *p r* . *F* ({.*p*.} *o* [:*r*:]) = {.*A p*.} *o* [:*B r*:]) $\Longrightarrow$ *Iterate F* ({.*p*.} *o* [:*r*:]) *n*
= {.*x* . ($\forall$ *i* < *n* .($\forall$ *y* . *IterateRel B r i x y* $\longrightarrow$ (*A* ^^*i*) *p y*)).} *o* [:*IterateRel B r n*:]

lemma *IterateRel-app*: $\bigwedge$ *y* . *IterateRel next r n x y* = ($\exists$ *a* . *a 0* = *x* $\wedge$ *a n* = *y* $\wedge$ ($\forall$ *i* < *n*. *r* ((*a i*)[*i*..]) ((*a* (*Suc i*))[*i*..])))

lemma *Iterate-Next-IterateRel*: *Iterate Next* ({.*p*.} *o* [:*r*:]) *n*
= {. *x* . ($\forall$ *k* < *n* . ($\forall$ *y* . *IterateRel next r k x y* $\longrightarrow$ (*next* ^^ *k*) *p y*)).} *o* [:*IterateRel next r n*:]

lemma *IterateOmegaNextMask-spec-aux*: *IterateOmegaNextMask* ({.*p*.} *o* [:*r*:])
= {. *INF x*. (λ*xa*. $\forall$ *k*<*x*. $\forall$ *y*. *IterateRel next r k xa y* $\longrightarrow$ (*next* ^^ *k*) *p y*) .} $\circ$ [: *INF n*. *IterateRel next r n OO eqtop n* :]

lemma *IterateOmegaNextMask-spec*: *IterateOmegaNextMask* ({.*p*.} *o* [:*r*:])
= {. *INF k* . (λ*xa*. $\forall$ *y*. *IterateRel next r k xa y* $\longrightarrow$ (*next* ^^ *k*) *p y*) .} $\circ$ [: *INF n*. *IterateRel next r n OO eqtop n* :]

lemma *power-spec*: ({.*p*.} *o* [:*r*:]) ^^ *n*
= {.*x* . ($\forall$ *i* < *n* .($\forall$ *y* . (*r* ^^ *i*) *x y* $\longrightarrow$ *p y*)).} *o* [:*r* ^^ *n*:]

lemma *Iterate-SkipNext-IterateSkipRel*: *Iterate SkipNext* ({.*p*.} *o* [:*r*:]) *n*
= {. *x* . ($\forall$ *k* < *n* . ($\forall$ *y* . *IterateRel idnext r k x y* $\longrightarrow$ (*next* ^^ *k*) *p y*)).} *o* [:*IterateRel idnext r n*:]

lemma *IterateOmegaSkipNextMask-spec*: *IterateOmegaSkipNextMask* ({.*p*.} *o* [:*r*:])
= {. (λ*x* . $\forall$ *n* . $\forall$ *y*. *IterateRel idnext r n x y* $\longrightarrow$ (*next* ^^ *n*) *p y*) .}
$\circ$ [: *INF n*. *IterateRel idnext r n OO eqtop n* :]

lemma *IterateOmegaSkipNextMask-demonic*: *IterateOmegaSkipNextMask* [:*r*:] = [: *INF n*. *IterateRel idnext r n OO eqtop n* :]

lemma [*simp*]: (*next* ^^ *n*) ⊤ *x*

lemma *power-idnext*: (*idnext* ^^ *n*) *r* = ((*next* ^^ *n*) *r* ⊓ *eqtop n*)

lemma *example-feedback-delay-a*: ∀ *xb*. ∃ *z*. *IterateRel idnext* (λ*x y*. ∀ *xa*. *y xa* = ([*0*] .. *x*) *xa*) *xb x z* ∧ (∀ *i*<*xb*. *z i* = *xa i*) ⟹ *xa n* = *0*

lemma *example-feedback-delay-b*: ∀ *x*. *xa x* = *0* ⟹ ∃ *z*. *IterateRel idnext* (λ*x y*. ∀ *xa*. *y xa* = ([*0*] .. *x*) *xa*) *n x z* ∧ (∀ *i* < *n*. *z i* = *xa i*)

lemma *example-feedback-delay*: *IterateOmegaSkipNextMask* [:*x* ⇝ *y* . *y* = [*0*::*nat*] .. *x*:] = [:*x*⇝ *y* . *y* = (λ *i* . *0*):]

lemma *next-simp*: *next* (*r*::(*nat* ⇒ *'a*) ⇒(*nat* ⇒ *'b*) ⇒*bool*) *x y* = *r* (*x*[*Suc 0*..]) (*y*[*Suc 0*..])

lemma *idnext-simp*: *idnext* (*r*::(*nat* ⇒ *'a*) ⇒(*nat* ⇒ *'a*) ⇒*bool*) *x y* = (*r* (*x*[*Suc 0*..]) (*y*[*Suc 0*..]) ∧ *x 0* = *y 0*)

lemma *idnext-next-eqtop*: ⋀ (*x*::*nat*⇒*'a*) *y* . (*idnext* ^^ *n*) *r x y* = ((*next* ^^ *n*) *r x y* ∧ *eqtop n x y*)

lemma *IrerateRel-IterateSkipRel-aux*: ∀ *x y* . *IterateRel next* (*r*::(*nat* ⇒ *'a*) ⇒(*nat* ⇒ *'a*) ⇒*bool*) *n x y* ⟶ (∃ *z* . *y*[(*n*::*nat*)..] = *z*[*n*..] ∧ *IterateRel idnext r n x z*)

lemma *IrerateRel-IterateSkipRel*: *IterateRel next* (*r*::(*nat* ⇒ *'a*) ⇒(*nat* ⇒ *'a*) ⇒*bool*) *n x y* ⟹ (∃ *z* . *y*[(*n*::*nat*)..] = *z*[*n*..] ∧ *IterateRel idnext r n x z*)

lemma *next-eq*: ∀ *i*<*k*. (∀ *x*. *fst* (*snd* (*ab i*)) (*i* + *x*) = *fst* (*snd* (*ab* (*Suc i*))) (*i* + *x*)) ⟹
i ≤ *k* ⟹ (∀ *j* . *fst* (*snd* (*ab i*)) (*i* + *j*) = *fst* (*snd* (*ab 0*)) (*i* + *j*))

lemma *IterateSkipRel-SymRel-zero*: ⋀ *u' x' y'* . (*IterateRel idnext* (λ(*u*, *x*, *y*) (*u'*, *x'*, *y'*). *r* (*u 0*) (*u'* (*Suc 0*)) (*x 0*) (*y' 0*) ∧ (*x* = *x'*) ∧ (*u 0* = *u' 0*)) *0*) (*u*, *x*, *y*) (*u'*, *x'*, *y'*)
= (*u* = *u'* ∧ *x* = *x'* ∧ *y* = *y'*)

lemma *IterateSkipRel-SymRel-Suc*: ⋀ *u' x' y'* . (*IterateRel idnext* (λ(*u*, *x*, *y*) (*u'*, *x'*, *y'*). *r* (*u 0*) (*u'* (*Suc 0*)) (*x 0*) (*y' 0*) ∧ (*x* = *x'*) ∧ (*u 0* = *u' 0*)) (*Suc n*)) (*u*, *x*, *y*) (*u'*, *x'*, *y'*)
= ((*u' 0* = *u 0*) ∧ (∀ *i* < (*Suc n*) . *r* (*u' i*) (*u'* (*Suc i*)) (*x i*) (*y' i*)) ∧ *x* = *x'*)

lemma *IterateSkipRel-SymRel*: ⋀ *u' x' y'* . (*IterateRel idnext* (λ(*u*, *x*, *y*) (*u'*, *x'*, *y'*). *r* (*u 0*) (*u'* (*Suc 0*)) (*x 0*) (*y' 0*) ∧ (*x* = *x'*) ∧ (*u 0* = *u' 0*)) *n*) (*u*, *x*, *y*) (*u'*, *x'*, *y'*)
= ((*u' 0* = *u 0*) ∧ (∀ *i* < *n* . *r* (*u' i*) (*u'* (*Suc i*)) (*x i*) (*y' i*)) ∧ *x* = *x'* ∧ (*n* = *0* ⟶ (*u* = *u'* ∧ *y* = *y'*)))

lemma *IterateSkipRel-SymRel-eqtop*: (*IterateRel idnext* (λ(*u*, *x*, *y*) (*u'*, *x'*, *y'*). *r* (*u* (*0*::*nat*)) (*u'* (*Suc 0*)) (*x* (*0*::*nat*)) (*y'* (*0*::*nat*)) ∧ (*x* = *x'*) ∧ (*u 0* = *u' 0*)) *n OO* (*eqtop n*)) (*u*, *x*, *y*) (*u'*, *x'*, *y'*)
= (∃ *v* . (*v 0* = *u 0*) ∧ (∀ *i* < *n* . *r* (*v i*) (*v* (*Suc i*)) (*x i*) (*y' i*) ∧ *v i* = *u' i* ∧ (*x i* = *x' i*)))

lemma *INF-IterateSkipRel-SymRel-eqtop*: (*INF n. IterateRel idnext* (λ(*u*, *x*, *y*) (*u′*, *x′*, *y′*). *r* (*u* (*0*::*nat*)) (*u′* (*Suc 0*)) (*x* (*0*::*nat*)) (*y′* (*0*::*nat*)) ∧ *x* = *x′* ∧ *u 0* = *u′ 0*) *n OO eqtop n*) (*u*, *x*, *y*) (*u′*, *x′*, *y′*)
= (*u′ 0* = *u 0* ∧ *x* = *x′* ∧ (□ (λ (*u*, *x*, *y*) . *r* (*u 0*) (*u* (*Suc 0*)) (*x 0*) (*y 0*))) (*u′*, *x*, *y′*))

lemma *INF-IterateSkipRel-SymRel-eqtop-abs*: (*INF n. IterateRel idnext* (λ(*u*, *x*, *y*) (*u′*, *x′*, *y′*). *r* (*u* (*0*::*nat*)) (*u′* (*Suc 0*)) (*x* (*0*::*nat*)) (*y′* (*0*::*nat*)) ∧ *x* = *x′* ∧ *u 0* = *u′ 0*) *n OO eqtop n*)
= (λ (*u*, *x*, *y*) (*u′*, *x′*, *y′*) . (*u′ 0* = *u 0* ∧ *x* = *x′* ∧ (□ (λ (*u*, *x*, *y*) . *r* (*u 0*) (*u* (*Suc 0*)) (*x 0*) (*y 0*))) (*u′*, *x*, *y′*)))

lemma *move-down*: *p* ⟹ *p*

lemma *IterateSkipRel-prec-loc-st*: (λ*x*. ∀ *a*. *init* (*a 0*) ⟶
(∀ *b n aa aaa ba*.
IterateRel idnext (λ(*u*, *x*, *y*) (*u′*, *x′*, *y′*). *r* (*u* (*0*::*nat*)) (*u′* (*Suc 0*)) (*x* (*0*::*nat*)) (*y′* (*0*::*nat*)) ∧ *x* = *x′* ∧ *u 0* = *u′ 0*) *n* (*a*, *x*, *b*) (*aa*, *aaa*, *ba*) ⟶
(*next* ^^ *n*) (λ(*u*, *x*, *y*). *p* (*u 0*) (*x 0*)) (*aa*, *aaa*, *ba*)))
= *prec-pre-sts init* (λ (*u*, *x*) . *p u x*) (λ (*u*,*x*) (*u′*, *y*) . *r u u′ x y*)

theorem *DelayFeedback-SymbolicSystem-aux*: *DelayFeedback init* ({.(*x*, *y*).*p x y*.} ∘ [:(*u*, *x*)⇝(*u′*, *y*).*r u u′ x y*:])
= *LocalSystem init* (λ (*u*, *x*) . *p u x*) (λ (*u*,*x*) (*u′*, *y*) . *r u u′ x y*)

theorem *DelayFeedback-LocalSystem*: *DelayFeedback init* ({.*p*.} ∘ [:*r*:])
= *LocalSystem init p r*

lemma *DelayFeedback-simp*: *DelayFeedback init* ({.*p*.} *o* [:*r*:]) = {.*prec-pre-sts init p r*.} *o* [:*rel-pre-sts init r*:]

lemma *prec-pre-sts-prec-rel*: (⋀ *s s′ x y* . *p* (*s*, *x*) ⟹ *r* (*s*, *x*) (*s′*, *y*) = *r′* (*s*, *x*) (*s′*, *y*)) ⟹ *prec-pre-sts init p r* = *prec-pre-sts init p r′*

theorem *DelayFeedback-a-simp*: *DelayFeedback init* ({.*p*.} *o* [:*r*:]) =
{.*x* .(∀ *u y* . *init* (*u 0*) ⟶ (∀ *n* . (∀ *i* < *n* . *r* (*u i*, *x i*) (*u* (*Suc i*), *y i*)) ⟶ *p* (*u n*, *x n*))).}
o [:*x* ⇝ *y* . (∃ *u* . *init* (*u 0*) ∧ (∀ *i* . *r* (*u i*, *x i*) (*u* (*Suc i*), *y i*))):]

theorem *DelayFeedback-b-simp*: *DelayFeedback init* ([:*r*:])
= [:*rel-pre-sts init r*:]

lemma *DelayFeedback-comp*: *p* ≤ *inpt r* ⟹ *p′* ≤ *inpt r′* ⟹ *init′ b* ⟹
DelayFeedback init ({.*p*.} *o* [:*r*:]) *o DelayFeedback init′* ({.*p′*.} *o* [:*r′*:]) =
DelayFeedback (*prod-pred init init′*) ({.*prec-comp-sts p r p′*.} *o* [:*rel-comp-sts r r′*:])

lemma *DelayFeedback-empty-init*[*simp*]: *DelayFeedback* ⊥ *S′* = ⊤

lemma *assert-bot*: {.⊥::*′a*::*boolean-algebra*.} = *Fail*

lemma *Fail-comp*: *Fail o S = Fail*

lemma *DelayFeedback-Fail*[*simp*]: *init a ⟹ DelayFeedback init* (*Fail*:: (*'a × 'b ⇒ bool*) *⇒* (*'a × 'c ⇒ bool*)) *= Fail*

lemma *prod-empty* [*simp*]: *prod-pred X ⊥ = ⊥*

lemma *sts-serial-comp-empty-init*: *DelayFeedback* (*prod-pred ⊤ ⊥*) (*sts-comp Fail S'*) *≠ DelayFeedback ⊤ Fail o DelayFeedback ⊥ S'*

thm *DelayFeedback-LocalSystem*

theorem *sts-serial-comp*: *implementable S ⟹ implementable S' ⟹ init' b ⟹*
DelayFeedback (*prod-pred init init'*) (*sts-comp S S'*) *= DelayFeedback init S o DelayFeedback init' S'*

theorem *implementableI*: *p ≤ inpt r ⟹ implementable* ({.*p*.} *o* [:*r*:])

lemma *implementable-inpt*[*simp*]: *implementable* ({.*inpt r*.} *o* [:*r*:])

theorem *implementable-DelayFeedback*: *implementable S ⟹ init a ⟹ implementable* (*DelayFeedback init S*)

theorem *LocalSystem-impt-implementable*: *init a ⟹ implementable* (*LocalSystem init* (*inpt r*) *r*)

lemma *prec-pre-sts-inpt*: *init a ⟹ prec-pre-sts init* (*inpt r*) *r ≤ inpt* (*rel-pre-sts init r*)

lemma *comp-middle*: *A o B o C o D = A o* (*B o C*) *o D*

lemma *fun-eq*: (*∀ x. f x = g x*) *= f = g*

lemma [*simp*]: *SkipNext ⊥ = ⊥*

lemma *SkipNext ⊥ = ⊥*

lemma *SkipNext ⊤ ⊥ = ⊤*

lemma *SkipNext ⊤ = ⊤*

4.6 Examples

definition *PREC-ID = ⊤*
definition *REL-ID = (λ* (*u*,*x*) (*u'*, *y*) .(*u = u'*) *∧* (*u = y*))
definition *INIT-ID u = (u = 0)*

lemma *all-eq*: *∀ x. u x = u* (*Suc x*) *⟹ u x = u 0*

lemma *LocalSystem INIT-ID PREC-ID REL-ID =* [:*x ⇝ y . ∀ i . y i = 0*:]

definition *PREC-COUNTER = ⊤*
definition *REL-COUNTER = (λ* (*u*, *x*) (*u'*, *y*) . (*u' = u + 1*) *∧* (*u = y*))
definition *INIT-COUNTER u = (u = 0)*

lemma *add-suc*: *∀ x. u* (*Suc x*) *= Suc* (*u x*) *⟹ u x = x + u 0*

lemma *LocalSystem INIT-COUNTER PREC-COUNTER REL-COUNTER* = [:$x \rightsquigarrow y$. $\forall\ i$. $y\ i = i$:]

definition *PREC-SUM* = $\top$
definition *REL-SUM* = $(\lambda\ (u,\ x)\ (u',\ y)\ .\ (u' = u + x) \wedge (u = y))$
definition *INIT-SUM* $u = (u = 0)$

primrec *Summ* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$ **where**
Summ $x\ 0 = 0$ |
Summ $x\ (Suc\ n) =$ *Summ* $x\ n + x\ n$

lemma *sum-suc*: $\forall\, n.\ u\ (Suc\ n) = u\ n + x\ n \Longrightarrow u\ n =$ *Summ* $x\ n + u\ 0$

lemma *LocalSystem INIT-SUM PREC-SUM REL-SUM* = [:$x \rightsquigarrow y\ .\ y =$ *Summ* x:]

definition *PREC-A* = $\top$
definition *REL-A* = $(\lambda\ (u,\ x)\ (u',\ y)\ .\ (u' = x) \wedge (u = y))$
definition *INIT-A* $u = (u = 0)$

lemma *LocalSystem INIT-A PREC-A REL-A* = [:$x \rightsquigarrow y\ .\ y = [0]\ ..\ x$:]

definition *PREC-SUM-A* = $(\lambda\ (u,\ x)\ .\ u \leq 100)$
definition *REL-SUM-A* = $(\lambda\ (u,\ x)\ (u',\ y)\ .\ (u' = u + x) \wedge (u = y))$
definition *INIT-SUM-A* $u = (u = 0)$

lemma *sum-suc-le*: $\forall\, n < k\ .\ u\ (Suc\ n) = u\ n + x\ n \Longrightarrow u\ k =$ *Summ* $x\ k + u\ 0$

lemma *LocalSystem INIT-SUM-A PREC-SUM-A REL-SUM-A* = $\{.x\ .\ \forall\ i\ .$ *Summ* $x\ i \leq 100.\}$ *o* [:$x \rightsquigarrow y\ .\ y =$ *Summ* x:]

definition *PREC-SUM-B* = $(\lambda\ (u,\ x)\ .\ u \leq 100)$
definition *REL-SUM-B* = $(\lambda\ (u,\ x)\ (u',\ y)\ .\ (u' = u + x \vee u' = x) \wedge (u = y))$
definition *INIT-SUM-B* $u = (u = 0)$

lemma *le-sum-suc*: $\forall\, n < k\ .\ u\ (Suc\ n) = u\ n + x\ n \vee u\ (Suc\ n) = x\ n \Longrightarrow u\ k \leq$ *Summ* $x\ k + u\ 0$

lemma *LocalSystem INIT-SUM-B PREC-SUM-B REL-SUM-B*
$=$ $\{.x\ .\ \forall\ i\ .$ *Summ* $x\ i \leq 100.\}$ *o* [:$x \rightsquigarrow y\ .\ y\ 0 = 0 \wedge (\forall\ i\ .\ y\ (Suc\ i) = y\ i + x\ i \vee y\ (Suc\ i) = x\ i)$:]

lemma *prod-comp-spec*[*simp*]: *pa* $\leq$ *inpt ra* $\Longrightarrow$ *pb* $\leq$ *inpt rb* $\Longrightarrow$ (($\{.pa.\} \circ [:ra:]$) $**$ ($\{.pb.\} \circ [:rb:]$)) $\circ$ (($\{.pc.\} \circ [:rc:]$) $**$ ($\{.pd.\} \circ [:rd:]$))
$=$ ($\{.pa.\}$ *o* $[:ra:]$ *o* $\{.pc.\}$ *o* $[:rc:]$) $**$ ($\{.pb.\}$ *o* $[:rb:]$ *o* $\{.pd.\}$ *o* $[:rd:]$)

lemma *prod-comp-implem*: *implementable* $S \Longrightarrow$ *implementable* $S' \Longrightarrow$ *sconjunctive* $T \Longrightarrow$ *sconjunctive* $T' \Longrightarrow$ (S $**$ S') *o* (T $**$ T') = (S *o* T) $**$ (S' *o* T')

definition *ang-rel* $S\ s\ q = S\ q\ s$
definition *dem-rel* $S\ q\ s' = q\ s'$

lemma *mono-rep*: *mono* $S \Longrightarrow S =$ $\{:$*ang-rel* $S:\}$ *o* $[:$*dem-rel* $S:]$

lemma *mono-repE*: *mono* $(S::('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)) \Longrightarrow \exists\ (r::'b \Rightarrow ('a \Rightarrow bool) \Rightarrow bool)\ (r')$.

$S = \{:r:\}\ o\ [:r':]$

lemma *prod-comp-a*: $(S\ o\ T)\ **\ (S'\ o\ T') \leq (S\ **\ S')\ o\ (T\ **\ T')$

lemma *prod-comp-angelic-demonic*: $(\{:r::'a \Rightarrow 'b \Rightarrow bool:\}\ **\ \{:r'::'c \Rightarrow 'd \Rightarrow bool:\})\ o\ ([:t:]\ **\ [:t':]) = (\{:r:\}\ o\ [:t:])\ **\ (\{:r':\}\ o\ [:t':])$

definition *prod-rel r r'* $= (\lambda\ (x,\ y)\ (u,\ v)\ .\ r\ x\ u \wedge r'\ y\ v)$

lemma *Prod-angelic*: $\{:r:\}\ **\ \{:r':\} = \{:\ prod\text{-}rel\ r\ r'\ :\}$

lemma *Prod-demonic-rel*: $[:r:]\ **\ [:r':] = [:\ prod\text{-}rel\ r\ r'\ :]$

lemma *prod-rel-comp*: *prod-rel r r' OO prod-rel t t'* $=$ *prod-rel* $(r\ OO\ t)\ (r'\ OO\ t')$

lemma *prod-comp-angelic-demonic-demonic*: $((\{:ra:\}\ o\ [:rd:])\ **\ (\{:ra':\}\ o\ [:rd':]))\ o\ ([:r:]\ **\ [:r':]) = (\{:ra:\}\ o\ [:rd:]\ o\ [:r:])\ **\ ((\{:ra':\}\ o\ [:rd':])\ o\ [:r':])$

lemma *prod-comp-demonic*: *mono* $(S::('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)) \Longrightarrow$ *mono* $(S'::('c \Rightarrow bool) \Rightarrow ('d \Rightarrow bool)) \Longrightarrow$
$(S\ **\ S')\ o\ ([:r:]\ **\ [:r':]) = (S\ o\ [:r:])\ **\ (S'\ o\ [:r':])$

theorem *DelayFeedback-prod*: *init a* $\Longrightarrow$ *init' a'* $\Longrightarrow$ *implementable S* $\Longrightarrow$ *implementable S'* $\Longrightarrow$ *DelayFeedback init S ** DelayFeedback init' S'* $=$
$[-\ (x,\ y) \rightsquigarrow x\ ||\ y\ -]\ o$ *DelayFeedback* $(prod\text{-}pred\ init\ init')\ (prod\text{-}sts\ S\ S')\ o\ [-\lambda\ x\ .\ (fst\ o\ x,\ snd\ o\ x)\ -]$

lemma *rel-fun-power*: $((\lambda x\ y.\ y = (f::'a \Rightarrow 'a)\ x)\ \hat{}\hat{}\ n) = (\lambda\ x\ y\ .\ (y = (f\ \hat{}\hat{}\ n)\ x))$

lemma $[simp]$: $[:\bot:] = Magic$

definition *IterateMask S n* $= Mask\ n\ ((S::('a::trace \Rightarrow bool) \Rightarrow ('a \Rightarrow bool))\ \hat{}\hat{}\ n)$

lemma *IterateMask-simp*: *IterateMask S* $= (\lambda\ n.\ Mask\ n\ (S\ \hat{}\hat{}\ n))$

definition *IterateOmega S* $=$ *Fusion* $(IterateMask\ S)$

definition *IterateMaskA S n* $= Mask\ (n - 1)\ ((S::('a::trace \Rightarrow bool) \Rightarrow ('a \Rightarrow bool))\ \hat{}\hat{}\ n)$

lemma *IterateMaskA-simp*: *IterateMaskA S* $= (\lambda\ n.\ Mask\ (n-1)\ (S\ \hat{}\hat{}\ n))$

definition *IterateOmegaA S* $=$ *Fusion* $(IterateMaskA\ S)$

lemma *IterateMaskA S n* $= (S\ \hat{}\hat{}\ n) o\ [:x \rightsquigarrow y\ .\ \forall\ (i::nat) < n - 1\ .\ ((y\ i)::'a) = x\ i:]$

lemma *power-refin*: *mono S* $\Longrightarrow (S::'a::order \Rightarrow 'a) \leq T \Longrightarrow S\ \hat{}\hat{}\ n \leq T\ \hat{}\hat{}\ n$

lemma *IterateMaskA-refin*: *mono S* $\Longrightarrow S \leq T \Longrightarrow$ *IterateMaskA S n* $\leq$ *IterateMaskA T n*

lemma *IterateOmegaA-refin*: $mono\ S \Longrightarrow S \leq T \Longrightarrow IterateOmegaA\ S \leq IterateOmegaA\ T$

lemma *IterateOmega-spec*: $IterateOmega\ (\{.p.\}\ o\ [:r:])$
$= \{.\ (\lambda x\ .\ \forall\ n\ .\ \forall y.\ (r\ \hat{}\hat{}\ n)\ x\ y \longrightarrow p\ y)\ .\}$
$\circ\ [:\ INF\ n.\ (r\ \hat{}\hat{}\ n)\ OO\ eqtop\ n\ :]$

lemma *IterateOmegaA-spec*: $IterateOmegaA\ (\{.p.\}\ o\ [:r:])$
$= \{.\ (\lambda x\ .\ \forall\ n\ y.\ (r\ \hat{}\hat{}\ n)\ x\ y \longrightarrow p\ y)\ .\}$
$\circ\ [:\ INF\ n.\ (r\ \hat{}\hat{}\ n)\ OO\ eqtop\ (n-1)\ :]$

lemma *rel-power-a*: $\bigwedge\ y\ .\ ((r :: 'a \Rightarrow 'a \Rightarrow bool)\ \hat{}\hat{}\ n)\ x\ y \Longrightarrow \exists\ a\ .\ x = a\ 0 \wedge y = a\ n \wedge (\forall\ i < n\ .\ r\ (a\ i)\ (a\ (Suc\ i)))$

lemma *rel-power-b*: $\bigwedge\ y\ .\ \exists\ a\ .\ x = a\ 0 \wedge y = a\ n \wedge (\forall\ i < n\ .\ r\ (a\ i)\ (a\ (Suc\ i))) \Longrightarrow ((r :: 'a \Rightarrow 'a \Rightarrow bool)\ \hat{}\hat{}\ n)\ x\ y$

lemma *rel-power*: $((r :: 'a \Rightarrow 'a \Rightarrow bool)\ \hat{}\hat{}\ n)\ x\ y = (\exists\ a\ .\ x = a\ 0 \wedge y = a\ n \wedge (\forall\ i < n\ .\ r\ (a\ i)\ (a\ (Suc\ i))))$

lemma *IterateOmega-demonic-spec*: $IterateOmega\ [:r:] = [:\ INF\ n.\ r\ \hat{}\hat{}\ n\ OO\ eqtop\ n\ :]$

lemma *IterateOmega-func*: $IterateOmega\ [-\ f\ -] = [:\ x \rightsquigarrow y\ .\ \forall\ n.\ eqtop\ n\ ((f\ \hat{}\hat{}\ n)\ x)\ y\ :]$

lemma *IterateOmega-func-aux-a*: $(\forall\ n.\ eqtop\ n\ ((f\ \hat{}\hat{}\ n)\ x)\ y) = (\forall\ n\ .\ \forall\ i < n\ .\ (f\ \hat{}\hat{}\ n)\ x\ i = y\ i)$

lemma *IterateOmega-func-a*: $IterateOmega\ [-\ f\ -] = [:\ x \rightsquigarrow y\ .\ (\forall\ n\ .\ \forall\ i < n\ .\ (f\ \hat{}\hat{}\ n)\ x\ i = y\ i):]$

definition $apply\ x\ i = ((fst\ (fst\ x)\ i,\ snd\ (fst\ x)\ i),\ snd\ x\ i)$

lemma *IterateOmega-func-aux-b*: $(\forall\ n.\ eqtop\ n\ ((f\hat{}\hat{}n)\ x)\ y) = (\forall\ n{::}nat\ .\ \forall\ i{::}nat < n\ .\ apply\ ((f\ \hat{}\hat{}\ n)\ x)\ i = apply\ y\ i)$

lemma *IterateOmega-func-aa*: $IterateOmega\ [-\ f\ -] = [:\ x \rightsquigarrow y\ .\ (\forall\ n\ .\ \forall\ i{::}nat < n\ .\ apply\ ((f\ \hat{}\hat{}\ n)\ x)\ i = apply\ y\ i):]$

lemma *IterateOmega-func-b*: $(\forall\ x\ n\ .\ \forall\ i < n\ .\ (f\ \hat{}\hat{}\ n)\ x\ i = (f\ \hat{}\hat{}\ (Suc\ i))\ x\ i) \Longrightarrow IterateOmega\ [-\ f\ -] = [-\lambda\ x\ .\ (\lambda\ i\ .\ (f\ \hat{}\hat{}\ (Suc\ i))\ x\ i)-]$

lemma *IterateOmega-func-bb*: $(\forall\ x\ n\ .\ \forall\ i{::}nat < n\ .\ apply\ (((f{::}(\ ((nat \Rightarrow 'a) \times (nat \Rightarrow 'b)) \times (nat \Rightarrow 'c) \Rightarrow\ ((nat \Rightarrow 'a) \times (nat \Rightarrow 'b)) \times (nat \Rightarrow 'c))\ )\ \hat{}\hat{}\ n)\ x)\ i = apply\ ((f\ \hat{}\hat{}\ (Suc\ i))\ x)\ i)$
$\Longrightarrow$
$IterateOmega\ [-\ f\ -] = [-\ (\lambda\ x\ .\ (let\ z = (\lambda\ i\ .\ apply\ ((f\ \hat{}\hat{}\ (Suc\ i))\ x)\ i)\ in\ ((fst\ o\ fst\ o\ z,\ snd\ o\ fst\ o\ z),\ snd\ o\ z)))\ -]$

lemma *IterateOmega-func-c*: $\forall\ x\ .\ \neg\ (\forall\ n\ .\ \forall\ i < n\ .\ (f\ \hat{}\hat{}\ n)\ x\ i = (f\ \hat{}\hat{}\ (Suc\ i))\ x\ i) \Longrightarrow IterateOmega\ [-\ f\ -] = Magic$

lemma *IterateOmega-assert-update*: $IterateOmega\ (\{.p.\}\ o\ [-f-])$
$= \{.\ (\lambda x\ .\ \forall\ n\ .\ p\ ((f\ \hat{}\hat{}\ n)\ x))\ .\}$

$\circ$ [: $x \rightsquigarrow y$. $\forall$ n. *eqtop* n ((f ^^ n) x) y :]

lemma *IterateOmega-assert-update-a*: *IterateOmega* ({.p.} o[− f −]) = {. (λx . $\forall$ n . p ((f ^^ n) x)) .} o [: $x \rightsquigarrow y$. ($\forall$ n . $\forall$ $i < n$. (f ^^ n) x i = y i):]

lemma *IterateOmega-assert-update-b*: ($\forall$ x n . $\forall$ $i < n$. (f ^^ n) x i = (f ^^ (*Suc* i)) x i) $\Longrightarrow$ *IterateOmega* ({.p.} o[− f −]) = {.(λx . $\forall$ n . p ((f ^^ n) x)).} o [−λ x . (λ i . (f ^^ (*Suc* i)) x i)−]

lemma *IterateOmega-assert-update-c*: *IterateOmega* ({.p.} o [− f −]) = {.(λx . $\forall$ n . p ((f ^^ n) x)).} o [: $x \rightsquigarrow y$. ($\forall$ n . $\forall$ i::*nat* $< n$. *apply* ((f ^^ n) x) i = *apply* y i):]

thm *IterateOmega-spec*

lemma *IterateOmega-assert-update-d*: ($\forall$ x n . $\forall$ i::*nat* $< n$. *apply* (((*f*::(((*nat* $\Rightarrow$ *'a*) $\times$ (*nat* $\Rightarrow$ *'b*)) $\times$ (*nat* $\Rightarrow$ *'c*) $\Rightarrow$ ((*nat* $\Rightarrow$ *'a*) $\times$ (*nat* $\Rightarrow$ *'b*)) $\times$ (*nat* $\Rightarrow$ *'c*))) ^^ n) x) i = *apply* ((f ^^ (*Suc* i)) x) i) $\Longrightarrow$
IterateOmega ({.p.} o [− f −]) = {.(λx . $\forall$ n . p ((f ^^ n) x)).} o [− (λ x . (*let* z = (λ i . *apply* ((f ^^ (*Suc* i)) x) i) *in* ((*fst o fst o* z, *snd o fst o* z), *snd o* z))) −]

lemma *IterateOmega-assert-update-e*: $\forall$ x . $\neg$ ($\forall$ n . $\forall$ $i < n$. (f ^^ n) x i = (f ^^ (*Suc* i)) x i) $\wedge$ ($\forall n$. p ((f ^^n) x)) $\Longrightarrow$ *IterateOmega* ({.p.} o [− f −]) = *Magic*

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

fun *calcu* :: (*nat* $\Rightarrow$ *'a*) $\Rightarrow$ (*nat* $\Rightarrow$ *'b*) $\Rightarrow$ (*'a* $\times$ *'b* $\Rightarrow$ *'a* $\times$ *'c* $\Rightarrow$ *bool*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a* **where**
calcu u x r n i = (*if* $i \leq n$ *then* u i *else SOME* u' . ($\exists y$. r (*calcu* u x r n (i−1), x (i−1)) (u', y)))

thm *choice-iff'*

lemma *prec-loc-st-defined-simp*: *defined* $r \Longrightarrow$ *prec-pre-sts init* p r
= (λ x . $\forall$ u . *init* (u 0) $\longrightarrow$ ($\forall n$. $\exists$ y. r (u n, x n) (u (*Suc* n), y)) $\longrightarrow$ ($\forall$ n . p (u n, x n)))

lemma *DelayFeedback-defined-simp*: *defined* $r \Longrightarrow$ *DelayFeedback init* ({.p.} o [:r:])
= {.x . $\forall$ (u::*nat* $\Rightarrow$ *'a*) . *init* (u 0) $\wedge$ (($\forall n$. $\exists$ y. r (u n, x n) (u (*Suc* n), y))) $\longrightarrow$ ($\forall$ n . p (u n, x n)).}
o [:*rel-pre-sts init* r :]

lemma *defined-fun*[*simp*]: *defined* (λ x y . y = f x)

definition *map-f f x n* = f (*fst* x n, *snd* x n)

lemma *DelayFeedback-update-simp-aux-b*: ($\forall n$. $\exists y$. (u (*Suc* n), y) = f (u n, x n)) = (($\odot$ u) = *map-f* (*fst o* f) (u, x))

lemma *DelayFeedback-update-simp-aux-a*: *rel-pre-sts init* (λx y. y = f x) = (λ x y . $\exists u$. *init* (u 0) $\wedge$ $\odot$ u = *map-f* (*fst* $\circ$ f) (u, x) $\wedge$ y = *map-f* (*snd* $\circ$ f) (u, x))

lemma *DelayFeedback-update-simp*: *DelayFeedback init* ({.*p*.} *o* [−*f*−])
= {. λ*x*. ∀ (*u*::*nat* ⇒ *'a*). *init* (*u* 0) ∧ (⊙ *u*) = *map-f* (*fst o f*) (*u*, *x*) ⟶ (∀ *n* . *p* (*u n*, *x n*)) .}
∘ [: λ*x y*. ∃ (*u*:: *nat* ⇒ *'a*). *init* (*u* 0) ∧ (⊙ *u*) = *map-f* (*fst o f*) (*u*, *x*) ∧ *y* = *map-f* (*snd o f*) (*u*, *x*) :]

primrec *itr* :: (*'a* × *'b* ⇒ *'a*) ⇒ *'a* ⇒ (*nat* ⇒ *'b*) ⇒ *nat* ⇒ *'a* **where**
itr f u0 x 0 = *u0* |
itr f u0 x (*Suc n*) = *f* (*itr f u0 x n*, *x n*)

lemma *map-itr-aux*: ((⊙ *u*) = *map-f f* (*u*, *x*)) ⟹ (*u n* = *itr f* (*u* 0) *x n*)

lemma *map-itr-simp*: ((⊙ *u*) = *map-f f* (*u*, *x*)) = (*u* = *itr f* (*u* 0) *x*)

lemma *DelayFeedback-update-itr-simp*: *DelayFeedback init* ({.*p*.} *o* [−*f*−])
= {. *x*. ∀ *a* . *init a* ⟶ (∀ *i* . *p* (*itr* (*fst o f*) *a x i*, *x i*)) .}
∘ [: λ*x y*. ∃ *a*. *init a* ∧ *y* = *map-f* (*snd o f*) (*itr* (*fst o f*) *a x*, *x*) :]

definition *DelayFeedbackInit a S* = *DelayFeedback* (λ *u* . *u* = *a*) *S*

definition *lft-1-2 p* = (λ (*x*, *y*) . *p* (*x* (*0*::*nat*), *y* (*0*::*nat*)))
definition *lft-2-2 r* = (λ (*x*, *y*) (*z*, *t*). *r* (*x* (*0*::*nat*), *y* (*0*::*nat*)) (*z* (*0*::*nat*), *t* (*0*::*nat*)))

theorem *DelayFeedbackInit-update-simp-a*: *DelayFeedbackInit u* ({.*p*.} *o* [−*f*−])
= {. *x*. (∀ *n* . *p* (*itr* (*fst o f*) *u x n*, *x n*)) .} ∘ [−λ*x* . *map-f* (*snd o f*) (*itr* (*fst o f*) *u x*, *x*)−]

lemma [*simp*]: (□ *lft-1-2* ⊤) = ⊤

theorem *DelayFeedbackInit-update-simp-b*: *DelayFeedbackInit u* [−*f*−] = [−λ*x* . *map-f* (*snd o f*) (*itr* (*fst o f*) *u x*, *x*)−]

lemma *prec-itr-simp*: ((□ *lft-1-2 p*) (*itr f u x*, *x*)) = (∀ *n*. *p* (*itr f u x n*, *x n*))

lemma *prec-itr-induction-aux*: *p* (*u*, *x* 0) ⟹ (⋀ *n a* . *p* (*a*, *x n*) ⟹ *p* (*f* (*a* ,*x n*), *x* (*Suc n*))) ⟹ *p* (*itr f u x n*, *x n*)

lemma *prec-itr-induction*: *p* (*u*, *x* 0) ⟹ (⋀ *n a* . *p* (*a*, *x n*) ⟹ *p* (*f* (*a* ,*x n*), *x* (*Suc n*))) ⟹ ((□ *lft-1-2 p*) (*itr f u x*, *x*))

definition *lft-r r x y* = *r* (*fst x 0*, *snd x 0*) (*fst y 0*, *snd y 0*)
definition *lft-r-b r x y* = *r* (*x 0*) (*y 0*)

lemma *rel-itr-simp*: (□ (*lft-r-b r*)) *x* (*map-f g* (*itr f u x*, *x*)) = (∀ *n* . *r* (*x n*) (*g* (*itr f u x n*, *x n*)))

lemma *rel-itr-induction-aux*: *r* (*x* 0) (*g* (*u*, *x* 0)) ⟹ (⋀ *n a* . *r* (*x n*) (*g* (*a*, *x n*)) ⟹ *r* (*x* (*Suc n*)) (*g* (*f* (*a* ,*x n*), *x* (*Suc n*)))) ⟹ *r* (*x n*) (*g* (*itr f u x n*, *x n*))

lemma *rel-itr-induction*: *r* (*x* 0) (*g* (*u*, *x* 0)) ⟹ (⋀ *n a* . *r* (*x n*) (*g* (*a*, *x n*)) ⟹ *r* (*x* (*Suc n*)) (*g* (*f* (*a* ,*x n*), *x* (*Suc n*)))) ⟹ (□ (*lft-r-b r*)) *x* (*map-f g* (*itr f u x*, *x*))

lemma *rel-bounded-itr-induction-aux*: (*0* ∈ *b* ⟹ *r* (*x* 0) (*g* (*u*, *x* 0))) ⟹

$(\bigwedge\ n\ a\ .\ (n \in b \Longrightarrow r\ (x\ n)\ (g\ (a,\ x\ n))) \Longrightarrow Suc\ n \in b \Longrightarrow r\ (x\ (Suc\ n))\ (g\ (f\ (a\ ,x\ n),\ x\ (Suc\ n)))) \Longrightarrow n \in b \Longrightarrow r\ (x\ n)\ (g\ (itr\ f\ u\ x\ n,\ x\ n))$

lemma *rel-bounded-itr-induction*: $(0 \in b \Longrightarrow r\ (x\ 0)\ (g\ (u,\ x\ 0))) \Longrightarrow (\bigwedge\ n\ a\ .\ (n \in b \Longrightarrow r\ (x\ n)\ (g\ (a,\ x\ n))) \Longrightarrow Suc\ n \in b \Longrightarrow r\ (x\ (Suc\ n))\ (g\ (f\ (a\ ,x\ n),\ x\ (Suc\ n))))$
$\Longrightarrow (\Box b\ b\ (lft\text{-}r\text{-}b\ r))\ x\ (map\text{-}f\ g\ (itr\ f\ u\ x,\ x))$

lemma *refin-demonic-spec*: $([:r:] \leq \{.p.\}\ o\ [:r':]) = (p = \top \wedge r' \leq r)$

lemma *spec-delay-feedback-fun-refine*: $(\{.p'.\}\ o\ [:\ r\ :] \leq DelayFeedbackInit\ u\ (\{.p.\}\ o\ [-f-])) = ((p' \leq (\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (fst\ o\ f)\ u\ x,\ x)))$
$\wedge\ (\forall\ x\ .\ p'\ x \longrightarrow\ r\ x\ (\ map\text{-}f\ (snd\ o\ f)\ (itr\ (fst\ o\ f)\ u\ x,\ x))\ ))$

lemma *prec-itr-inductionA*: $(p'\ x \Longrightarrow p\ (u,\ x\ 0)) \Longrightarrow (\bigwedge\ n\ a\ .\ p'\ x \Longrightarrow p\ (a,\ x\ n) \Longrightarrow p\ (f\ (a\ ,x\ n),\ x\ (Suc\ n))) \Longrightarrow p'\ x \Longrightarrow ((\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ f\ u\ x,\ x))$

lemma *prec-itr-inductionB*: $(\bigwedge\ x\ .\ p'\ x \Longrightarrow p\ (u,\ x\ 0)) \Longrightarrow (\bigwedge\ x\ n\ a\ .\ p'\ x \Longrightarrow p\ (a,\ x\ n) \Longrightarrow p\ (f\ (a\ ,x\ n),\ x\ (Suc\ n))) \Longrightarrow p' \leq (\lambda\ x\ .\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ f\ u\ x,\ x))$

lemma *rel-itr-inductionA*: $(\bigwedge\ x\ .\ p'\ x \Longrightarrow r\ (x\ 0)\ (g\ (u,\ x\ 0))) \Longrightarrow (\bigwedge\ x\ n\ a\ .\ p'\ x \Longrightarrow r\ (x\ n)\ (g\ (a,\ x\ n)) \Longrightarrow r\ (x\ (Suc\ n))\ (g\ (f\ (a\ ,x\ n),\ x\ (Suc\ n))))$
$\Longrightarrow p'\ x \Longrightarrow (\Box\ (lft\text{-}r\text{-}b\ r))\ x\ (map\text{-}f\ g\ (itr\ f\ u\ x,\ x))$

lemma $\{:z \rightsquigarrow x\ .\ x \neq (0::nat):\}\ o\ [:x \rightsquigarrow y\ .\ x = 0 \wedge y = (0::nat):] = \top$

lemma $\{:z \rightsquigarrow x\ .\ x \neq (Suc\ n):\}\ o\ [:x \rightsquigarrow y\ .\ x = 0 \wedge y = (0::nat):] = \top$

lemma $(\{.p'.\}\ o\ [:\ \Box\ (lft\text{-}r\text{-}b\ r)\ :] \leq DelayFeedbackInit\ u\ (\{.p.\}\ o\ [-f-])) = ((p' \leq (\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (fst\ o\ f)\ u\ x,\ x)))$
$\wedge\ (\forall\ x\ .\ p'\ x \longrightarrow\ (\Box\ (lft\text{-}r\text{-}b\ r))\ x\ (\ map\text{-}f\ (snd\ o\ f)\ (itr\ (fst\ o\ f)\ u\ x,\ x))\ ))$

lemma *demonic-delay-feedback-fun-refine*: $([:r\ :] \leq DelayFeedbackInit\ u\ (\{.p.\}\ o\ [-f-])) = (((\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (fst\ o\ f)\ u\ x,\ x)) = \top)$
$\wedge\ (\forall\ x\ .\ r\ x\ (map\text{-}f\ (snd\ o\ f)\ (itr\ (fst\ o\ f)\ u\ x,\ x))\ ))$

lemma $([:\ \Box\ (lft\text{-}r\text{-}b\ r)\ :] \leq DelayFeedbackInit\ u\ (\{.p.\}\ o\ [-f-])) = (((\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (fst\ o\ f)\ u\ x,\ x)) = \top)$
$\wedge\ (\forall\ x\ .\ (\Box\ (lft\text{-}r\text{-}b\ r))\ x\ (map\text{-}f\ (snd\ o\ f)\ (itr\ (fst\ o\ f)\ u\ x,\ x))\ ))$

lemma *refin-update-spec*: $([:\ \Box b\ b\ (lft\text{-}r\text{-}b\ r)\ :] \leq DelayFeedbackInit\ u\ (\{.p.\}\ o\ [-f-])) = (((\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (fst\ o\ f)\ u\ x,\ x)) = \top)$
$\wedge\ (\forall\ x\ y\ .\ y = map\text{-}f\ (snd\ o\ f)\ (itr\ (fst\ o\ f)\ u\ x,\ x) \longrightarrow\ (\Box b\ b\ (lft\text{-}r\text{-}b\ r))\ x\ y\ ))$

definition *prec-delay p f-state u* $= (\lambda x.\ (\Box\ lft\text{-}1\text{-}2\ p)\ (itr\ (f\text{-}state)\ u\ x,\ x))$
definition *func-delay f-state f-out u* $= (\lambda x\ .\ map\text{-}f\ f\text{-}out\ (itr\ f\text{-}state\ u\ x,\ x))$

theorem *DelayFeedbackInit-update-simp-c*: *DelayFeedbackInit u* $(\{.p.\}\ o\ [-f-])$
$= \{.prec\text{-}delay\ p\ (fst\ o\ f)\ u.\} \circ [-func\text{-}delay\ (fst\ o\ f)\ (snd\ o\ f)\ u-]$

theorem *DelayFeedbackInit-update-simp-d*: *DelayFeedbackInit u* $[-f-] =\ [-func\text{-}delay\ (fst\ o\ f)\ (snd\ o\ f)\ u-]$

lemma *always-lft-bot*: $(\Box\ \mathit{lft\text{-}1\text{-}2}\ (\bot{::}('a \times 'b \Rightarrow \mathit{bool})\ )) = \bot$

lemma *DelayFeedbackInit-bot*: $\mathit{DelayFeedbackInit}\ u\ ((\bot{::}(('a \times 'b) \Rightarrow \mathit{bool}) \Rightarrow ('a \times 'c) \Rightarrow \mathit{bool})) = \bot$

lemma *simp-prec*: $\{.\ p\ .\} \circ [:\ \lambda x\ y.\ \neg\ p\ x \vee r\ x\ y\ :] = \{.p.\}\ o\ [:r:]$

lemma *inpt-and-rel*: $(\mathit{inpt}\ r\ x \wedge r\ x\ y) = r\ x\ y$

lemma [*simp*]: $\mathit{inpt}\ (\lambda x\ y.\ \mathit{inpt}\ r\ x \wedge r\ x\ y) = \mathit{inpt}\ r$

thm *DelayFeedback-defined-simp*
lemma *DelayFeedback-inpt*: $\mathit{DelayFeedback}\ \mathit{init}\ (\{.\mathit{inpt}\ r.\}\ o\ [:r:])$
$= \{.x.\ \forall (u{::}\mathit{nat} \Rightarrow 'a).\ \mathit{init}\ (u\ 0) \wedge ((\forall n.\ \exists y.\ \neg\ \mathit{inpt}\ r\ (u\ n,\ x\ n) \vee r\ (u\ n,\ x\ n)\ (u\ (\mathit{Suc}\ n),\ y)))$
$\longrightarrow (\forall n.\ \mathit{inpt}\ r\ (u\ n,\ x\ n)).\} \circ$
$[:\ \mathit{rel\text{-}pre\text{-}sts}\ \mathit{init}\ (\lambda x\ y.\ \neg\ \mathit{inpt}\ r\ x \vee r\ x\ y)\ :]$

declare *comp-skip*[*simp del*]
declare *skip-comp*[*simp del*]
declare *prod-skip-skip*[*simp del*]
declare *fail-comp*[*simp del*]

4.7 Data Refinement

definition $\mathit{data\text{-}refin\text{-}sts}\ d\ S\ S' = (\{:t,\ x\ \rightsquigarrow s,\ x'\ .\ x = x' \wedge d\ t\ s:\}\ o\ S \leq S'\ o\ \{:t',\ y \rightsquigarrow s',\ y'\ .\ y = y' \wedge d\ t'\ s':\})$

lemma *data-refin-sts-simp*: $\mathit{data\text{-}refin\text{-}sts}\ d\ (\{.\ p\ .\} \circ [:\ r\ :])\ (\{.\ p'\ .\} \circ [:\ r'\ :]) =$
$((\forall t\ x\ s.\ d\ t\ s \wedge p\ (s,\ x) \longrightarrow p'\ (t,\ x)) \wedge$
$(\forall t\ x\ s\ t'\ y.\ d\ t\ s \wedge p\ (s,\ x) \wedge r'\ (t,\ x)\ (t',\ y) \longrightarrow (\exists s'.\ d\ t'\ s' \wedge r\ (s,\ x)\ (s',\ y))))$

primrec $\mathit{s\text{-}r} :: ('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow ('b \Rightarrow \mathit{bool}) \Rightarrow ('b \times 'c \Rightarrow 'b \times 'd \Rightarrow \mathit{bool}) \Rightarrow (\mathit{nat} \Rightarrow 'c) \Rightarrow (\mathit{nat} \Rightarrow 'd) \Rightarrow (\mathit{nat} \Rightarrow 'a) \Rightarrow \mathit{nat} \Rightarrow 'b$ **where**
$\mathit{s\text{-}r}\ d\ \mathit{init}\ r\ x\ y\ t\ 0 = (\mathit{SOME}\ s\ .\ d\ (t\ 0)\ s \wedge \mathit{init}\ s)\ |$
$\mathit{s\text{-}r}\ d\ \mathit{init}\ r\ x\ y\ t\ (\mathit{Suc}\ n) = (\mathit{SOME}\ s\ .\ d\ (t\ (\mathit{Suc}\ n))\ s \wedge r\ (\mathit{s\text{-}r}\ d\ \mathit{init}\ r\ x\ y\ t\ n,\ x\ n)\ (s,\ y\ n))$

theorem *data-refinement-sts*: $(\bigwedge t\ .\ \mathit{init}'\ t \Longrightarrow \exists\ s\ .\ d\ t\ s \wedge \mathit{init}\ s) \Longrightarrow$
$\mathit{data\text{-}refin\text{-}sts}\ d\ (\{.p.\}\ o\ [:r:])\ (\{.p'.\}\ o\ [:r':]) \Longrightarrow \mathit{LocalSystem}\ \mathit{init}\ p\ r \leq \mathit{LocalSystem}\ \mathit{init}'\ p'\ r'$

4.8 Reachability and Refinement

definition $\mathit{reach}\ \mathit{init}\ r\ n\ x\ y\ s = (\mathit{init}\ (s\ 0) \wedge (\forall\ i < n\ .\ r\ (s\ i,\ x\ i)\ (s\ (\mathit{Suc}\ i),\ y\ i)))$

lemma *reach-prec-always*: $\mathit{reach}\ \mathit{init}\ r\ n\ x\ y\ s \Longrightarrow p \leq \mathit{inpt}\ r \Longrightarrow \mathit{prec\text{-}pre\text{-}sts}\ \mathit{init}\ p\ r\ x$
$\Longrightarrow \exists\ s'\ y'\ .\ \mathit{init}\ (s'\ 0) \wedge (\forall\ i < n\ .\ y'\ i = y\ i) \wedge (\forall\ i \leq n\ .\ s'\ i = s\ i) \wedge (\Box\ \mathit{lift\text{-}rel}\ r)\ (s',\ x)\ (s'[1..],\ y')$

lemma *refinemen-reachable-B*:
assumes *R*: $\mathit{LocalSystem}\ \mathit{init}\ p\ r \leq \mathit{LocalSystem}\ \mathit{init}'\ p'\ r'$
and [*simp*]: $p' \leq \mathit{inpt}\ r'$
shows $\mathit{prec\text{-}pre\text{-}sts}\ \mathit{init}\ p\ r\ x \Longrightarrow \mathit{reach}\ \mathit{init}'\ r'\ n\ x\ y\ t \Longrightarrow \exists\ s\ .\ \mathit{reach}\ \mathit{init}\ r\ n\ x\ y\ s$
and $\mathit{prec\text{-}pre\text{-}sts}\ \mathit{init}\ p\ r\ x \Longrightarrow \mathit{reach}\ \mathit{init}'\ r'\ n\ x\ y\ t \Longrightarrow p'\ (t\ n,\ x\ n)$

lemma *sel-inf-a*: *finite* $X \Longrightarrow (\bigwedge i :: nat \;.\; f\ i \in X) \Longrightarrow (\exists\ x \in X \;.\; infinite\ \{i \;.\; f\ i = x\})$

lemma $X \neq \{\} \Longrightarrow \exists\ (x::'a::wellorder) \in X \;.\; \forall\ y \in X \;.\; x \leq y$

primrec *min-rest* :: *nat set* ⇒ *nat* ⇒ *nat* **where**
min-rest X 0 = (*LEAST x . x ∈ X*) |
min-rest X (*Suc n*) = *min-rest* (*X* − {*LEAST x . x ∈ X*}) *n*

lemma *sel-inf-fun*: $\bigwedge X \;.\; infinite\ X \Longrightarrow min\text{-}rest\ X\ n \in X \wedge min\text{-}rest\ X\ n < min\text{-}rest\ X\ (Suc\ n)$

lemma *sel-inf*: *finite X* ⟹ (⋀ *i* :: *nat* . *f i* ∈ *X*) ⟹ (∃ *g x* . *x* ∈ *X* ∧ (∀ *i* . *f* (*g i*) = *x*) ∧ (∀ *i* . *g i* < *g* (*Suc i*)))

definition *sel-inf f X* = (*SOME g* . ∃ *x* . *x* ∈ *X* ∧ (∀ *i* . *f* (*g i*) = *x*) ∧ (∀ *i* . *g i* < *g* (*Suc i*)))

lemma *sel-inf-prop-aux*: *finite X* ⟹ (⋀ *i* :: *nat* . *f i* ∈ *X*) ⟹ (∃ *x* . *x* ∈ *X* ∧ (∀ *i* . *f* (*sel-inf f X i*) = *x*) ∧ (∀ *i* . *sel-inf f X i* < *sel-inf f X* (*Suc i*)))

lemma *sel-inf-prop*:
assumes *A*: *finite X* **and** *B*: (⋀ *i* :: *nat* . *f i* ∈ *X*)
shows *f* (*sel-inf f X i*) = *f* (*sel-inf f X 0*) **and** ⋀ *i* . *sel-inf f X i* < *sel-inf f X* (*Suc i*)
and *i* ≤ *sel-inf f X i*

fun *SSa* :: (*'a* ⇒ *bool*) ⇒ (*'a* × *'b* ⇒ *'a* × *'c* ⇒ *bool*) ⇒ (*nat* ⇒ *'b*) ⇒ (*nat* ⇒ *nat* ⇒ *'a*) ⇒ *nat* ⇒ *nat* ⇒ *'a* **where**
SSa init r x s 0 = (*s*[*Suc 0*..] *o sel-inf* (λ *i* . *s* (*Suc i*) *0*) {*s* . *init s*}) |
SSa init r x s (*Suc n*) = ((*SSa init r x s n*[*Suc 0*..]) *o*
sel-inf (λ *i* . *SSa init r x s n* (*Suc i*) (*Suc n*)) {*s'* . ∃ *y* . *r* ((*SSa init r x s n*[*Suc 0*..]) *0 n*, *x n*) (*s'*, *y*) })

lemma *refinemen-reachable-aux*:
assumes *finite-next*: ⋀ *s x* . *finite* {*s'* . ∃ *y* . *r* (*s*, *x*) (*s'*, *y*)}
and *finite-init*[*simp*]: *finite* {*s* . *init s*}
assumes *A*: (⋀ *n* . *reach init r* (*Suc n*) *x y* (*s n*))
shows (∀ *j* . ∀ *k* ≤ *n* . *SSa init r x s n j k* = *SSa init r x s n 0 k*) ∧ *reach init r n x y* (*SSa init r x s n n*)
∧ (∃ *k* . ∀ *i* . *k i* ≥ *n* ∧ *SSa init r x s n i* = *s* (*k i*) ∧ *k i* < *k* (*Suc i*))
∧ (∀ *j* . ∀ *k* ≤ *n* . *SSa init r x s* (*Suc n*) *j k* = *SSa init r x s n 0 k*)

lemma *refinemen-reachable-A*:
assumes *finite-next*: ⋀ *s x* . *finite* {*s'* . ∃ *y* . *r* (*s*, *x*) (*s'*, *y*)}
and *finite-init*: *finite* {*s* . *init s*}
assumes *A*: ⋀ *n x y t* . *prec-pre-sts init p r x* ⟹ *reach init' r' n x y t* ⟹ *p'* (*t n*, *x n*)
and *B*: ⋀ *n x y t* . *prec-pre-sts init p r x* ⟹ *reach init' r' n x y t* ⟹ ∃ *s* . *reach init r n x y s*
shows *LocalSystem init p r* ≤ *LocalSystem init' p' r'*

definition *symb-sts-refin init p r init' p' r'*
=

$((\forall\ n\ x\ y\ t\ .\ prec\text{-}pre\text{-}sts\ init\ p\ r\ x \longrightarrow reach\ init'\ r'\ n\ x\ y\ t \longrightarrow p'\ (t\ n,\ x\ n))$
$\wedge\ (\forall\ n\ x\ y\ t\ .\ prec\text{-}pre\text{-}sts\ init\ p\ r\ x \longrightarrow reach\ init'\ r'\ n\ x\ y\ t \longrightarrow (\exists\ s\ .\ reach\ init\ r\ n\ x\ y\ s)))$

lemma *refinemen-reachable-iff*:
assumes *finite-next*[*simp*]: $\bigwedge\ s\ x\ .\ finite\ \{s'\ .\ \exists\ y\ .\ r\ (s,\ x)\ (s',\ y)\}$
and *finite-init*[*simp*]: *finite* $\{s\ .\ init\ s\}$
and [*simp*]: $p' \leq inpt\ r'$
shows *LocalSystem init p r* $\leq$ *LocalSystem init' p' r'* $=$ *symb-sts-refin init p r init' p' r'*

definition *inv-top n P* $= (\forall\ u\ v\ .\ eqtop\ n\ u\ v \longrightarrow (P\ u = P\ v))$
definition *prec-pre-sts-bound init p r N x* $= ((\forall\, u.\ init\ (u\ 0) \longrightarrow (\forall\, y\ .\ \forall\ n < N.\ (\forall\, i{<}n.\ r\ (u\ i,\ x\ i)\ (u\ (Suc\ i),\ y\ i)) \longrightarrow p\ (u\ n,\ x\ n))))$

lemma *replace-variables*: $(inv\text{-}top\ (Suc\ N)\ (P\ N)) \Longrightarrow (inv\text{-}top\ N\ (R\ N)) \Longrightarrow (inv\text{-}top\ N\ (Q'\ N)) \Longrightarrow$
$(\forall\ (x{::}nat \Rightarrow {'}z)\ .\ P\ N\ x \wedge (ZZ\ (Q'\ N\ x)\ (Q\ N\ (x[N..]))) \wedge R\ N\ x \longrightarrow S\ N\ (x\ N))$
$=\ (\forall\ x\ xN\ y\ .\ P\ N\ (x(N := xN)) \wedge y\ 0 = xN \wedge (ZZ\ (Q'\ N\ x)\ (\ Q\ N\ (y))) \wedge R\ N\ x \longrightarrow S\ N\ (xN))$

lemma *prec-pre-sts-reach*: $\bigwedge\ x\ .\ prec\text{-}pre\text{-}sts\ init\ p\ r\ x = (\forall\ s\ n.\ (\exists\ y\ .\ reach\ init\ r\ n\ x\ y\ s) \longrightarrow p\ (s\ n,\ x\ n))$

lemma *prec-pre-sts-bound-simp*: $\bigwedge\ N\ x\ .\ prec\text{-}pre\text{-}sts\text{-}bound\ init\ p\ r\ N\ x =$
$(\forall\, u\ n.\ (n < N \wedge init\ (u\ 0) \wedge ((\exists\ y\ .\ \forall\, i{<}n\ .\ r\ (u\ i,\ x\ i)\ (u\ (Suc\ i),\ y\ i)))) \longrightarrow (\forall\ k \leq n\ .\ p\ (u\ k,\ x\ k)))$

lemma *prec-pre-sts-bound*: $\bigwedge\ x\ N\ .\ prec\text{-}pre\text{-}sts\ init\ p\ r\ x = (prec\text{-}pre\text{-}sts\text{-}bound\ init\ p\ r\ N\ x$
$\wedge\ (\forall\ s\ y\ .\ reach\ init\ r\ N\ x\ y\ s \longrightarrow\ prec\text{-}pre\text{-}sts\ (\lambda\ u\ .\ u = s\ N)\ p\ r\ (x[N..])))$

lemma *AA*: $\bigwedge\ t\ x\ N\ y\ .\ ((prec\text{-}pre\text{-}sts\ init\ p\ r\ x \wedge reach\ init'\ r'\ N\ x\ y\ t) \longrightarrow p'\ (t\ N,\ x\ N))$
$= ((prec\text{-}pre\text{-}sts\text{-}bound\ init\ p\ r\ N\ x \wedge (\forall\ s\ y\ .\ reach\ init\ r\ N\ x\ y\ s \longrightarrow\ prec\text{-}pre\text{-}sts\ (\lambda\ u\ .\ u = s\ N)\ p\ r\ (x[N..])) \wedge reach\ init'\ r'\ N\ x\ y\ t) \longrightarrow p'\ (t\ N,\ x\ N))$

lemma [*simp*]: *inv-top* $(Suc\ N)$ $(prec\text{-}pre\text{-}sts\text{-}bound\ init\ p\ r\ N)$

lemma [*simp*]: *inv-top N* $(\lambda x.\ \exists\, y.\ reach\ init'\ r'\ N\ x\ y\ t)$

lemma [*simp*]: *inv-top N* $(\lambda x\ s.\ \exists\, y.\ reach\ init\ r\ N\ x\ y\ s)$

lemma *sts-refinement-A-bounded*: $(\forall\ x\ y\ .\ (prec\text{-}pre\text{-}sts\ init\ p\ r\ x \wedge reach\ init'\ r'\ N\ x\ y\ t) \longrightarrow p'\ (t\ N,\ x\ N))$
$=\ (\forall\ xN\ .\ (\exists\ x\ .\ prec\text{-}pre\text{-}sts\text{-}bound\ init\ p\ r\ N\ (x(N := xN))$
$\wedge\ (\exists\ xz\ .\ xz\ 0 = xN \wedge (\forall\ s\ y\ .\ reach\ init\ r\ N\ x\ y\ s \longrightarrow\ prec\text{-}pre\text{-}sts\ (\lambda\ u\ .\ u = s\ N)\ p\ r\ xz))$
$\wedge\ (\exists\ y\ .\ reach\ init'\ r'\ N\ x\ y\ t))$
$\longrightarrow p'\ (t\ N,\ xN))$

lemma *reach-until*: $(\exists\ x\ s\ y\ n.\ reach\ init\ r\ n\ x\ y\ s \wedge s\ n = t)$
$= (\exists\ sa\ .\ init\ (sa\ 0) \wedge ((\lambda\ sa\ .\ (\exists\ x\ y\ .\ r\ (sa\ 0,\ x)\ (sa\ (Suc\ 0),\ y)))\ until\ (\lambda\ sa\ .\ sa\ 0 = t))\ sa)$

lemma *LocalSystem-prec-top*: *LocalSystem init* $\top$ *r* $=$ [: *rel-pre-sts init r*:]

lemma *LocalSystem-input-complete*: $(LocalSystem\ init\ p\ r = [:rel\text{-}pre\text{-}sts\ init\ r:])$
$= ((\forall\ x\ s\ .\ init\ s \longrightarrow p\ (s,x)) \wedge$
$(\forall\ s\ s'\ x\ x'\ y\ n\ .$

$(\exists\ x\ y\ .\ reach\ init\ r\ n\ x\ y\ s) \wedge p\ (s\ n,\ x) \wedge r\ (s\ n,x)\ (s',\ y) \longrightarrow p\ (s',\ x'))$

end

4.9 Reactive Feedback

theory *ReactiveFeedback*
imports *TransitionFeedback IterateOperators*

begin

definition *Feedback S* = $\{:x \rightsquigarrow (u,\ y),\ x'\ .\ (x = x'):\}$ *o IterateOmegaA* $([-\lambda\ ((u,\ y),\ x)\ .\ ((u,\ x),\ x)-]$ *o* $(S$ **** *Skip*$))$ *o* $[-\lambda\ ((u,y),\ x)\ .\ y\ -]$

lemma *Feedback-refin*: $S \leq T \Longrightarrow$ *Feedback* $S \leq$ *Feedback* T

definition *FeedbackX Init S* = $[:x \rightsquigarrow (u,\ y),\ x'\ .\ (u = ()) \wedge (x = x'):]$ *o* $((Init$ **** *Skip*$)$ **** *Skip*$)$ *o IterateOmega* $([-\lambda\ ((u,\ y),\ x)\ .\ ((u,\ x),\ x)-]$ *o* $(S$ **** *Skip*$))$ *o* $[-\lambda\ ((u,y),\ x)\ .\ y\ -]$

definition *FeedbackA Init S* = $[:x \rightsquigarrow (x'',\ y),\ x'\ .\ (x'' = x) \wedge (x = x'):]$ *o* $((Init$ **** *Skip*$)$ **** *Skip*$)$ *o IterateOmegaA* $([-\lambda\ ((u,\ y),\ x)\ .\ ((u,\ x),\ x)-]$ *o* $(S$ **** *Skip*$))$ *o* $[-\lambda\ ((u,y),\ x)\ .\ y\ -]$

lemma *feedback-update-simp-e*: *feedback* $([-\ \lambda\ (u,\ s,\ x)\ .\ (f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x)\ -])$
$= [-\lambda\ (s,\ x)\ .\ (g\ (f\ s\ x)\ s\ x,\ h\ (f\ s\ x)\ s\ x)-]$

definition *InitDF init* = $[:\ s \rightsquigarrow s'.\ (\Box(\lambda s.\ init\ (s\ (0::nat)))))\ s':]$

definition *Add* = $[-\ \lambda(x,y).\ x+y-]$
definition *UD* = $[-\lambda(x,s).\ (s,x)\ -]$
definition *Split* = $[-\lambda x.\ (x,x)-]$

definition *RT1* = $[-\ \lambda(u,\ (s,x)).\ ((u,x),s)-]$
definition *RT2* = $[-\ \lambda((v,y),s).\ (v,\ (s,y))\ -]$
definition *RT3* = $[-\ \lambda(x,s).\ (s,x)-]$

definition *Res* = $[-\lambda x.\ Summ\ x-]$

definition *init-ExFb* = $(\lambda\ u\ .\ u = (0::nat))$
definition *ExFb* = *RT1 o* $(Add$ **** *Skip*$)$ *o UD o* $(Split$ **** *Skip*$)$ *o RT2*

lemma *ExFb-simp* : *ExFb* = $[-\ \lambda(u,\ (s,x)).\ (s,\ (u+x,s))-]$

definition *ExFb-transfb* = *feedback ExFb*

lemma *ExFb-transfb-simp*: *ExFb-transfb* = $[-\ \lambda(s,x).\ (s+x,s)-]$

definition *ExFb-genfb* = *DelayFeedback init-ExFb ExFb-transfb*

lemma *DelayFeedback-example*: *ExFb-genfb* = *Res*

definition *RT4* = $[-\ \lambda(s,\ (u,\ x)).\ (u,\ (s,\ x))\ -]$

definition *RT5* = [− λ(*v*, (*s*, *y*)). (*s*, (*v*, *y*)) −]

definition *Res-aux* = [−λ(*u*, *x*). ((λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*)),(λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*)))−]

definition *ExFb-delayfb-aux* = *RT4 o ExFb o RT5*

lemma *ExFb-delayfb-aux-simp*: *ExFb-delayfb-aux* = [−λ(*s*, (*u*, *x*)). (*u*+*x*, (*s*, *s*))−]

definition *ExFb-delayfb* = [−λ(*u*,*x*). *nzip u x*−] *o* (*DelayFeedback* (λ *u* . *u* = (*0*::*nat*)) *ExFb-delayfb-aux*) *o* [−λ*x*. (*fst o x*, *snd o x*)−]

lemma *aaa-ind*:∀ *x*. (*x* = *0* ⟶ *aa 0* = *0*) ∧ (*0* < *x* ⟶ *aa x* = *a* (*x* − *Suc 0*) + *b* (*x* − *Suc 0*)) ⟹
∀ *x*. (*x* = *0* ⟶ *ba 0* = *0*) ∧ (*0* < *x* ⟶ *ba x* = *a* (*x* − *Suc 0*) + *b* (*x* − *Suc 0*)) ⟹ (*aa x* = *ba x*)

lemma *ExFb-delayfb-simp*: *ExFb-delayfb* = *Res-aux*

definition *Init-ExFb* = *InitDF init-ExFb*

lemma *Res-aux-simp*: [−λ((*u*, *y*), *x*). ((*u*, *x*), *x*)−] ∘ *Res-aux* ∗∗ *Skip* = [− λ((*u*, *y*), *x*). (((λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*)),(λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*))), *x*) −]

definition *Res-aux-fun* = (λ((*u*::*nat*⇒*nat*, *y*::*nat*⇒*nat*), *x*::*nat*⇒*nat*). (((λ(*i*::*nat*). *if i* = (*0*::*nat*) *then* (*0*::*nat*) *else u* (*i*−(*1*::*nat*)) + *x* (*i*−(*1*::*nat*))), (λ(*i*::*nat*). *if i* = (*0*::*nat*) *then* (*0*::*nat*) *else u* (*i*−(*1*::*nat*)) + *x* (*i*−(*1*::*nat*)))), *x*))

lemma *Res-aux-fun-aux-a*: ⋀ *a b c* . (*Res-aux-fun* ^^ (*n*::*nat*)) *z* = ((*a*,*b*), *c*)
⟹ (∀ *i* < *n* . *a i* = (*Summ c* (*i*::*nat*)) ∧ *b i* = (*Summ c* (*i*::*nat*))) ∧ *c* = (*snd z*)

lemma *Res-aux-fun-aux-b*: (*i* < *n* ⟹ *apply* (((*Res-aux-fun*) ^^ *n*) *z*) *i* = *apply* ((*Res-aux-fun* ^^ (*Suc i*)) *z*) *i*)

lemma *Res-aux-fun-aux-c*: (λ*x*. *let z* = λ*i*. *apply* (*Res-aux-fun* ((*Res-aux-fun* ^^ *i*) *x*)) *i in* ((*fst* ∘ *fst* ∘ *z*, *snd* ∘ *fst* ∘ *z*), *snd* ∘ *z*))
= (λ *x* . ((*Summ* (*snd x*), *Summ* (*snd x*)), *snd x*))

definition *Init-adder3* = [− λ*x*. (λ (*i*::*nat*). (*2*::*nat*))−]
definition *S-adder3* = [− λ (*x*, (*x*′::*nat* ⇒ *unit*)) . *x*−] *o* [− λ*x* . (λ (*i*::*nat*). (*x i*) + *1*)−] *o* [− λ*x* . (λ (*i*::*nat*). *if i* = *0 then* (*0*::*nat*) *else x* (*i*−*1*)) −] *o*
[− λ*x*. (λ (*i*::*nat*) . *x i* + *2*) −] *o* [− λ*x*. (*x*, *x*) −]
definition *Res-adder3* = [− λ*x* . (λ (*i*::*nat*) . *3* ∗ *i* + *2*) −]

definition *S-simp-adder3* = [− λ (*x*, (*x′*::*nat* ⇒ *unit*)). ((λ*i*. *if i* = *0 then 2 else x*(*i*−*1*) + *3*), (λ*i*. *if i* = *0 then 2 else x*(*i*−*1*) + *3*))−]

lemma *S-adder3-simp*: *S-adder3* = *S-simp-adder3*

lemma *Adder3-inner-simp*: [−λ((*u*, *y*), *x*). ((*u*, *x*), *x*)−] ∘ *S-simp-adder3* ∗∗ *Skip* = [− λ((*u*, *y*), *x*). (((λ*i*. *if i* = *0 then 2 else u*(*i*−*1*) + *3*), (λ*i*. *if i* = *0 then 2 else u*(*i*−*1*) + *3*)), *x*) −]

definition *Adder3-iter-fun* = (λ((*u*::*nat* ⇒ *nat*, *y*::*nat* ⇒ *nat*), *x*::*nat* ⇒ *unit*). ((λ*i*::*nat*. *if i* = (*0*::*nat*) *then 2*::*nat else u* (*i* − (*1*::*nat*)) + (*3*::*nat*), λ*i*::*nat*. *if i* = (*0*::*nat*) *then 2*::*nat else u* (*i* − (*1*::*nat*)) + (*3*::*nat*)), *x*))

lemma *Adder3-iter-aux-a*: ⋀ *a b c* . (*Adder3-iter-fun* ^^ (*n*::*nat*)) *z* = ((*a*,*b*), *c*) ⟹ (∀ *i* < *n* . *a i* = *3* ∗ *i* + *2* ∧ *b i* = *3* ∗ *i* + *2*) ∧ *c* = (*snd z*)

lemma *Adder3-iter-aux-b*[*simp*]: *i* < *n* ⟹ *apply* ((*Adder3-iter-fun* ^^ *n*) *z*) *i* = *apply* ((*Adder3-iter-fun* ^^ *Suc i*) *z*) *i*

lemma *Adder3-iter-aux-c*: (λ*x*. *let z* = λ*i*. *apply* (*Adder3-iter-fun* ((*Adder3-iter-fun* ^^ *i*) *x*)) *i in* ((*fst* ∘ *fst* ∘ *z*, *snd* ∘ *fst* ∘ *z*), *snd* ∘ *z*)) = (λ *x* . (((λ *i* . *3* ∗ *i* + *2*), (λ *i* . *3* ∗ *i* + *2*)), *snd x*))

lemma *FeedbackX Init-adder3 S-adder3* = *Res-adder3*

definition *Init-sum* = [−λ*x*. (λ (*i*::*nat*). (*0*::*nat*))−]
definition *S-sum* = [−λ(*x*, *x′*). (λ*i*. *x i* + *x′ i*) −] *o* [− λ*x* . (λ (*i*::*nat*). *if i* = *0 then* (*0*::*nat*) *else x* (*i*−*1*)) −] *o* [− λ*x*. (*x*, *x*) −]
definition *Res-sum* = [−λ*x*. *Summ x*−]

definition *S-simp-sum* = [−λ(*x*, *x′*). ((λ*i*. *if i* = *0 then 0 else x* (*i*−*1*) + *x′* (*i*−*1*)),(λ*i*. *if i* = *0 then 0 else x* (*i*−*1*) + *x′* (*i*−*1*))) −]

lemma *S-sum-simp*: *S-sum* = *S-simp-sum*

lemma *Sum-inner-simp*: [−λ((*u*, *y*), *x*). ((*u*, *x*), *x*)−] ∘ *S-simp-sum* ∗∗ *Skip* = [− λ((*u*, *y*), *x*). (((λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*)),(λ*i*. *if i* = *0 then 0 else u* (*i*−*1*) + *x* (*i*−*1*))), *x*) −]

definition *Sum-iter-fun* = (λ((*u*::*nat*⇒*nat*, *y*::*nat*⇒*nat*), *x*::*nat*⇒*nat*). (((λ(*i*::*nat*). *if i* = (*0*::*nat*) *then* (*0*::*nat*) *else u* (*i*−(*1*::*nat*)) + *x* (*i*−(*1*::*nat*))), (λ(*i*::*nat*). *if i* = (*0*::*nat*) *then* (*0*::*nat*) *else u* (*i*−(*1*::*nat*)) + *x* (*i*−(*1*::*nat*)))), *x*))

lemma *Sum-iter-aux-a*: ⋀ *a b c* . (*Sum-iter-fun* ^^ (*n*::*nat*)) *z* = ((*a*,*b*), *c*) ⟹ (∀ *i* < *n* . *a i* = (*Summ c* (*i*::*nat*)) ∧ *b i* = (*Summ c* (*i*::*nat*))) ∧ *c* = (*snd z*)

lemma *Sum-iter-aux-b*: (*i* < *n* ⟹ *apply* (((*Sum-iter-fun*) ^^ *n*) *z*) *i* = *apply* ((*Sum-iter-fun* ^^ (*Suc i*)) *z*) *i*)

lemma *Sum-iter-aux-c*: $(\lambda x.\ let\ z = \lambda i.\ apply\ (Sum\text{-}iter\text{-}fun\ ((Sum\text{-}iter\text{-}fun\ \hat{}\hat{}\ i)\ x))\ i\ in\ ((fst \circ fst \circ z,\ snd \circ fst \circ z),\ snd \circ z))$
$= (\lambda\ x\ .\ ((Summ\ (snd\ x),\ Summ\ (snd\ x)),\ snd\ x)\ )$

lemma *FeedbackX Init-sum S-sum = Res-sum*

definition *Init-adder3-wp* $= [-\ \lambda x.\ (\lambda\ (i{::}nat).\ (2{::}nat))-]$
definition *S-adder3-wp* $= [-\ \lambda\ (x,\ (x'{::}nat \Rightarrow unit))\ .\ x-]\ o\ \{.\Box\ (\lambda x.\ x\ 0 \neq 0).\}\ o\ [-\ \lambda x\ .\ (\lambda\ (i{::}nat).\ (x\ i) + 1)-]\ o\ [-\ \lambda x\ .\ (\lambda\ (i{::}nat).\ if\ i = 0\ then\ (0{::}nat)\ else\ x\ (i-1))\ -]\ o\ [-\ \lambda x.\ (\lambda\ (i{::}nat)\ .\ x\ i + 2)\ -]\ o\ [-\ \lambda x.\ (x,\ x)\ -]$
definition *Res-adder3-wp* $= \{.\ x.\ True.\}\ o\ [-\ \lambda x\ .\ (\lambda\ (i{::}nat)\ .\ 3 * i + 2)\ -]$

definition *S-simp-adder3-wp* $= \{.\ \Box\ (\lambda\ (x,\ (x'{::}nat \Rightarrow unit)).\ x\ 0 \neq 0)\ .\}\ o[-\ \lambda\ (x,\ \ (x'{::}nat \Rightarrow unit)).\ ((\lambda i.\ if\ i = 0\ then\ 2\ else\ x(i-1) + 3),\ (\lambda i.\ if\ i = 0\ then\ 2\ else\ x(i-1) + 3))-]$

lemma *S-adder3-wp-simp*: *S-adder3-wp = S-simp-adder3-wp*

lemma *Adder3-wp-inner-simp*: $[-\lambda((u,\ y),\ x).\ ((u,\ x),\ x)-] \circ$ *S-simp-adder3-wp* $**$ *Skip* $= \{.\ \Box\ (\lambda\ ((u,\ y),\ x).\ u\ 0 \neq 0).\}\ o\ [-\lambda((u,\ y),\ x).\ (((\lambda i.\ if\ i = 0\ then\ 2\ else\ u(i-1) + 3),\ (\lambda i.\ if\ i = 0\ then\ 2\ else\ u(i-1) + 3)),\ x)-]$

definition *Adder3-iter-wp-fun* $= (\lambda((u{::}nat \Rightarrow nat,\ y{::}nat \Rightarrow nat),\ x{::}nat \Rightarrow unit).\ ((\lambda i{::}nat.\ if\ i = (0{::}nat)\ then\ 2{::}nat\ else\ u\ (i - (1{::}nat)) + (3{::}nat),\ \lambda i{::}nat.\ if\ i = (0{::}nat)\ then\ 2{::}nat\ else\ u\ (i - (1{::}nat)) + (3{::}nat)),\ x))$
definition *Adder3-iter-wp-prec* $= (\Box\ (\lambda\ ((u,\ y),\ x).\ u\ 0 \neq 0))$

lemma *Adder3-iter-wp-aux-a*: $\bigwedge\ a\ b\ c\ .\ (Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ (n{::}nat))\ z = ((a,b),\ c) \Longrightarrow (\forall\ i < n\ .\ a\ i = 3 * i + 2 \wedge b\ i = 3 * i + 2) \wedge c = (snd\ z)$

lemma *Adder3-iter-wp-aux-b*: $i < n \Longrightarrow apply\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ n)\ z)\ i = apply\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ Suc\ i)\ z)\ i$

lemma *Adder3-iter-wp-aux-c*: $(\lambda x.\ let\ z = \lambda i.\ apply\ (Adder3\text{-}iter\text{-}wp\text{-}fun\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ i)\ x))\ i\ in\ ((fst \circ fst \circ z,\ snd \circ fst \circ z),\ snd \circ z)) = (\lambda\ x\ .\ (((\lambda\ i\ .\ 3 * i + 2),\ (\lambda\ i\ .\ 3 * i + 2)),\ snd\ x)\ )$

lemma *Adder3-iter-wp-aux-d*: $\bigwedge\ i\ .\ i \geq n \Longrightarrow fst\ (fst\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ n)\ ((\lambda i.\ 2,\ b),\ ba)))\ i = 3 * n + 2$

lemma *Adder3-iter-wp-aux-e*: $\bigwedge\ i.\ i < n \Longrightarrow fst\ (fst\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ n)\ ((\lambda i.\ 2,\ b),\ ba)))\ i = 3 * i + 2$

lemma *Adder3-iter-wp-prec-aux*: $0 < fst\ (fst\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ n)\ ((\lambda i.\ 2,\ b),\ ba)))\ i$

lemma *Adder3-iter-wp-prec*: $(\Box\ (\lambda((u,\ y),\ x).\ 0 < u\ 0))\ ((Adder3\text{-}iter\text{-}wp\text{-}fun\ \hat{}\hat{}\ n)\ ((\lambda i.\ 2,\ b),\ ba))$

lemma *FeedbackX Init-adder3-wp S-adder3-wp = Res-adder3-wp*

definition *Init-adder3-havoc* = [−λx. (λi. 0)−]
definition *Res-adder3-havoc* = ⊥

lemma [*simp*]: (λx. ∀ b ba n. (□ (λ((u, y), x). 0 < u 0)) ((*Adder3-iter-wp-fun* ^^ n) ((λi. 0, b), ba))) = ⊥

lemma [*simp*]: {.λx. *False*.} o [:r:] = ⊥

lemma *FeedbackX Init-adder3-havoc S-adder3-wp = Res-adder3-havoc*

lemma *Feedback-ExFb*: *FeedbackX Init-ExFb ExFb-delayfb = Res*

lemma *feedback-in-simp-aaa*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow$
feedback ({. u, (s,x) . p′ (s,x) ∧ p (u, (s,x)).} o [:u, (s,x) ⇝ v, (s′,y) . r′ (s,x) v ∧ r (u, (s,x)) (s′,y):])
= {. (s,x) . p′ (s,x) ∧ (∀ b. r′ (s,x) b ⟶ p (b,(s,x))).} o [:(s,x) ⇝ (s′,y) . ∃ v . r′ (s,x) v ∧ r (v, (s,x)) (s′,y):]

lemma *IterateOmega-spec-a*: *IterateOmega* ({. p .} ∘ [: r :]) = {.((u,y),x) .∀ n v y′ z. (r ^^ n) ((u,y), x) ((v, y′), z) ⟶ p ((v,y′),z).} ∘ [: *INF n. r ^^ n OO eqtop n* :]

lemma *AAA*: ⋀u′ y′ . (((λ((u::′a, y::′b), x) ((u′::′a, y′::′b), x′). r (u, x) (u′, y′) ∧ x = x′) ^^ n) ((u,y::′b), x) ((u′, y′::′b), x′)) ⟹ x = x′

lemma *BBB*: ⋀u′ y′ . (((λ((u::′a, y::′b), x) ((u′::′a, y′::′b), x′). r (u, x) (u′, y′) ∧ x = x′) ^^ n) ((u,y::′b), x) ((u′, y′::′b), x′)) =
(x = x′ ∧ (((λ (u::′a, y::′b) (u′::′a, y′::′b) . r (u, x) (u′, y′)) ^^ n) (u,y::′b) (u′, y′::′b)))

lemma *CCC*: (((λ((u::′a, y), x) ((u′::′a, y′), x′). r (u, x) (u′, y′) ∧ x = x′) ^^ n) ((u,y::′b), x) ((u′, y′::′b), x′)) =
(x = x′ ∧ (∃ U Y . U 0 = u ∧ U n = u′ ∧ Y 0 = y ∧ Y n = y′ ∧ (∀ i < n . r (U i, x) (U (*Suc* i), Y (*Suc* i)))))

lemma *IterateOmegaA-simp-a*: *IterateOmegaA* ([−λ ((u, y::*nat* ⇒′a), x) . ((u, x), x)−] o (({.p.} o [:r:]) ∗∗ *Skip*)) =
{.((ua, ya), xa).∀ n a. (∃ b U. U 0 = ua ∧ U n = a ∧ (∃ Y. Y 0 = ya ∧ Y n = b ∧ (∀ i<n. r (U i, xa) (U (*Suc* i), Y (*Suc* i))))) ⟶ p (a, xa).} ∘
[: *INF n.* (λ((u, y), x) ((u′, y′), x′). r (u, x) (u′, y′) ∧ x = x′) ^^ n *OO eqtop* (n−1) :]

lemma *IterateOmegaA-simp-b*: *IterateOmegaA* ([−λ ((u, y::*nat* ⇒′a), x) . ((u, x), x)−] o (({.p.} o [:r:]) ∗∗ *Skip*)) =
{.((ua, ya), xa).∀ n U Y . (U 0 = ua ∧ Y 0 = ya ∧ (∀ i<n. r (U i, xa) (U (*Suc* i), Y (*Suc* i))))

$\longrightarrow p\ (U\ n,\ xa).\}\ \circ$
$[:\ INF\ n.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ r\ (u,\ x)\ (u',\ y') \wedge x = x')\ \hat{}\hat{}\ n\ OO\ eqtop\ (n-1)\ :]$

lemma *IterateOmegaA-simp-aux*: $(INF\ n.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ r\ (u,\ x)\ (u',\ y') \wedge x = x')\ \hat{}\hat{}\ n\ OO\ eqtop\ (n-1))\ ((u{::}nat \Rightarrow 'a, y{::}nat \Rightarrow 'b),\ x{::}nat \Rightarrow 'c)\ \ ((u'{::}nat \Rightarrow 'a, y'{::}nat \Rightarrow 'b),\ x'{::}nat \Rightarrow 'c) =$
$(x = x' \wedge (\forall xa.\ \exists a\ b.\ (\exists U.\ U\ 0 = u \wedge U\ xa = a \wedge (\exists Y.\ Y\ 0 = y \wedge Y\ xa = b \wedge (\forall i{<}xa.\ r\ (U\ i,\ x)\ (U\ (Suc\ i),\ Y\ (Suc\ i))))))) \wedge (\forall i{<}xa{-}1.\ a\ i = u'\ i) \wedge (\forall i{<}xa{-}1.\ b\ i = y'\ i)))$

lemma *IterateOmegaA-simp-c*: *IterateOmegaA* $([-\lambda\ ((u{::}nat \Rightarrow 'a,\ y{::}nat \Rightarrow 'b),\ x{::}nat \Rightarrow 'c)\ .\ ((u,\ x),\ x)-]\ o\ (([.p.\}\ o\ [:r:])\ **\ Skip)) =$
$\{.((ua,\ ya),\ xa).\forall n\ U\ Y\ .\ (U\ 0 = ua \wedge Y\ 0 = ya \wedge (\forall i{<}n.\ r\ (U\ i,\ xa)\ (U\ (Suc\ i),\ Y\ (Suc\ i))))$
$\longrightarrow p\ (U\ n,\ xa).\}\ \circ$
$[:\ (u,\ y),\ x \rightsquigarrow (u'{::}nat \Rightarrow 'a,\ y'{::}nat \Rightarrow 'b),\ x'{::}nat \Rightarrow 'c\ .\ x = x'$
$\wedge (\forall xa.\ \exists a\ b.\ (\exists U.\ U\ 0 = u \wedge U\ xa = a \wedge (\exists Y.\ Y\ 0 = y \wedge Y\ xa = b \wedge (\forall i{<}xa.\ r\ (U\ i,\ x)\ (U\ (Suc\ i),\ Y\ (Suc\ i)))))) \wedge (\forall i{<}xa{-}1.\ a\ i = u'\ i) \wedge (\forall i{<}xa{-}1.\ b\ i = y'\ i))\ :]$

lemma *IterateOmegaA-simp-d*: *IterateOmegaA* $([-\lambda\ ((u{::}nat \Rightarrow 'a,\ y{::}nat \Rightarrow 'b),\ x{::}nat \Rightarrow 'c)\ .\ ((u,\ x),\ x)-]\ o\ (([.p.\}\ o\ [:r:])\ **\ Skip)) =$
$\{.((ua,\ ya),\ xa).\forall n\ U\ Y\ .\ (U\ 0 = ua \wedge Y\ 0 = ya \wedge (\forall i{<}n.\ r\ (U\ i,\ xa)\ (U\ (Suc\ i),\ Y\ (Suc\ i))))$
$\longrightarrow p\ (U\ n,\ xa).\}\ \circ$
$[:\ (u,\ y),\ x \rightsquigarrow (u'{::}nat \Rightarrow 'a,\ y'{::}nat \Rightarrow 'b),\ x'{::}nat \Rightarrow 'c\ .\ x = x'$
$\wedge (\forall xa.\ (\exists U.\ U\ 0 = u \wedge (\exists Y.\ Y\ 0 = y \wedge (\forall i{<}xa.\ r\ (U\ i,\ x)\ (U\ (Suc\ i),\ Y\ (Suc\ i))) \wedge (\forall i{<}xa{-}1.\ U\ xa\ i = u'\ i) \wedge (\forall i{<}xa{-}1.\ Y\ xa\ i = y'\ i))))\ :]$

lemma *DelayFeedback-feedback-simp*: *DelayFeedback init* $(feedback\ (\{.(u,\ s,\ x).\ \ p\ u\ s\ x.\} \circ [-\lambda(u,\ s,\ x).\ (f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x)-]))) =$
$\{.prec\text{-}pre\text{-}sts\ init\ (\lambda(s,\ x)\ .\ p\ (f\ s\ x)\ s\ x)\ (\lambda(s,\ x)\ y\ .\ y = (g\ (f\ s\ x)\ s\ x,\ h\ (f\ s\ x)\ s\ x)).\} \circ$
$[:rel\text{-}pre\text{-}sts\ init\ (\lambda(s,\ x)\ y.\ y = (g\ (f\ s\ x)\ s\ x,\ h\ (f\ s\ x)\ s\ x)):]$

lemma *input-output-switch*: $([-\lambda(s,\ u,\ x).\ (u,\ s,\ x)-] \circ (\{.\ p\ .\} \circ [-\lambda(u,\ s,\ x).\ (f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x)-]) \circ [-\lambda(v,\ s,\ y).\ (s,\ v,\ y)-]) =$
$\{.\ (s,u,x).\ p\ (u,\ s,\ x)\ .\}\ o\ [-\ \lambda(s,\ u,\ x).(g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x)\ -]$

primrec $ss :: 'a \Rightarrow ('b \Rightarrow 'a \Rightarrow 'c \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'c \Rightarrow 'b) \Rightarrow (nat \Rightarrow 'c) \Rightarrow nat \Rightarrow 'a$ **where**
$ss\ a\ g\ f\ xa\ 0 = a\ |$
$ss\ a\ g\ f\ xa\ (Suc\ i) = g\ (f\ (ss\ a\ g\ f\ xa\ i)\ (xa\ i))\ (ss\ a\ g\ f\ xa\ i)\ (xa\ i)$

primrec $ssu :: 'a \Rightarrow ('b \Rightarrow 'a \Rightarrow 'c \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'b) \Rightarrow (nat \Rightarrow 'c) \Rightarrow nat \Rightarrow 'a$ **where**
$ssu\ a\ g\ u\ x\ 0 = a\ |$
$ssu\ a\ g\ u\ x\ (Suc\ i) = g\ (u\ i)\ (ssu\ a\ g\ u\ x\ i)\ (x\ i)$

lemma *BBBd*: $a = sa\ 0 \Longrightarrow \forall fb{<}fa.\ sa\ (Suc\ fb) = g\ (u\ fb)\ (sa\ fb)\ (x\ fb)\ \Longrightarrow i \leq fa \Longrightarrow ssu\ a\ g\ u\ x\ i = sa\ i$

definition *prec-pre-sts-st init p r u x* $= (\forall\ y\ .\ init\ (u\ 0) \longrightarrow (lift\text{-}rel\ r\ leads\ lift\text{-}pre\ p)\ (u,\ x)\ (u[1..],\ y))$

lemma *prec-pre-sts-st-simp*: *prec-pre-sts-st init p r u x* $=$
$(\forall\ y\ .\ init\ (u\ 0) \longrightarrow (\forall n\ .\ (\forall\ i < n\ .\ r\ (u\ i,\ x\ i)\ (u\ (Suc\ i),\ y\ i)) \longrightarrow p\ (u\ n,\ x\ n)))$

lemma *BBBc*: $s = ssu\ a\ g\ u\ x \Longrightarrow prec\text{-}pre\text{-}sts\ (\lambda\ s\ .\ s = a)\ (\lambda(s,\ u,\ x).\ p\ (u,\ s,\ x))\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (\lambda i.\ (u\ i,\ x\ i)) =$
$((\forall\ fa.\ (\forall\ fb{<}fa.\ s\ (Suc\ fb) = g\ (u\ fb)\ (s\ fb)\ (x\ fb)) \longrightarrow p\ (u\ fa,\ s\ fa,\ x\ fa)))$

lemma *BBBx*: $s = ssu\ a\ g\ u\ x \Longrightarrow$ *prec-pre-sts* $(\lambda\ s\ .\ s = a)\ (\lambda(s,\ u,\ x).\ p\ (u,\ s,\ x))\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (\lambda i.\ (u\ i,\ x\ i)) =$
$((\forall\ fa.\ p\ (u\ fa,\ s\ fa,\ x\ fa)))$

lemma *BBBy*: $(prec\text{-}pre\text{-}sts\ (\lambda\ s\ .\ s = a)\ (\lambda(s,\ u,\ x).\ p\ (u,\ s,\ x))\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (\lambda i.\ (u\ i,\ x\ i))) =$
$((\forall\ fa.\ p\ (u\ fa,\ \ ssu\ a\ g\ u\ x\ fa,\ x\ fa)))$

lemmas *BBBu* = *BBBd* [*of* - - - - $(\lambda\ i\ .\ f\ (s\ i)\ (x\ i))$]

lemma *BBBe*: $a = sa\ 0 \Longrightarrow \forall fb{<}fa.\ sa\ (Suc\ fb) = g\ (f\ (sa\ fb)\ (x\ fb))\ (sa\ fb)\ (x\ fb) \Longrightarrow i \leq fa \Longrightarrow ss\ a\ g\ f\ x\ i = sa\ i$

lemma *BBBz*: $(prec\text{-}pre\text{-}sts\ (\lambda\ s\ .\ s = a)\ (\lambda(s,\ x).\ p\ (f\ s\ x,\ s,\ x))\ (\lambda(s,\ x)\ y.\ y = (g\ (f\ s\ x)\ s\ x,\ h\ (f\ s\ x)\ s\ x))\ x)$
$= ((\forall\ fa.\ p\ (f\ (\ ss\ a\ g\ f\ x\ fa)\ (x\ fa),\ \ ss\ a\ g\ f\ x\ fa,\ x\ fa)))$

primrec $ssc :: 'c \Rightarrow (nat \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'c \Rightarrow 'de \Rightarrow 'c)\ \Rightarrow (nat \Rightarrow 'de) \Rightarrow nat \Rightarrow nat \Rightarrow 'c$ **where**
$ssc\ a\ U\ g\ xa\ i\ 0 = a\ |$
$ssc\ a\ U\ g\ xa\ i\ (Suc\ fa) = \ g\ (U\ fa)\ (ssc\ a\ U\ g\ xa\ i\ fa)\ (xa\ fa)$

primrec $UUc :: (nat \Rightarrow 'a) \Rightarrow 'b \Rightarrow ('b \Rightarrow 'c \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'b) \Rightarrow (nat \Rightarrow 'c) \Rightarrow nat \Rightarrow nat \Rightarrow 'a$ **where**
$UUc\ u\ a\ f\ g\ x\ 0 = u\ |$
$UUc\ u\ a\ f\ g\ x\ (Suc\ i) = (\lambda\ xa\ .\ f\ (ssc\ a\ (UUc\ u\ a\ f\ g\ x\ i)\ g\ x\ i\ xa)\ (x\ xa))$

lemma *DDDa*: $\forall fa.\ sa\ (Suc\ fa) = g\ (U\ i\ fa)\ (sa\ fa)\ (xa\ fa) \wedge U\ (Suc\ i)\ fa = f\ (sa\ fa)\ (xa\ fa) \wedge Y\ (Suc\ i)\ fa = h\ (U\ i\ fa)\ (sa\ fa)\ (xa\ fa) \Longrightarrow$
$a = sa\ 0 \Longrightarrow sa\ k = ssc\ a\ (U\ i)\ g\ xa\ i\ k$
lemma *AAAAU*:$U\ 0 = ua \Longrightarrow aa = U\ n \Longrightarrow \forall i{<}n.\ \forall fa.\ U\ (Suc\ i)\ fa = f\ (ssc\ a\ (U\ i)\ g\ xa\ i\ fa)\ (xa\ fa) \Longrightarrow k \leq n \Longrightarrow UUc\ (U\ 0)\ a\ f\ g\ xa\ k = U\ k$

lemma *AAAAAka*:$0 < n \Longrightarrow\ (\exists b\ U.\ (n = 0 \longrightarrow U\ 0 = ua \wedge U\ 0 = aa \wedge ya = b) \wedge$
$(0 < n \longrightarrow U\ 0 = ua \wedge U\ n = aa \wedge (\forall i{<}n.\ \forall fa.\ U\ (Suc\ i)\ fa = f\ (ssc\ a\ (U\ i)\ g\ xa\ i\ fa)\ (xa\ fa)) \wedge$
$(\forall fa.\ h\ (U\ (n - Suc\ 0)\ fa)\ (ssc\ a\ (U\ (n - Suc\ 0))\ g\ xa\ (n - Suc\ 0)\ fa)\ (xa\ fa) = b\ fa)))$
$= (UUc\ ua\ a\ f\ g\ xa\ n = aa)$

lemma *AAAAAk*: $(\exists b\ U.\ (n = 0 \longrightarrow U\ 0 = ua \wedge U\ 0 = aa \wedge ya = b) \wedge$
$(0 < n \longrightarrow U\ 0 = ua \wedge U\ n = aa \wedge (\forall i{<}n.\ \forall fa.\ U\ (Suc\ i)\ fa = f\ (ssc\ a\ (U\ i)\ g\ xa\ i\ fa)\ (xa\ fa))$
$\wedge (\forall fa.\ h\ (U\ (n - Suc\ 0)\ fa)\ (ssc\ a\ (U\ (n - Suc\ 0))\ g\ xa\ (n - Suc\ 0)\ fa)\ (xa\ fa) = b\ fa)))$
$= (UUc\ ua\ a\ f\ g\ xa\ n = aa)$

lemma *ZZZp*: $\forall xa{::}nat.\ sa\ (Suc\ xa) = g\ (U\ i\ xa)\ (sa\ xa)\ (x\ xa) \wedge U\ (Suc\ i)\ xa = f\ (sa\ xa)\ (x\ xa) \wedge Y\ (Suc\ i)\ xa = h\ (U\ i\ xa)\ (sa\ xa)\ (x\ xa) \Longrightarrow a = sa\ (0{::}nat) \Longrightarrow sa\ k = ssc\ a\ (U\ i)\ g\ x\ i\ k$

lemma *ZZZq*: $s = ssc\ a\ (U\ i)\ g\ x\ i \Longrightarrow\ (\exists\, s.\ s\ 0 = a \wedge (\forall\, xa.\ s\ (Suc\ xa) = g\ (U\ i\ xa)\ (s\ xa)\ (x\ xa)$
$\wedge\ U\ (Suc\ i)\ xa = f\ (s\ xa)\ (x\ xa) \wedge Y\ (Suc\ i)\ xa = h\ (U\ i\ xa)\ (s\ xa)\ (x\ xa))) =$
$(\forall\ xa\ .\ U\ (Suc\ i)\ xa = f\ (s\ xa)\ (x\ xa) \wedge Y\ (Suc\ i)\ xa = h\ (U\ i\ xa)\ (s\ xa)\ (x\ xa))$

lemma *ZZZr*: $0 < xa \Longrightarrow(\exists\, Y\ .\ Y\ 0 = y \wedge Y\ xa = b \wedge (\forall\, i{<}xa.\ \forall\, xa{::}nat.\ U\ (Suc\ i)\ xa = f\ (ssc\ a$
$(U\ i)\ g\ x\ i\ xa)\ (x\ xa) \wedge Y\ (Suc\ i)\ xa = h\ (U\ i\ xa)\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa)))$
$= ((\forall\, i{<}xa.\ \forall\, xa{::}nat.\ U\ (Suc\ i)\ xa = f\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge (\forall\ k\ .\ h\ (U\ (xa\ {-}1)\ k)$
$(ssc\ a\ (U\ (xa\ {-}1))\ g\ x\ (xa\ {-}1)\ k)\ (x\ k) = b\ k))$

lemma *ZZZc*: $(\exists\, Y\ .\ Y\ 0 = y \wedge Y\ xa = b \wedge (\forall\, i{<}xa.\ \forall\, xa{::}nat.\ U\ (Suc\ i)\ xa = f\ (ssc\ a\ (U\ i)\ g\ x$
$i\ xa)\ (x\ xa) \wedge Y\ (Suc\ i)\ xa = h\ (U\ i\ xa)\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa))) =$
$(if\ xa = 0\ then\ y = b\ else\ ((\forall\, i{<}xa.\ \forall\, xa{::}nat.\ U\ (Suc\ i)\ xa = f\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge$
$(\forall\ k\ .\ h\ (U\ (xa\ {-}1)\ k)\ (ssc\ a\ (U\ (xa\ {-}1))\ g\ x\ (xa\ {-}1)\ k)\ (x\ k) = b\ k)))$

lemma [*simp*]: $\forall\, i{<}xa.\ \forall\, xa{::}nat.\ Ua\ (Suc\ i)\ xa = f\ (ssc\ a\ (Ua\ i)\ g\ x\ i\ xa)\ (x\ xa) \Longrightarrow UUc\ (Ua$
$(0{::}nat))\ a\ f\ g\ x\ xa = Ua\ xa$

lemma *TTTb*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (0 < xa \longrightarrow (\exists\ U\ .\ U\ 0 = u \wedge U\ xa = aa \wedge (\forall\, i{<}xa.\ \forall\, xa.$
$U\ (Suc\ i)\ xa = f\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge$
$(\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x$
$k) = b\ k)))$
$= (0 < xa \longrightarrow (U\ xa = aa \wedge (\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa -$
$Suc\ 0)\ k)\ (x\ k) = b\ k)))$

lemma *TTTa*: $(\exists\, U.\ (xa = 0 \longrightarrow U\ 0 = u \wedge U\ 0 = aa \wedge y = b) \wedge$
$(0 < xa \longrightarrow U\ 0 = u \wedge U\ xa = aa \wedge (\forall\, i{<}xa.\ \forall\, xa.\ U\ (Suc\ i)\ xa = f\ (ssc\ a$
$(U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge (\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x$
$k) = b\ k)))$
$=((xa = 0 \longrightarrow u = aa \wedge y = b) \wedge$
$(0 < xa \longrightarrow (\exists\ U\ .\ U\ 0 = u \wedge U\ xa = aa \wedge (\forall\, i{<}xa.\ \forall\, xa.\ U\ (Suc\ i)\ xa =$
$f\ (ssc\ a\ (U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge$
$(\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x$
$k) = b\ k))))$

lemma *TTTc*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (\exists\, U.\ (xa = 0 \longrightarrow U\ 0 = u \wedge U\ 0 = aa \wedge y = b) \wedge$
$(0 < xa \longrightarrow U\ 0 = u \wedge U\ xa = aa \wedge (\forall\, i{<}xa.\ \forall\, xa.\ U\ (Suc\ i)\ xa = f\ (ssc\ a$
$(U\ i)\ g\ x\ i\ xa)\ (x\ xa)) \wedge (\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x$
$k) = b\ k)))$

$=((xa = 0 \longrightarrow u = aa \wedge y = b) \wedge (0 < xa \longrightarrow\ (U\ xa = aa \wedge (\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc$
$a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = b\ k))))$

lemma *TTTe*: $(\exists\ b.\ ((xa = 0 \longrightarrow u = aa \wedge y = b) \wedge (0 < xa \longrightarrow\ (U\ xa = aa \wedge (\forall\, k.\ h\ (U\ (xa$
$- Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = b\ k)))) \wedge$
$(\forall\, i{<}xa - Suc\ 0.\ aa\ i = u'\ i) \wedge (\forall\, i{<}xa - Suc\ 0.\ b\ i = y'\ i))$
$= (((xa = 0 \longrightarrow u = aa) \wedge (0 < xa \longrightarrow\ ((U\ xa = aa \wedge (\exists\ b\ .\ (\forall\, k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a$
$(U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = b\ k) \wedge$
$(\forall\, i{<}xa - Suc\ 0.\ aa\ i = u'\ i) \wedge (\forall\, i{<}xa - Suc\ 0.\ b\ i = y'\ i)))))))$

lemma *TTTf*: $(\exists\ b.\ ((xa = 0 \longrightarrow u = aa \wedge y = b) \wedge (0 < xa \longrightarrow\ (U\ xa = aa \wedge (\forall\, k.\ h\ (U\ (xa$

$-$ *Suc 0*) *k*) (*ssc a* (*U* (*xa* $-$ *Suc 0*)) *g x* (*xa* $-$ *Suc 0*) *k*) (*x k*) = *b k*)))) $\wedge$
($\forall$ *i*<*xa* $-$ *Suc 0*. *aa i* = *u' i*) $\wedge$ ($\forall$ *i*<*xa* $-$ *Suc 0*. *b i* = *y' i*))
= (((*xa* = *0* $\longrightarrow$ *u* = *aa*) $\wedge$ (*0* < *xa* $\longrightarrow$ ((*U xa* = *aa* $\wedge$
($\forall$ *i*<*xa* $-$ *Suc 0*. *aa i* = *u' i*) $\wedge$ ($\forall$ *k*<*xa* $-$ *Suc 0*. *h* (*U* (*xa* $-$ *Suc 0*) *k*) (*ssc a* (*U* (*xa* $-$ *Suc 0*)) *g x* (*xa* $-$ *Suc 0*) *k*) (*x k*) = *y' k*))))))

thm *UUc.simps*

thm *ssc.simps*

primrec *SS*::'*b* $\Rightarrow$ ('*a* $\Rightarrow$ '*b* $\Rightarrow$ '*c* $\Rightarrow$ '*b*) $\Rightarrow$ ('*b* $\Rightarrow$ '*c* $\Rightarrow$ '*a*) $\Rightarrow$ (*nat* $\Rightarrow$ '*c*) $\Rightarrow$ *nat* $\Rightarrow$ '*b* **where**
SS a g f x 0 = *a* |
SS a g f x (*Suc i*) = *g* (*f* (*SS a g f x i*) (*x i*)) (*SS a g f x i*) (*x i*)

lemma *UU-SS*: $\bigwedge$ *xa* . *i* < *xa* $\Longrightarrow$ *UUc u a f g x xa i* = *f* (*SS a g f x i*) (*x i*) $\wedge$ *ssc a* (*UUc u a f g x xa*) *g x xa i* = *SS a g f x i*

lemma *TTTza*: (*x* = *x'* $\wedge$ ($\forall$ *xa*>*0*::*nat*. ($\forall$ *i*<*xa* $-$ *Suc* (*0*::*nat*). *f* (*SS a g f x i*) (*x i*) = *u' i*) $\wedge$ ($\forall$ *k*<*xa* $-$ *Suc* (*0*::*nat*). *h* (*f* (*SS a g f x k*) (*x k*)) (*SS a g f x k*) (*x k*) = *y' k*))) =
(*x* = *x'* $\wedge$ ($\forall$ *k* . *f* (*SS a g f x k*) (*x k*) = *u' k*) $\wedge$ ($\forall$ *k* . *h* (*f* (*SS a g f x k*) (*x k*)) (*SS a g f x k*) (*x k*) = *y' k*))

lemma *AAAAta*:*0* < *n* $\Longrightarrow$ *s* = (λ *i* . *ssc a* (*U i*) *g xa i*) $\Longrightarrow$
($\exists$ *Y*. *Y 0* = *ya* $\wedge$ *Y n* = *b* $\wedge$ ($\forall$ *i*<*n*. *rel-pre-sts* (λ*b*. *b* = *a*) (λ(*s*, *u*, *x*) *y*. *y* = (*g u s x*, *f s x*, *h u s x*)) (*U i* || *xa*) (*U* (*Suc i*) || *Y* (*Suc i*)))) =
(($\forall$ *i*<*n*. $\forall$ *fa* . *U* (*Suc i*) *fa* = *f* (*s i fa*) (*xa fa*)) $\wedge$ (($\forall$ *fa* . *h* (*U* (*n* $-$ *1*) *fa*) (*s* (*n* $-$ *1*) *fa*) (*xa fa*) = *b fa*)))

lemma *AAAAt*: *s* = (λ *i* . *ssc a* (*U i*) *g xa i*) $\Longrightarrow$ ($\exists$ *Y*. *Y 0* = *ya* $\wedge$ *Y n* = *b* $\wedge$ ($\forall$ *i*<*n*. *rel-pre-sts* (λ*b*. *b* = *a*) (λ(*s*, *u*, *x*) *y*. *y* = (*g u s x*, *f s x*, *h u s x*)) (*U i* || *xa*) (*U* (*Suc i*) || *Y* (*Suc i*))))
= (*if n* = *0 then ya* = *b else* (($\forall$ *i*<*n*. $\forall$ *fa* . *U* (*Suc i*) *fa* = *f* (*s i fa*) (*xa fa*)) $\wedge$ (($\forall$ *fa* . *h* (*U* (*n* $-$ *1*) *fa*) (*s* (*n* $-$ *1*) *fa*) (*xa fa*) = *b fa*))))

lemma *BBBq*: *s* = *ssc a* (*UUc ua a f g xa n*) *g xa n* $\Longrightarrow$ ($\forall$ *s*. *s 0* = *a* $\longrightarrow$ ($\forall$ *xb*. ($\forall$ *fa*<*xb*. *s* (*Suc fa*) = *g* (*UUc ua a f g xa n fa*) (*s fa*) (*xa fa*)) $\longrightarrow$ *p* (*UUc ua a f g xa n xb*, *s xb*, *xa xb*))) =
(($\forall$ *xb*. *p* (*UUc ua a f g xa n xb*, *s xb*, *xa xb*)))

lemma *BBBk*: *prec-pre-sts* (λ*b*. *b* = *a*) (λ(*s*, *u*, *x*). *p* (*u*, *s*, *x*)) (λ(*s*, *u*, *x*) *y*. *y* = (*g u s x*, *f s x*, *h u s x*)) (*UU ua a f g xa n* || *xa*) =
($\forall$ *s* . *s 0* = *a* $\longrightarrow$ ($\forall$ *xb*. ($\forall$ *fa* < *xb*. *s* (*Suc fa*) = *g* (*UU ua a f g xa n fa*) (*s fa*) (*xa fa*)) $\longrightarrow$ *p* (*UU ua a f g xa n xb*, *s xb*, *xa xb*)))

lemma *ZZZaa*: (*INF x*. (λ((*u*, *y*), *x*) ((*u'*, *y'*), *x'*). *rel-pre-sts* (λ*b*. *b* = *a*) (λ(*s*, *u*, *x*) *y*. *y* = (*g u s x*, *f s x*, *h u s x*)) (*u* || *x*) (*u'* || *y'*) $\wedge$ *x* = *x'*) ^^ *x OO eqtop* (*x* $-$ *Suc 0*))
((*u*, (*y*::*nat* $\Rightarrow$ '*c*)), *x*) ((*u'*, *y'*::*nat* $\Rightarrow$ '*c*), *x'*) =
(*x* = *x'* $\wedge$ ($\forall$ *xa*. $\exists$ *aa b*. ($\exists$ *U*. *U 0* = *u* $\wedge$ *U xa* = *aa* $\wedge$ ($\exists$ *Y*. *Y 0* = *y* $\wedge$ *Y xa* = *b* $\wedge$ ($\forall$ *i*<*xa*. *rel-pre-sts* (λ*b*. *b* = *a*) (λ(*s*, *u*, *x*) *y*. *y* = (*g u s x*, *f s x*, *h u s x*)) (*U i* || *x*) (*U* (*Suc i*) || *Y* (*Suc i*))))))
$\wedge$
($\forall$ *i*<*xa* $-$ *Suc 0*. *aa i* = *u' i*) $\wedge$ ($\forall$ *i*<*xa* $-$ *Suc 0*. *b i* = *y' i*)))

lemma *TTTd*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') =$
$(x = x' \wedge (\forall xa.\ \exists aa\ b.\ ((xa = 0 \longrightarrow u = aa \wedge y = b) \wedge (0 < xa \longrightarrow\ (U\ xa = aa \wedge (\forall k.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = b\ k)))) \wedge$
$(\forall i{<}xa - Suc\ 0.\ aa\ i = u'\ i) \wedge (\forall i{<}xa - Suc\ 0.\ b\ i = y'\ i)))$

lemma *TTTr*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') =$
$(x = x' \wedge (\forall xa.\ \exists aa\ .\ (((xa = 0 \longrightarrow u = aa) \wedge (0 < xa \longrightarrow\ ((U\ xa = aa \wedge$
$(\forall i{<}xa - Suc\ 0.\ aa\ i = u'\ i) \wedge (\forall k{<}xa - Suc\ 0.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = y'\ k))))))))$

lemma *TTTt*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') =$
$(x = x' \wedge (\forall xa.\ (((xa = 0 \longrightarrow True) \wedge (0 < xa \longrightarrow\ ((\forall i{<}xa - Suc\ 0.\ U\ xa\ i = u'\ i) \wedge (\forall k{<}xa - Suc\ 0.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = y'\ k))))))))$

lemma *TTTy*: $U = UUc\ u\ a\ f\ g\ x \Longrightarrow (INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') =$
$(x = x' \wedge (\forall xa.\ (((0 < xa \longrightarrow\ ((\forall i{<}xa - Suc\ 0.\ U\ xa\ i = u'\ i) \wedge (\forall k{<}xa - Suc\ 0.\ h\ (U\ (xa - Suc\ 0)\ k)\ (ssc\ a\ (U\ (xa - Suc\ 0))\ g\ x\ (xa - Suc\ 0)\ k)\ (x\ k) = y'\ k))))))))$

lemma *TTTz*: $(INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') =$
$(x = x' \wedge (\forall xa{>}0::nat.\ (\forall i{<}xa - Suc\ (0::nat).\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i) = u'\ i) \wedge (\forall k{<}xa - Suc\ (0::nat).\ h\ (f\ (SS\ a\ g\ f\ x\ k)\ (x\ k))\ (SS\ a\ g\ f\ x\ k)\ (x\ k) = y'\ k)))$

lemma *TTTyt*: $(INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x') = (x = x' \wedge (\forall\ k\ .\ f\ (SS\ a\ g\ f\ x\ k)\ (x\ k) = u'\ k) \wedge (\forall k\ .\ h\ (f\ (SS\ a\ g\ f\ x\ k)\ (x\ k))\ (SS\ a\ g\ f\ x\ k)\ (x\ k) = y'\ k))$

lemma *TTT*: $(INF\ x.\ (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ rel\text{-}pre\text{-}sts\ (\lambda b.\ b = a)\ (\lambda(s,\ u,\ x)\ y.\ y = (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x))\ (u \mid\mid x)\ (u' \mid\mid y') \wedge x = x')\ \hat{}\hat{}\ x\ OO\ eqtop\ (x - Suc\ 0))$
$= (\lambda\ ((u,\ (y::nat \Rightarrow 'c)),\ x)\ ((u',\ y'::nat \Rightarrow 'c),\ x')\ .\ (x = x' \wedge (\lambda\ k\ .\ f\ (SS\ a\ g\ f\ x\ k)\ (x\ k)) = u' \wedge (\lambda\ k\ .\ h\ (f\ (SS\ a\ g\ f\ x\ k)\ (x\ k))\ (SS\ a\ g\ f\ x\ k)\ (x\ k)) = y'))$

lemma *IterateOmegaA-DelayFeedback*: $IterateOmegaA\ ([-\lambda((u,\ y),\ x).\ ((u,\ x),\ x)-] \circ ([-\lambda(x,\ y).\ x \mid\mid y-]$
$\circ\ DelayFeedback\ (\lambda x.\ x = a)\ (\{.(s,\ u,\ x).p\ (u,\ s,\ x).\} \circ [-\lambda(s,\ u,\ x).\ (g\ u\ s\ x,\ f\ s\ x,\ h\ u\ s\ x)-]) \circ [-z \rightsquigarrow fst \circ z,\ snd \circ z-])\ **\ Skip) =$

$\{.((ua,\ ya),\ xa).\forall\ n\ xb.\ p\ (UUc\ ua\ a\ f\ g\ xa\ n\ xb,\ ssc\ a\ (UUc\ ua\ a\ f\ g\ xa\ n)\ g\ xa\ n\ xb,\ xa\ xb).\}\ \circ$
$[:((u,\ y),\ x) \rightsquigarrow ((u',\ y'),\ x').x = x' \wedge (\lambda\ k\ .\ f\ (SS\ a\ g\ f\ x\ k)\ (x\ k)) = u' \wedge (\lambda\ k.\ h\ (f\ (SS\ a\ g\ f\ x\ k)\ (x\ k))\ (SS\ a\ g\ f\ x\ k)\ (x\ k)) = y' :]$

lemma *angelic-not-demonic*: $p = (r \sqcap (\lambda\ x\ uy\ .\ x = snd\ uy)) \Longrightarrow \{:x \rightsquigarrow uy\ .\ p\ x\ uy:\}\ o\ [:uy \rightsquigarrow z\ .\ q\ (snd\ uy)\ z:] = \{.x\ .\ (\exists\ u\ .\ p\ x\ (u,\ x)).\}\ o\ [:y \rightsquigarrow z\ .\ q\ y\ z:]$

lemma *SS-simp*: $\bigwedge\ xa\ .\ i < xa \Longrightarrow ssc\ a\ (\lambda i.\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i))\ g\ x\ xa\ i = SS\ a\ g\ f\ x\ i$

lemma *SS-simp-a*:$\bigwedge\ xa\ .\ xa \leq i \Longrightarrow\ u = (\lambda\ i\ .\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i)) \Longrightarrow ssc\ a\ u\ g\ x\ xa\ i = SS\ a\ g\ f\ x\ i$

lemma *SS-simp-b*: $u = (\lambda\ i\ .\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i)) \Longrightarrow ssc\ a\ u\ g\ x\ xa\ i = SS\ a\ g\ f\ x\ i$

lemma *UU-SS-simp*: $\bigwedge\ i\ .\ u = (\lambda\ i\ .\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i)) \Longrightarrow UUc\ u\ a\ f\ g\ x\ xa\ i =\ f\ (SS\ a\ g\ f\ x\ i)\ (x\ i) \wedge ssc\ a\ (UUc\ u\ a\ f\ g\ x\ xa)\ g\ x\ xa\ i = SS\ a\ g\ f\ x\ i$

declare *ssc.simps* [*simp del*]
declare *SS.simps* [*simp del*]
declare *UUc.simps* [*simp del*]

lemma *SSS*: $(\exists\ aa.\ \forall\ n\ xb.\ p\ (UUc\ aa\ a\ f\ g\ x\ n\ xb,\ ssc\ a\ (UUc\ aa\ a\ f\ g\ x\ n)\ g\ x\ n\ xb,\ x\ xb))$
$= (\forall\ n.\ p\ (f\ (SS\ a\ g\ f\ x\ n)\ (x\ n)\ ,\ SS\ a\ g\ f\ x\ n,\ x\ n))$

lemma *SSSa*: $\forall fa{<}xaa.\ s\ (Suc\ fa) = g\ (f\ (s\ fa)\ (x\ fa))\ (s\ fa)\ (x\ fa) \Longrightarrow i \leq xaa \Longrightarrow\ s\ i = SS\ (s\ 0)\ g\ f\ x\ i$

lemma *SSSb*: *prec-pre-sts* $(\lambda\ s\ .\ s = a)\ (\lambda pa.\ p\ (f\ (fst\ pa)\ (snd\ pa),\ pa))\ (\lambda p\ y.\ y = (g\ (f\ (fst\ p)\ (snd\ p))\ (fst\ p)\ (snd\ p),\ h\ (f\ (fst\ p)\ (snd\ p))\ (fst\ p)\ (snd\ p)))$
$= (\lambda\ x\ .\ \forall\ n.\ p\ (f\ (SS\ a\ g\ f\ x\ n)\ (x\ n)\ ,\ SS\ a\ g\ f\ x\ n,\ x\ n))$

lemma *SSSc*: $\forall fa.\ s\ (Suc\ fa) = g\ (f\ (s\ fa)\ (x\ fa))\ (s\ fa)\ (x\ fa) \Longrightarrow SS\ (s\ 0)\ g\ f\ x\ i = s\ i$

lemma *SSSd*: $(rel\text{-}pre\text{-}sts\ (\lambda\ s.\ s = a)\ (\lambda p\ y.\ y = (g\ (f\ (fst\ p)\ (snd\ p))\ (fst\ p)\ (snd\ p),\ h\ (f\ (fst\ p)\ (snd\ p))\ (fst\ p)\ (snd\ p))))$
$= (\lambda\ x\ y\ .\ y = (\lambda k.\ h\ (f\ (SS\ a\ g\ f\ x\ k)\ (x\ k))\ (SS\ a\ g\ f\ x\ k)\ (x\ k)))$

thm *IterateOmegaA-spec*

lemma *IterateOmegaA-update*: $IterateOmegaA\ [-f-] = [:\ INF\ n.\ (\lambda\ x\ y\ .\ f\ x = y)\ \hat{}\hat{}\ n\ OO\ eqtop\ (n - 1)\ :]$

lemma *power-example*: $(n{::}nat) > 0 \Longrightarrow$
$((\lambda((u{::}nat \Rightarrow 'a,\ y{::}nat \Rightarrow 'a),\ x{::}nat \Rightarrow 'b)\ ((u',\ y'),\ x').\ u = u' \wedge u = y' \wedge x = x')\ \hat{}\hat{}\ n)$
$= (\lambda((u,\ y),\ x)\ ((u',\ y'),\ x').\ u = u' \wedge u = y' \wedge x = x')$

lemma *power-example-a*: $(n{::}nat) > 0 \Longrightarrow$
$((\lambda((u{::}nat \Rightarrow 'a,\ y{::}nat \Rightarrow 'a),\ x{::}nat \Rightarrow 'b)\ ((u',\ y'),\ x').\ u = u' \wedge u = y' \wedge x = x')\ \hat{}\hat{}\ n)\ ((a,b),\ c)\ ((a',\ b'),\ c')$
$= (a = a' \wedge a = b' \wedge c = c')$

lemma *example-simp*: $\{:x \rightsquigarrow ((u,\ y),\ x').x = x':\} \circ [:((u,\ y),\ x) \rightsquigarrow ((u',\ y'),\ x').u = u' \wedge u = y' \wedge x = x':] \circ [-\lambda((u,\ y),\ x).\ y-] = \{:\top:\}$

lemma *Feedback-example*: $Feedback([-u::nat \Rightarrow 'a,\ x::nat \Rightarrow 'b \rightsquigarrow u,\ u-]) = \{:\top:\}$

lemma *Feedback-deterministic*: $init = (\lambda\ x\ .\ x = a) \Longrightarrow$
$DelayFeedback\ init\ (feedback(\{.(u,\ s,\ x).\ \ p\ (u,\ s,\ x).\} \circ [-\lambda(u,\ s,\ x).\ (f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x)-])) =$
$Feedback\ ([-\ u,x \rightsquigarrow\ \ u\ ||\ x\ -]\ o\ (DelayFeedback\ init\ ([-\ \lambda\ (s,(u,x))\ .\ (u,\ s,\ x)\ -]$
$o\ (\{.\ p\ .\} \circ [-\ u,\ s,\ x \rightsquigarrow f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x\ -])$
$o\ [-\ v,\ s,\ y \rightsquigarrow s,\ v,\ y\ -]))\ o\ [-\ z \rightsquigarrow fst\ o\ z,\ snd\ o\ z\ -])$

lemma *DF-fb-simp*: $init = (\lambda\ x\ .\ x = a) \Longrightarrow$
$DelayFeedback\ init\ (feedback(\{.(u,\ s,\ x).\ \ p\ (u,s,\ x).\} \circ [-\ u,\ s,\ x \rightsquigarrow f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x-])) =$
$\{.x.\forall\ n.\ p\ (f\ (SS\ a\ g\ f\ x\ n)\ (x\ n),\ SS\ a\ g\ f\ x\ n,\ x\ n).\} \circ [:y \rightsquigarrow z.\ z = (\lambda k.\ h\ (f\ (SS\ a\ g\ f\ y\ k)\ (y\ k))\ (SS\ a\ g\ f\ y\ k)\ (y\ k)):]$

lemma *DF-fb-simp-a*: $init = (\lambda\ x\ .\ x = a) \Longrightarrow$
$DelayFeedback\ init\ (feedback(\{.\ p\ .\} \circ [-\lambda(u,\ s,\ x).\ (f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x)-])) =$
$\{.x.\forall\ n.\ p\ (f\ (SS\ a\ g\ f\ x\ n)\ (x\ n),\ SS\ a\ g\ f\ x\ n,\ x\ n).\} \circ [:y \rightsquigarrow z.\ z = (\lambda k.\ h\ (f\ (SS\ a\ g\ f\ y\ k)\ (y\ k))\ (SS\ a\ g\ f\ y\ k)\ (y\ k)):]$

lemma *FB-DF-simp*: $init = (\lambda\ x\ .\ x = a) \Longrightarrow$
$Feedback\ ([-\ u,x \rightsquigarrow\ \ nzip\ u\ x\ -]\ o\ (DelayFeedback\ init\ ([-\ \lambda\ (s,(u,x))\ .\ (u,\ s,\ x)\ -]$
$o\ (\{.(u,\ s,\ x).\ \ p\ (u,\ s,\ x).\} \circ [-\ u,\ s,\ x \rightsquigarrow f\ s\ x,\ g\ u\ s\ x,\ h\ u\ s\ x-])$
$o\ [-\ v,\ s,\ y \rightsquigarrow s,\ v,y\ -]))\ o\ [-\ z \rightsquigarrow fst\ o\ z,\ snd\ o\ z\ -])$
$= \{.x.\forall\ n.\ p\ (f\ (SS\ a\ g\ f\ x\ n)\ (x\ n),\ SS\ a\ g\ f\ x\ n,\ x\ n).\} \circ [:y \rightsquigarrow z.\ z = (\lambda k.\ h\ (f\ (SS\ a\ g\ f\ y\ k)\ (y\ k))\ (SS\ a\ g\ f\ y\ k)\ (y\ k)):]$

definition *init-ex* $= (\lambda\ s\ .\ s = (0::nat))$
definition *p1* $= (\lambda(u,s,x).\ u = s + 1)$
definition *f1* $= (\lambda\ s\ x.\ s + 1)$
definition *g1* $= (\lambda\ u\ s\ x.\ s + 1)$
definition *h1* $= (\lambda\ u\ s\ x.\ x)$
definition *spec-ex* $= \{.(u,\ s,\ x).\ p1\ (u,\ s,\ x).\}\ o\ [-\lambda\ (u,\ s,\ x).\ (f1\ s\ x,\ g1\ u\ s\ x,\ h1\ u\ s\ x)-]$

lemma *DelayFeedback-feedback-ex*: $DelayFeedback\ init\text{-}ex\ (feedback\ (\ spec\text{-}ex\ )) = [:y \rightsquigarrow z.\ z = y:]$

lemma *jjj*: $[:x \rightsquigarrow ((x'',\ y),\ x').x'' = x \wedge x = x':] \circ ([:\ \lambda s.\ \Box\ (\lambda s.\ s\ 0 = b)\ :] **\ Skip) **\ Skip = [:x \rightsquigarrow ((x'',\ y),\ x').(\Box\ (\lambda s.\ s\ 0 = b))\ x'' \wedge x' = x:]$

lemma [*simp*]: $(\forall\ a.\ (\Box\ (\lambda s.\ s\ 0 = b))\ a \longrightarrow (\forall\ n\ xb.\ UUc\ a\ 0\ (\lambda s\ x.\ Suc\ s)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n\ xb = Suc\ (ssc\ 0\ (UUc\ a\ 0\ (\lambda s\ x.\ Suc\ s)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n\ xb)))$
$= ((\forall\ n\ xb.\ UUc\ (\lambda\ i\ .\ b)\ 0\ (\lambda s\ x.\ Suc\ s)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n\ xb = Suc\ (ssc\ 0\ (UUc\ (\lambda\ i\ .\ b)\ 0\ (\lambda s\ x.\ Suc\ s)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n\ xb)))$

lemma [*simp*]: $((\forall\ n\ xb.\ UUc\ (\lambda\ i\ .\ b)\ 0\ (\lambda s\ x.\ Suc\ s)\ (\lambda u\ s\ x.\ Suc\ s)\ x\ n\ xb = Suc\ (ssc\ 0\ (UUc\ (\lambda$

i . b) 0 (λs x. Suc s) (λu s x. Suc s) x n) (λu s x. Suc s) x n xb))) = False

lemma *FeedbackA-example*: *init* = (λ *s* . *s* = *b*) ⟹
FeedbackA (*InitDF init*) ([− *u*,*x* ⇝ *nzip u x* −] *o* (*DelayFeedback init-ex* ([− λ (*s*,(*u*,*x*)) . (*u*, *s*, *x*) −]
o spec-ex
o [− *v*, *s*, *y* ⇝ *s*, *v*, *y* −])) *o* [− *z* ⇝ *fst o z*, *snd o z* −]) =
⊥

definition *init-ex-a* = (λ *s* . *s* = (*0*::*nat*))
definition *p1-a* = (λ (*u*, *s*, *x*) . *u* = *s* + *1*)
definition *f1-a* = (λ *s x*. *s* + *1*)
definition *g1-a* = (λ *u s x*. *s* + *1*)
definition *h1-a* = (λ *u s x*. *x* + *s*)
definition *spec-ex-a* = {.*p1-a*.} *o* [−λ (*u*, *s*, *x*). (*f1-a s x*, *g1-a u s x*, *h1-a u s x*)−]

lemma [*simp*]: *SS 0* (λ*u s x*. *Suc s*) (λ*s x*. *Suc s*) *y k* = *k*

lemma *DelayFeedback-feedback-ex-a*: *DelayFeedback init-ex-a* (*feedback* (*spec-ex-a*)) = [:*y* ⇝ *z*. *z* = (λ*k*. *y k* + *k*):]
end

5 Overview of the Refinement Calculus of Reactive Systems (RCRS)

theory *RCRS-Overview* **imports** *ReactiveFeedback*
begin

This theory file refers to the results presented in the paper "The Refinement Calculus of Reactive Systems", by Preoteasa, Dragomir, and Tripakis, on arxiv.org, 2017, and under submission to a journal.

The section, subsection, etc., numbers and titles below refer to those in the aforementioned paper.

5.1 Section 3: Language

5.1.1 Section 3.1: An Algebra of Components

The grammar of components defined in Section 3.1 is not explicitly formalized in this theory. However, GEN_STS, STATELESS_STS, DET_STS, DET_STATELESS_STS, and QLTL components can be defined as semantic objects as they are given in Section 4.3

5.1.2 Section 3.2: Symbolic Transition System Components

5.1.3 Section 3.2.1: General STS Components

The semantics version of an STS component is given by the next definition which matches equation (6) from the paper. Another difference between the semantic sts defined here and the syntactic version from the paper is that init and r are functions in the semantic version.

definition *sts init r* = {. −*illegal-sts init* (*inpt r*) *r* .} *o* [: *x* ⇝ *y* . ∃ *s* . (*init* (*s 0*) ∧ *run-sts r s x y*) :]

definition *C1-sts* = *sts* $(\lambda\ s\ .\ s > 0)\ (\lambda\ (s,(n,x))\ (s',\ y)\ .\ s' > s \wedge y + s = x\ \hat{}\ n)$
definition *C2-sts* = *sts* $(\lambda\ s\ .\ s > 0)\ (\lambda\ (s,z)\ (s',\ y)\ .\ s' > s \wedge y + s = (snd\ z)\ \hat{}\ (fst\ z))$

lemma *C1-sts* = *C2-sts*

definition *UnitDelay* = *sts* $(\lambda\ s\ .\ s = 0)\ (\lambda\ (s,x)\ (s',y)\ .\ y = s \wedge s' = x)$

definition *Sum-sts* = *sts* $(\lambda\ s\ .\ s = (0::nat))\ (\lambda\ (s,x)\ (s',y)\ .\ y = s \wedge s' = s + x)$

definition *C-sts* = *sts* $(\lambda\ s\ .\ s = 0)\ (\lambda\ (s,\ x)\ (s',\ y)\ .\ \ x + s \leq y\ )$

definition *Div-sts* = *sts* $\top\ (\lambda\ (s::unit,(x,y))\ (s'::unit,\ z)\ .\ y \neq 0 \wedge z = x\ /\ y)$

definition *Integrator dt* = *sts* $(\lambda\ s\ .\ s = 0)\ (\lambda\ (s,x)\ (s',y)\ .\ y = s \wedge s' = s + x * dt)$

definition *TransferFcn dt* = *sts* $(\lambda\ (s,t)\ .\ s = 0 \wedge t = 0)\ (\lambda\ ((s,t),x)\ ((s',t'),\ y)\ .$
$y = -8 * s + 2 * x \wedge s' = s + (-4 * s - 2 * t + x) * dt \wedge t' = t + s * dt)$

5.1.4 Section 3.2.2: Variable Name Scope

definition *A-sts* = *sts* $(\lambda\ s.\ s > 0)\ (\lambda\ (s,\ (x,y))\ (s',\ z).\ z > s + x + y)$
definition *B-sts* = *sts* $(\lambda\ t.\ t > 0)\ (\lambda\ (t,\ (u,\ v))\ (t',\ w).\ w > t + u + v)$

lemma *A-sts* = *B-sts*

5.1.5 Section 3.2.3: Stateless STS Components

The semantic version of the stateless STS component is defined using the mapping stateless2sts from the paper.

definition *stateless-sts r* = *sts* $\top\ (\lambda\ (u::unit,x)\ (v::unit,\ y)\ .\ r\ x\ y)$

definition *Id-sts* = *stateless-sts* $(\lambda\ x\ y\ .\ y = x)$

definition *Add-sts* = *stateless-sts* $(\lambda\ (x,y)\ z\ .\ z = x + y)$

definition *Split-sts* = *stateless-sts* $(\lambda\ x\ (y,z)\ .\ y = x \wedge z = x)$

Div components can also be defined as sts component

lemma *Div-stateless*: *Div-sts* = *stateless-sts* $(\lambda\ (x,y)\ z\ .\ y \neq 0 \wedge z = x\ /\ y)$

5.1.6 Section 3.2.3: Deterministic STS Components

The semantic version of the deterministic STS component is defined using the mapping det2sts from the paper.

definition *det-sts s0 p state out* = *sts* $(\lambda\ s\ .\ s = s0)\ (\lambda\ (s,x)\ (s',y)\ .\ p\ (s,\ x) \wedge s' = state\ (s,x) \wedge y = out\ (s,\ x))$

lemma *UnitDelay-det*: *UnitDelay* = *det-sts 0* $\top\ (\lambda\ (s::'a::zero,\ x)\ .\ x)\ (\lambda\ (s,\ x)\ .\ s)$

lemma *Id-sts-det*: *Id-sts* = *det-sts* $()\ \top\ (\lambda\ (s::unit,\ x)\ .\ ())\ (\lambda\ (s::unit,\ x)\ .\ x)$

lemma *Add-sts-det*: *Add-sts* = *det-sts* $()\ \top\ (\lambda\ (s::unit,\ (x,y))\ .\ ())\ (\lambda\ (s::unit,\ (x,y))\ .\ x + y)$

lemma *Div-sts-det*: *Div-sts* = *det-sts* () (λ (*s*::*unit*, (*x*,*y*)) . *y* ≠ *0*) (λ (*s*::*unit*, (*x*,*y*)) . ()) (λ (*s*::*unit*, (*x*,*y*)) . *x* / *y*)

lemma *Split-sts-det*: *Split-sts* = *det-sts* () ⊤ (λ (*s*::*unit*, *x*) . ()) (λ (*s*::*unit*, *x*) . (*x*, *x*))

lemma *Sum-sts-det*: *Sum-sts* = *det-sts 0* ⊤ (λ (*s*, *x*) . *s* + *x*) (λ (*s*, *x*) . *s*)

5.1.7 Section 3.2.3: Stateless Deterministic STS Components

The semantic version of the stateless deterministic STS component is defined using the mapping stateless_det2det from the paper.

definition *stateless-det-sts p out* = *det-sts* () (λ (*s*::*unit*, *x*) . *p x*) (λ (*s*::*unit*, *x*) . ()) (λ (*s*::*unit*, *x*) . *out x*)

Many of the examples introduced above are both deterministic and stateless

lemma *Id-sts-stateless-det*: *Id-sts* = *stateless-det-sts* ⊤ (λ *x* . *x*)

lemma *Add-sts-stateless-det*: *Add-sts* = *stateless-det-sts* ⊤ (λ (*x*, *y*) . *x* + *y*)

lemma *Split-sts-stateless-det*: *Split-sts* = *stateless-det-sts* ⊤ (λ *x* . (*x*, *x*))

lemma *Div-sts-stateless-det*: *Div-sts* = *stateless-det-sts* (λ (*x*, *y*) . *y* ≠ *0*) (λ (*x*, *y*) . *x* / *y*)

fdbk is similar to Feedback but it requires the argument to have as input and output traces of pairs, while Feedback has as input and output pairs of traces.

definition *fdbk S* = *Feedback* ([− *u*, *x* ⇝ *u* || *x* −] *o S o* [− *uy* ⇝ *fst o uy*, *snd o uy* −])

Here is how the "Sum" composite component is defined (Simulink diagram in Fig.2).

definition *Sum-comp* = *fdbk* (*Add-sts o UnitDelay o Split-sts*)

We can prove later that Sum_sts = Sum_comp

5.1.8 Section 3.3: Quantified Linear Temporal Logic Components

5.1.9 Section 3.3.1: QLTL

For details on how QLTL is formalized in RCRS/Isabelle, see Temporal.thy

Lemma 1.

1. *top_dep p* is the semantic equivalent of *p* does not contain temporal operators.

definition *EXISTS* = *SUPREMUM UNIV*
definition *FORALL* = *INFIMUM UNIV*

The functions EXISTS and FORALL model the existential and universal quantifiers for QLTL formulas. If $p : A \to B \to bool$ is a predicate with two parameters, then $EXISTSp : B \to bool$ is a predicate with one parameter and $EXISTSpb = (\exists a.p\ a\ b)$.

lemma *lemma-1-1*: *top-dep p* ⟹ *EXISTS* (□ *p*) = □ (*EXISTS p*)

2.

lemma *lemma-1-2*: *p leads p* = □ *p*

3.

lemma *lemma-1-3*: $\top$ *leads* $p = \Box\ p$

4.

lemma *lemma-1-4*: p *leads* $\top = \top$

5.

lemma *lemma-1-5*: p *leads* $\bot = \bot$

6.

lemma *lemma-1-6*: *top-dep* $p \Longrightarrow$ *FORALL* $(p$ *leads* $(\lambda\ y\ .\ q)) = ((EXISTS\ p)$ *leads* $q)$

5.1.10 Section 3.3.2: QLTL Components

Semantically a QLTL component is a guarded property transformer on input output traces defined by a QLTL property. If $\alpha\ x\ y$ is a QLTL property then the QLTL component of α is:

definition *qltl* $\alpha = \{:\ \alpha\ :]$

However, for QLTL components, we use the syntax $\{:\ \alpha\ :]$ and its variant $\{:\ x \rightsquigarrow y.expr\ :]$, where *expr* is a QLTL expression on x and y

For example the oven QLTL component is defined by

definition *thermostat* $= \Box\ (\lambda\ t\ .\ 180 \leq t\ (0::nat) \wedge t\ 0 \leq 220)$
definition *oven* $= \ (\lambda\ t\ .\ t\ 0 = (20::nat)) \sqcap ((\lambda\ t\ .\ t\ 0 < t\ 1 \wedge t\ 0 < 180)$ *until thermostat*)
definition *theromstat-liveness* $= \diamondsuit\ (\lambda\ t.\ t\ (0::nat) > 200)$

definition *Oven-qltl* $= \{:x::(nat \Rightarrow unit) \rightsquigarrow t\ .\ oven\ t:]$

5.1.11 Section 3.4: Well Formed Components

Since in Isabelle the components are semantic objects, they are well formed if they type check in Isabelle

Next definition introduced a variant of the parallel composition closer to the parallel composition from the paper. In the paper we assume that traces of pairs are equivalent to pair of traces $(x, y) = (\lambda i.(x\ i, y\ i))$. The input of the new parallel composition variant is a trace of pairs, and the output is also a trace of pairs.

definition *parallel-component* :: $(((nat \Rightarrow 'a) \Rightarrow bool) \Rightarrow ((nat \Rightarrow 'b) \Rightarrow bool)) \Rightarrow (((nat \Rightarrow 'c) \Rightarrow bool) \Rightarrow ((nat \Rightarrow 'd) \Rightarrow bool))$
$\Rightarrow (((nat \Rightarrow 'a \times 'c) \Rightarrow bool) \Rightarrow ((nat \Rightarrow 'b \times 'd) \Rightarrow bool))$
(**infixr** $***$ *70*)
where
$(S\ ***\ T) = [-uv \rightsquigarrow fst\ o\ uv,\ snd\ o\ uv\ -]\ o\ (S\ **\ T)\ o\ [-x,y \rightsquigarrow x\ ||\ y\ -]$

definition *Switch1* $=$ *stateless-det-sts* $\top\ (\lambda\ (x,y).\ ((x,y),x))$
definition *Switch2* $=$ *stateless-det-sts* $\top\ (\lambda\ ((u,v),x).\ ((u,x),v))$

definition *Fig3 A B C* $=$ *A o Switch1 o* $(B\ ***$ *Id-sts*$)$ *o Switch2 o* $(C\ ***$ *Id-sts*$)$

5.2 Section 4: Semantics

5.2.1 Section 4.1: Monotonic Property Transformers

Definition 8 (Skip) can be found in Refinement.thy. You can see the definition by placing your cursor on the line "thm Skip_def". You can also control-click on "Skip_def" to be taken automatically to the definition.

thm *Skip-def*

Definition 9 (Fail) can be found in Refinement.thy.

thm *Fail-def*

Definition 10 (Assert) can be found in Refinement.thy.

thm *assert-def*

Definition 11 (Demonic update) can be found in Refinement.thy.

thm *demonic-def*

definition *DemonicEx1* = $[:x,\ y \rightsquigarrow z.\ (\forall\ i.\ z\ i = x\ i + y\ i) :]$
definition *DemonicEx3* = $[:\ x \rightsquigarrow y.\ \ y = (\lambda\ i.\ x\ i + 1)\ :]$

Lemma 2. The first equality is proved below; the second and third are proved in Refinement.thy by lemmas assert_true_skip and assert_rel_skip, whose definitions are repeated below.

lemma *skip-demonic-rel*: *Skip* = $[:\ x \rightsquigarrow x'.\ x'{=}x\ :]$
thm *assert-true-skip*
thm *assert-rel-skip*

Definition 12 (Angelic update) can be found in Refinement.thy.

thm *angelic-def*

Lemma 3.

lemma *assert-angelic-upd*: $\{.p.\} = \{:\ x \rightsquigarrow x'.\ p\ x \wedge x' = x:\}$

Results for serial composition. These results are proved in Refinement.thy by mono_comp_a, comp_skip and skip_comp.

thm *mono-comp-a*
thm *comp-skip*
thm *skip-comp*

Definition 13 (Product) can be found in Refinement.thy. Instead of the product notation $\otimes$ used in the paper, the notation ** is used in RCRS/Isabelle. That is, product corresponds to parallel composition.

thm *Prod-def*

Lemma 4 is proved in Refinement.thy by lemma mono_prod.

thm *mono-prod*

Skip with Unit as input and output type is the neutral element for product.

lemma $[:x \rightsquigarrow y.\ r\ x\ y:] ** (Skip::(unit \Rightarrow bool) \Rightarrow (unit \Rightarrow bool)) = [:\ (x,\ u::unit) \rightsquigarrow (y,\ v::unit).\ r\ x\ y\ :]$

lemma $(Skip::(unit \Rightarrow bool) \Rightarrow (unit \Rightarrow bool)) ** [:r:] = [:\ (u::unit,\ x) \rightsquigarrow (v::unit,\ y).\ r\ x\ y\ :]$

Definition 14 (Fusion) can be found in Refinement.thy.

thm *Fusion-def*

Lemma 5 is proved in Refinement.thy by lemma Fusion_spec.

thm *Fusion-spec*

Definition 15 (IterateOmega) can be found in DelayFeedback.thy.

thm *IterateOmegaA-def*

Definition 16 (Feedback) can be found in DelayFeedback.thy.

thm *Feedback-def*
thm *IterateOmegaA-def*
thm *IterateMaskA-def*
thm *Mask-def*

Computing feedback of delayed sum.

definition *S-comp* = $[- \lambda\ (u,\ x).\ ((\lambda i.\ if\ i = 0\ then\ 0\ else\ x(i - 1) + u(i - 1)),\ (\lambda i.\ if\ i = 0\ then\ 0\ else\ x(i - 1) + u(i - 1)))-]$

definition *T-comp* = $[-(u,\ (y{::}nat \Rightarrow nat)),\ x \rightsquigarrow ((u{::}nat \Rightarrow nat),\ x),\ x-] \circ S\text{-}comp **\ Skip$

lemma *T-comp-simp*: *T-comp*
$= [-(u,y),\ x \rightsquigarrow ((\lambda i.\ if\ i = 0\ then\ 0\ else\ x(i - 1) + u(i - 1)),(\lambda i.\ if\ i = 0\ then\ 0\ else\ x(i - 1) + u(i - 1))),\ x\ -]$

thm *Summ.simps*

lemma *Summ-Suc*: $Summ\ (\lambda a.\ b\ (Suc\ a))\ n + b\ 0 = Summ\ b\ n + b\ n$

lemma *Summ-at-Suc*: $\bigwedge\ b\ .\ Summ\ (b[Suc\ k\ ..])\ n + b\ k = Summ\ (b[k..])\ n + b\ (n + k)$

lemma *T-comp-power*: *T-comp* ^^ $(Suc\ n) =$
$[-(u,y),\ x \rightsquigarrow ((\lambda i.\ if\ i \leq n\ then\ Summ\ x\ i\ else\ Summ\ (x[i - Suc\ n..])\ (Suc\ n)\ + u\ (i - Suc\ n)),$
$(\lambda i.\ if\ i \leq n\ then\ Summ\ x\ i\ else\ Summ\ (x[i - Suc\ n..])\ (Suc\ n)\ + u\ (i - Suc\ n))),\ x\ -]$

lemma *T-comp-IterateMaskA*: *IterateMaskA T-comp* $n = [:(u,\ y),\ x \rightsquigarrow (u',\ y'),\ x'\ .$
$(\forall\ i < n - 1\ .\ x'\ i = x\ i \wedge y'\ i = Summ\ x\ i \wedge u'\ i = y'\ i):]$

Next lemma proves relation (1) from the paper.

lemma *Feedback-S-comp*: *Feedback S-comp* $= [-Summ-]$

Definition 17 (Refinement) is part of the Isabelle libraries (Orderings.thy). Since MPTs are functions, refinement is simply an ordering on functions:

thm *le-fun-def*

Theorem 1

Theorem 1.1. These results are proved in Refinement.thy by lemmas mono_comp, prod_mono1, prod_mono2, and fusion_mono1.

thm *mono-comp*
thm *prod-mono1*
thm *prod-mono2*
thm *Fusion-refinement*

thm *fusion-mono1*

Theorem 1.2.

lemma *theorem-1-2*: $mono\ S \Longrightarrow S \leq T \Longrightarrow IterateOmegaA\ S \leq IterateOmegaA\ T$

Theorem 1.3.

lemma *theorem-1-3*: $S \leq T \Longrightarrow Feedback\ S \leq Feedback\ T$

5.2.2 Section 4.2: Subclasses of MPTs

Def.18 simply defines the terminology RPT. Note that Property Transformers are instances of Predicate Transformers (and predicate transformers are themselves instances of functions). A predicate transformer is a function of type ('a → bool) → ('b → bool) where types 'a and 'b are arbitrary. When these types are types of infinite sequences, we get a property transformer, which is a function of type: ((nat → 'a) → bool) → ((nat → 'b) → bool).

Much of the RCRS formalization in Isabelle is done in terms of predicate transformers, in order to establish more general results. Results that hold for (general) predicate transformers automatically hold also for (the more specific) property transformers.

We sometimes wish to work with property transformers directly. Below, we define the construct "sts init r", which produces a property transformer of type ((nat → 'a) → bool) → ((nat → 'b) → bool) where init is of type ('c → bool) and r of type ('c × 'b → 'c × 'a → bool).

A series of small RPT examples after Def.18, stated as lemmas:

lemma *Fail-is-a-RPT*: $Fail = \{.\ x\ .\ False\ .\}\ o\ [:\ x \rightsquigarrow y\ .\ True\ :]$

lemma *Skip-is-a-RPT*: $Skip = \{.\ x.\ True.\}\ o\ [:\ x \rightsquigarrow y.\ y = x\ :]$

lemma *Assert-is-a-RPT*: $\{.p.\} = \{.p.\}\ o\ [:\ x \rightsquigarrow y.\ y{=}x:]$

lemma *Demonic-is-a-RPT*: $[:r:] = \{.\top.\}\ o\ [:r:]$

definition *RPT-S1* $= \{.\ \top\ .\}\ o\ [:\ (x,\ y) \rightsquigarrow z\ .\ y \neq 0 \wedge z = x\ /\ y\ :]$
definition *RPT-S2* $= \{.\ (x,\ y)\ .\ y \neq 0\ .\}\ o\ [:\ (x,\ y) \rightsquigarrow z\ .\ z = x\ /\ y\ :]$

Theorem 2 is proved in Refinement.thy by lemmas assert_demonic_comp, Prod_spec, fusion_spec

thm *assert-demonic-comp*
thm *Prod-spec*
thm *Fusion-spec*

The theorem 2 in the paper uses Fusion applied to two RPTs, but Fusion_spec is proved for an arbitrary number of RPTs.

RPTs are not closed under Feedback operation.

lemma *Feedback* $[-u{::}nat \Rightarrow {'a},\ x{::}nat \Rightarrow {'b} \rightsquigarrow u,\ u-] = \{:\top:\}$

Theorem 3 is proved in Refinement.thy by lemma assert_demonic_refinement

thm *assert-demonic-refinement*

5.2.3 Section 4.2.2: Guarded MPTs

Definition 19 is given in Refinement.thy by the definition of trs

thm *trs-def*

thm *Magic-def*
lemma *MagicAlternativeDef*: $Magic = [: x \rightsquigarrow y \ . \ False :]$

lemma *Fail-is-a-GPT*: $Fail = \{: \bot :]$

lemma *Skip-is-s-GPT*: $Skip = \{: x \rightsquigarrow y.\ y = x :]$

lemma *Assert-is-a-GPT*: $\{.p.\} = \{: x \rightsquigarrow y.\ p\ x \wedge y = x :]$

lemma $inpt\ r = \top \Longrightarrow [:r:] = \{:r:]$

lemma $[:r:] = \{: r:] \Longrightarrow inpt\ r = \top$

Theorem 4 is proved in Refinement.thy by lemmas trs_trs and trs_prod.

thm *trs-trs*
thm *trs-prod*

Corollary 1 is proved in Refinement.thy by lemmas trs_refinement.

thm *trs-refinement*

5.2.4 Section 4.3: Semantics of Components as MPTs

As mentioned already, the components are semantic objects. The semantics of qltl component, relation (2), is the definition qltl_def. The semantics of the serial composition, relation (3), is the function composition of property transformers. The semantics of the parallel composition, relation (4), is the definition parallel_component_def. The semantics of the feedback composition, relation (5), is the definition fdbk_def

thm *qltl-def*
thm *parallel-component-def*
thm *fdbk-def*

Lemma 6.

lemma *lemma-6*: $\{: x \rightsquigarrow y.\ inpt\ r\ x \wedge r\ x\ y :] = \{: x \rightsquigarrow y.\ r\ x\ y:]$

The semantics of the sts components, relation (6), is given by sts_def

thm *sts-def*

The semantics of the other components are given by their definitions:

thm *stateless-sts-def*
thm *det-sts-def*
thm *stateless-det-sts-def*

Next lemma is an auxiliary results that links the definition oo sts to LocalSystem defined in RefinementReactive.thy.

lemma *sts-LocalSystem*: $sts\ init\ r = LocalSystem\ init\ (inpt\ r)\ r$

lemma *sts-inpt-top*: $inpt\ r = \top \Longrightarrow sts\ init\ r = [:rel\text{-}pre\text{-}sts\ init\ r:]$

lemma *stateless2LocalSystem*: *stateless-sts r = LocalSystem* (⊤::*unit*⇒*bool*) (λ (*s*::*unit*, *x*) . *inpt r x*) (λ (*s*::*unit*, *x*) (*s'*::*unit*, *y*) . *r x y*)

lemma *det2LocalSystem*: *det-sts s0 p state out = LocalSystem* (λ *s* . *s = s0*) *p* (λ (*s*,*x*) (*s'*,*y*) . *s' = state* (*s*,*x*) ∧ *y = out* (*s*, *x*))

lemma *stateless-det2LocalSystem*: *stateless-det-sts p out = LocalSystem* (⊤::*unit*⇒*bool*) (λ (*s*::*unit*, *x*) . *p x*) (λ (*s*::*unit*, *x*) (*s'*::*unit*, *y*) . *y = out x*)

Lemma 7.

theorem *stateless-det2stateless*: *stateless-det-sts p out = stateless-sts* (λ *x y* . *p x* ∧ *y = out x*)

thm *Sum-comp-def*

5.2.5 Section 4.3.1: Example: Two Alternative Derivations of the Semantics of Diagram Sum

lemma *Add-sts-simp*: *Add-sts* = [−*ux* ⇝ (λ *i* . *fst* (*ux i*) + *snd* (*ux i*))−]

lemma *UnitDelay-simp*: *UnitDelay* = [−*x* ⇝ (λ *i* . *if i = 0 then 0 else x* (*i* − *1*))−]

lemma *Split-sts-simp*: *Split-sts* = [−*x* ⇝ (λ *i* . (*x i*, *x i*))−]

lemma *Sum-comp-simp*: *Sum-comp* = [−*Summ*−]

The SumAtomic sts is the same as Sum_sts defined above

thm *Sum-sts-def*

lemma *Sum-sts-simp*: *Sum-sts* = [: *x* ⇝ *y* . ∃ *s* . *s 0 = 0* ∧ (∀ *i* . *y i = s i* ∧ *s* (*Suc i*) = *s i* + *x i*) :]

lemma *Sum-comp-Sum-sts*: *Sum-comp = Sum-sts*

lemmas *ex1 = Sum-comp-Sum-sts*

5.2.6 Section 4.3.2: Characterization of Legal Input Traces

The function legal from the paper is implemented by the function prec in the Isabelle theories

definition *legal S = S* ⊤
lemma *legal-prec*: *legal S* = ((*prec S*)::*'a*::*boolean-algebra*)

Lemma 8 is proved below.

lemma *legal-RPT*: *legal* ({.*p*.} *o* [:*r*::*'a* ⇒ *'b* ⇒ *bool*:]) = *p*

lemma *legal-GPT*: *legal* ({:*r*:]) = (*inpt r*)

lemma *legal-sts-1*: *legal* (*sts init r*) = (−*illegal-sts init* (*inpt r*) *r*)

lemma *legal-sts-2*: *legal* (*sts init r*) = (*prec-pre-sts init* (*inpt r*) *r*)

lemma *legal-qltl*: *legal* (*qltl r*) = (*inpt r*)

lemmas *lemma-8 = legal-RPT legal-GPT legal-sts-1 legal-sts-2 legal-qltl*

Theorem 5. The first result is the associativity of function composition. The second item cannot be expressed as clean as in the paper. In the paper we assume concatenation of tuples that cannot be defined in Isabelle

thm *comp-assoc*

theorem *theorem-5-2*: S ** $(S'$ ** $S'') = [-x,y,z \rightsquigarrow (x,y),\ z-]$ o $((S$ ** $S')$ ** $S'')$ o $[-(x,y),z \rightsquigarrow x,y,z-]$

Theorem 5. The third item is proved next

lemma $(Skip$ ** $Magic)$ o $(Fail$ ** $Fail) \neq (Skip$ o $Fail)$ ** $(Magic$ o $Fail)$

theorem *theorem-5-3-aux*: $p \leq inpt\ r \Longrightarrow p' \leq inpt\ r' \Longrightarrow (([.p.]$ o $[:r:])$ ** $([.p'.]$ o $[:r':]))$ o $(([.q.]$ o $[:s:])$ ** $([.q'.]$ o $[:s':]))$
$= (([.p.]$ o $[:r:])$ o $([.q.]$ o $[:s:]))$ ** $(([.p'.]$ o $[:r':])$ o $([.q'.]$ o $[:s':]))$

theorem *theorem-5-3*: $([:r:]$ ** $[:r':])$ o $(([.q.]$ o $[:s:])$ ** $([.q'.]$ o $[:s':]))$
$= (([:r:])$ o $([.q.]$ o $[:s:]))$ ** $(([:r':])$ o $([.q'.]$ o $[:s':]))$

Theorem 5. The fourth result is proved by in Refinement.thy by lemma mono_comp, by lemma prod_ref below and in ReactiveRefinement.thy by lemma Feedback_refin, respectively.

thm *mono-comp*

lemma *prod-ref*: $S \leq S' \Longrightarrow T \leq T' \Longrightarrow S$ ** $T \leq S'$ ** T'

lemma *theorem-5-4-c*: $mono\ S \Longrightarrow S \leq T \Longrightarrow fdbk\ S \leq fdbk\ T$

lemmas *theorem-5 = comp-assoc theorem-5-2 theorem-5-3*
mono-comp prod-ref theorem-5-4-c

lemma *theorem-6*: $(S \leq T) = (\forall\ p\ q\ .\ Hoare\ (p::'a::order)\ S\ q \longrightarrow Hoare\ p\ T\ q)$

5.3 Section 5: Symbolic Reasoning

Theorem 7.

definition *sts2qltl init r* $= (\lambda\ x\ y\ .\ prec\text{-}pre\text{-}sts\ init\ (inpt\ r)\ r\ x \wedge rel\text{-}pre\text{-}sts\ init\ r\ x\ y)$

thm *prec-pre-sts-def*
thm *rel-pre-sts-def*

theorem *theorem-7-sts-a*: $init\ a \Longrightarrow sts\ init\ r = [:sts2qltl\ init\ r:]$

theorem *theorem-7-sts*: $init\ a \Longrightarrow sts\ init\ r = qltl\ (sts2qltl\ init\ r)$

lemma *stateless-sts-simp*: $stateless\text{-}sts\ r = [.(\Box\ (\lambda\ x\ .\ inpt\ r\ (x\ 0))).]$ o $[:\ (\Box\ (\lambda\ x\ y.\ r\ (x\ 0)\ (y\ 0))):]$

theorem *theorem-7-stateless-sts-a*: $stateless\text{-}sts\ r = [:\ (\Box\ (\lambda\ x\ y.\ r\ (x\ (0::nat))\ (y\ (0::nat)))):]$

theorem *theorem-7-stateless-sts*: $stateless\text{-}sts\ r = qltl\ (\Box\ (\lambda\ x\ y.\ r\ (\ x\ (0::nat))\ (y\ (0::nat))))$

lemmas *theorem-7 = theorem-7-sts theorem-7-stateless-sts*

lemma *stateless-sts* $(\lambda\ x\ y\ .\ y > x) = qltl\ (\Box\ (\lambda\ x\ y\ .\ y\ (0::nat) > x\ (0::nat)))$

lemma *stateless-sts* $(\lambda\ x\ (y::unit)\ .\ x > 0) = qltl\ (\Box\ (\lambda\ x\ y\ .\ x\ (0::nat) > 0))$

lemma *UnitDelay* $= qltl\ (\lambda\ x\ y\ .\ y\ 0 = 0 \wedge (\Box\ (\lambda\ x\ y\ .\ y\ (1::nat) = x\ (0::nat)))\ x\ y)$

5.3.1 Section 5.3: Symbolic Computation of Serial Composition.

Theorem 8 for Equation 13.

theorem *qltl-serial-a*: $r'' = (\lambda\ x\ z\ .\ (\forall\ y\ .\ r\ x\ y \longrightarrow inpt\ r'\ y) \wedge (\exists\ y\ .\ r\ x\ y \wedge r'\ y\ z))$
$\Longrightarrow \{:r:\]\ o\ \{:r':\] = \{:r'':\]$

theorem *qltl-serial*: $r'' = (\lambda\ x\ z\ .\ (\forall\ y\ .\ r\ x\ y \longrightarrow inpt\ r'\ y) \wedge (\exists\ y\ .\ r\ x\ y \wedge r'\ y\ z))$
$\Longrightarrow qltl\ r\ o\ qltl\ r' = qltl\ r''$

Theorem 8 for Equation 14.

definition *sts-comp-rel* $r\ r' = (\lambda\ ((u,v),\ x)\ ((u',v'),\ z)\ .\ inpt\ r\ (u,x) \wedge (\forall\ y\ u'\ .\ r\ (u,\ x)\ (u',y)$
$\longrightarrow inpt\ r'\ (v,y)) \wedge (\exists\ y\ .\ r\ (u,x)\ (u',y) \wedge r'\ (v,y)\ (v',z))\ )$

theorem *sts-serial*: $init'\ a \Longrightarrow sts\ init\ r\ o\ sts\ init'\ r' = sts\ (prod\text{-}pred\ init\ init')\ (sts\text{-}comp\text{-}rel\ r\ r')$

Theorem 8 for Equation 15.

theorem *stateless-serial*: *stateless-sts r o stateless-sts r'*
$=$ *stateless-sts* $(\lambda\ x\ z\ .\ (\forall\ y\ .\ r\ x\ y \longrightarrow inpt\ r'\ y) \wedge (\exists\ y\ .\ r\ x\ y \wedge r'\ y\ z))$

Theorem 8 for Equation 16.

theorem *det-serial*: *det-sts s0 p state out o det-sts s0' p' state' out'*
$=$ *det-sts* $(s0,\ s0')\ \ (\lambda\ ((s,s'),x)\ .\ p\ (s,\ x) \wedge p'\ (s',\ out\ (s,\ x)))\ (\lambda\ ((s,s'),x)\ .\ (state\ (s,x),\ state'\ (s',$
$out(s,x))))$
$(\lambda\ ((s,s'),x)\ .\ (out'\ (s',\ out(s,x))))$

Theorem 8 for Equation 17.

theorem *stateless-det-serial*: *stateless-det-sts p out o stateless-det-sts p' out'* $=$
stateless-det-sts $(p \sqcap\ (p'\ o\ out))\ (out'\ o\ out)$

lemmas *theorem-8* $=$ *qltl-serial sts-serial stateless-serial det-serial stateless-det-serial*

definition *C1-comp* $=$ *stateless-sts* $\top$
definition *C2-comp* $=$ *stateless-det-sts* $(\lambda\ (x,\ y)\ .\ y \neq (0::real))\ \ (\lambda\ (x,\ y)\ .\ x\ /\ y)$

lemma *C1-comp o C2-comp* $=$ *stateless-sts* $\bot$

lemma assumes x: $x = (\lambda\ x\ y\ .\ x\ (0::nat))$ **and** y: $y = (\lambda\ x\ y\ .\ y\ (0::nat))$
shows $qltl\ (\Box\ (x \rightarrow \Diamond\ y))\ o\ qltl\ (\Box \Diamond\ x) = qltl\ (\Box \Diamond\ x)$

5.3.2 Section 5.4: Symbolic Computation of Parallel composition

Theorem 9 for Equation 18.

theorem *qltl-parallel-a*: $\{:r:] ** \{:r':] = \{: (x,x') \rightsquigarrow (y,y') \;.\; r\ x\ y \wedge r'\ x'\ y':]$

theorem *qltl-parallel-b*: $\{:r:] *** \{:r':] = \{: x \rightsquigarrow y \;.\; r\ (fst\ o\ x)\ (fst\ o\ y) \wedge r'\ (snd\ o\ x)\ (snd\ o\ y):]$

theorem *qltl-parallel*: $qltl\ r ** qltl\ r' = qltl\ (\lambda\ (x, x')\ (y, y').\ r\ x\ y \wedge\ r'\ x'\ y')$

Theorem 9 for Equation 19.

theorem *sts-parallel-a*: $init\ a \Longrightarrow init'\ b \Longrightarrow sts\ init\ r ** sts\ init'\ r' =$
$[-\ (x, x') \rightsquigarrow x \parallel x' -] \circ sts\ (prod\text{-}pred\ init\ init')\ (rel\text{-}prod\text{-}sts\ r\ r')\ o\ [-\ y \rightsquigarrow (fst \circ y, snd \circ y) -]$

lemma *split-nzip*: $[-\ uv \rightsquigarrow (fst \circ uv, snd \circ uv) -] \circ [-\ (x, x') \rightsquigarrow x \parallel x' -] = [-id-]$

theorem *sts-parallel*: $init\ a \Longrightarrow init'\ b \Longrightarrow sts\ init\ r *** sts\ init'\ r' = sts\ (prod\text{-}pred\ init\ init')\ (rel\text{-}prod\text{-}sts\ r\ r')$

Theorem 9 for Equation 20.

theorem *stateless-parallel-a*: $stateless\text{-}sts\ r ** stateless\text{-}sts\ r' =$
$[-\ (x, x') \rightsquigarrow x \parallel x' -] \circ stateless\text{-}sts\ (\lambda\ (x,x')\ (y,y')\ .\ r\ x\ y \wedge r'\ x'\ y')\ o\ [-\ y \rightsquigarrow (fst \circ y, snd \circ y) -]$

theorem *stateless-parallel*: $stateless\text{-}sts\ r *** stateless\text{-}sts\ r' = stateless\text{-}sts\ (\lambda\ (x,x')\ (y,y')\ .\ r\ x\ y \wedge r'\ x'\ y')$

Theorem 9 for Equation 21.

theorem *det-parallel-a*: $(det\text{-}sts\ s0\ p\ state\ out) ** (det\text{-}sts\ s0'\ p'\ state'\ out')$
$= [-\ (x, x') \rightsquigarrow x \parallel x' -] \circ det\text{-}sts\ (s0, s0')\ (prec\text{-}prod\text{-}sts\ p\ p')\ (\lambda\ ((s,s'), (x,x')\ )\ .\ (state\ (s,x), state'\ (s',x'))\ )$
$(\lambda\ ((s,s'), (x,x')\ )\ .\ (out\ (s,x), out'\ (s',x'))\ )\ o\ [-\ y \rightsquigarrow (fst \circ y, snd \circ y) -]$

theorem *det-parallel*: $(det\text{-}sts\ s0\ p\ state\ out) *** (det\text{-}sts\ s0'\ p'\ state'\ out')$
$= det\text{-}sts\ (s0, s0')\ (prec\text{-}prod\text{-}sts\ p\ p')\ (\lambda\ ((s,s'), (x,x')\ )\ .\ (state\ (s,x), state'\ (s',x'))\ )$
$(\lambda\ ((s,s'), (x,x')\ )\ .\ (out\ (s,x), out'\ (s',x'))\ )$

Theorem 9 for Equation 22.

theorem *stateless-det-parallel-a*: $stateless\text{-}det\text{-}sts\ p\ out ** stateless\text{-}det\text{-}sts\ p'\ out' =$
$[-\ (x, x') \rightsquigarrow x \parallel x' -] \circ stateless\text{-}det\text{-}sts\ (prod\text{-}pred\ p\ p')\ (\lambda\ (x,x')\ .\ (out\ x, out'\ x'))\ o\ [-\ y \rightsquigarrow (fst \circ y, snd \circ y) -]$

theorem *stateless-det-parallel*: $stateless\text{-}det\text{-}sts\ p\ out *** stateless\text{-}det\text{-}sts\ p'\ out' =$
$stateless\text{-}det\text{-}sts\ (prod\text{-}pred\ p\ p')\ (\lambda\ (x,x')\ .\ (out\ x, out'\ x'))$

lemmas *theorem-9 = qltl-parallel sts-parallel stateless-parallel det-parallel stateless-det-parallel*

5.5 Symbolic Computation of Feedback Composition

Theorem 10 for Equation 23.

theorem *det-decomposable-feedback*: $Feedback\ ([-\ u, x \rightsquigarrow u \parallel x -]\ o\ det\text{-}sts\ s0\ p\ state\ (\lambda\ (s, (u,x))\ .\ (f\ s\ x, g\ u\ s\ x)\ )\ o\ [-\ uy \rightsquigarrow fst\ o\ uy, snd\ o\ uy -])$
$= det\text{-}sts\ s0\ (\lambda\ (s,x)\ .\ p\ (s, (f\ s\ x, x)))\ (\lambda\ (s,x)\ .\ state\ (s, (f\ s\ x, x)))\ (\lambda\ (s,x)\ .\ g\ (f\ s\ x)\ s\ x)$

theorem *det-decomposable-feedback-a*: *fdbk* (*det-sts s0 p state* (λ (*s*, (*u*,*x*)) . (*f s x*, *g u s x*)))
= *det-sts s0* (λ (*s*,*x*) . *p* (*s*, (*f s x*, *x*))) (λ (*s*,*x*) . *state* (*s*, (*f s x*, *x*))) (λ (*s*,*x*) . *g* (*f s x*) *s x*)

Theorem 10 for Equation 24.

theorem *stateless-det-decomposable-feedback*: *Feedback* ([− *u*, *x* ⇝ *u* || *x* −] *o stateless-det-sts p* (λ (*u*,*x*) . (*f x*, *g u x*)) *o* [− *uy* ⇝ *fst o uy*, *snd o uy* −])
= *stateless-det-sts* (λ *x* . *p* (*f x*, *x*)) (λ *x* . *g* (*f x*) *x*)

theorem *stateless-det-decomposable-feedback-a*: *fdbk* (*stateless-det-sts p* (λ (*u*,*x*) . (*f x*, *g u x*)))
= *stateless-det-sts* (λ *x* . *p* (*f x*, *x*)) (λ *x* . *g* (*f x*) *x*)

lemmas *theorem-10* = *det-decomposable-feedback-a stateless-det-decomposable-feedback-a*

lemma *Sum-comp* = *det-sts* (*0*::*nat*) ⊤ (λ (*s*, *y*) . *s* + *y*) (λ (*s*, *x*) . *s*)

5.3.3 Section 5.8: Checking Validity

Theorem 12 for QLTL components.

theorem *theorem-12-qltl-a*: ({:*r*:] = *Fail*) = (*r* = ⊥)

theorem *theorem-12-qltl*: (*qltl r* = *Fail*) = (*r* = ⊥)

Theorem 12 for stateless STS components.

theorem *theorem-12-stateless-sts*: (*stateless-sts r* = *Fail*) = (*r* = ⊥)

lemmas *theorem-12* = *theorem-12-qltl theorem-12-stateless-sts*

Legal inputs

Theorem 13 for Equation 25.

thm *legal-qltl*

Theorem 13 for Equation 26.

lemma *legal-sts*: *init a* ⟹ *legal* (*sts init r*) = *prec-pre-sts init* (*inpt r*) *r*

Theorem 13 for Equation 27.

lemma *legal-stateless*: *legal* (*stateless-sts r*) = (□ (λ*x* . *inpt r* (*x* (*0*::*nat*))))

Theorem 13 for Equation 28.

lemma *legal-det*: *legal* (*det-sts s0 p state out*) =
prec-pre-sts (λ*s*. *s* = *s0*) *p* (λ(*s*, *x*) (*s*′, *y*). (*s*′ = *state* (*s*, *x*) ∧ *y* = *out* (*s*, *x*)))

Theorem 13 for Equation 29.

lemma *legal-stateless-det*: *legal* (*stateless-det-sts p out*) = □ (λ *x* . *p* (*x 0*))

lemmas *theorem-13* = *legal-qltl legal-sts legal-det legal-stateless legal-stateless-det*

5.3.4 Section 5.10: Checking Refinement Symbolically

lemma *refinement-LocalSystem*: $init' \leq init \Longrightarrow p \leq p' \Longrightarrow (\bigwedge x\ .\ p\ x \Longrightarrow r'\ x \leq r\ x) \Longrightarrow$ *LocalSystem* $init\ p\ r \leq$ *LocalSystem* $init'\ p'\ r'$

Theorem 14 for STS components.

theorem *refinement-sts*: $init' \leq init \Longrightarrow inpt\ r \leq inpt\ r' \Longrightarrow (\bigwedge x\ .\ inpt\ r\ x \Longrightarrow r'\ x \leq r\ x) \Longrightarrow sts\ init\ r \leq sts\ init'\ r'$

Theorem 14 for stateless STS components.

theorem *refinement-stateless*: $(stateless\text{-}sts\ r \leq stateless\text{-}sts\ r') = ((inpt\ r \leq inpt\ r') \wedge ((\forall\ x\ .\ inpt\ r\ x \longrightarrow r'\ x \leq r\ x)))$

Theorem 14 for QLTL components.

theorem *refinement-qltl-a*: $(\{:r:] \leq \{:r':]) = ((\forall\ x\ .\ inpt\ r\ x \longrightarrow inpt\ r'\ x) \wedge (\forall\ x\ y\ .\ inpt\ r\ x \wedge r'\ x\ y \longrightarrow r\ x\ y))$

theorem *refinement-qltl*: $(qltl\ r \leq qltl\ r') = ((\forall\ x\ .\ inpt\ r\ x \longrightarrow inpt\ r'\ x) \wedge (\forall\ x\ y\ .\ inpt\ r\ x \wedge r'\ x\ y \longrightarrow r\ x\ y))$

lemmas *theorem-14* = *refinement-sts refinement-stateless refinement-qltl*

Data refinement

Theorem 15.

theorem *theorem-15*:
assumes *A*: $(\bigwedge t\ .\ init'\ t \Longrightarrow \exists\ s\ .\ d\ t\ s \wedge init\ s)$
and *B*: $\bigwedge t\ x\ s.\ d\ t\ s \Longrightarrow inpt\ r\ (s, x) \Longrightarrow inpt\ r'\ (t, x)$
and *C*: $\bigwedge t\ x\ s\ t'\ y.\ d\ t\ s \Longrightarrow inpt\ r\ (s, x) \Longrightarrow r'\ (t, x)\ (t', y) \Longrightarrow (\exists s'.\ d\ t'\ s' \wedge r\ (s, x)\ (s', y))$
shows *sts* $init\ r \leq sts\ init'\ r'$

Example of stateless sts refinement

lemma *stateless-sts* $(\lambda\ x\ y\ .\ x \geq 0 \wedge y \geq (x::nat)) \leq$ *stateless-sts* $(\lambda\ x\ y\ .\ x \leq y \wedge y \leq x + 10)$

5.3.5 Proof of refinement for the Oven example

datatype *oven-state* = *on* | *off*

definition *oven-trs* = $(\lambda\ ((s::nat, sw),\ x::unit)\ ((s', sw'), t)\ .\ (t = s) \wedge$
$(if\ sw = on\ then\ s < s' \wedge s' < s + 5\ else\ (if\ s > 10\ then\ s - 5 < s' \wedge s' < s\ else\ s' = s)) \wedge$
$(if\ sw = on \wedge s > 210\ then\ sw' = off\ else$
$(if\ sw = off \wedge s < 190\ then\ sw' = on\ else\ sw' = sw))\ )$

definition *oven-init* = $(\lambda\ (s,\ sw)\ .\ s = (20::nat) \wedge sw = on)$

lemma *oven-refinement*: *Oven-qltl* $\leq$ *sts oven-init oven-trs*

end

6 Instantaneous Feedback

theory *InstantaneousFeedback* **imports** *../RefinementReactive/Refinement*
begin

datatype *'a fail-option* = *Fail* (·) | *OK* (*elem* :*'a*)

class *order-bot-max* = *order-bot* +
fixes *maximal* :: *'a* ⇒ *bool*
assumes *maximal-def*: *maximal x* = (∀ *y* . ¬ *x* < *y*)
assumes [*simp*]: ¬ *maximal* ⊥
begin
lemma *ex-not-le-bot*[*simp*]: ∃ *a*. ¬ *a* ≤ ⊥
end

instantiation *option* :: (*type*) *order-bot-max*
begin
definition *bot-option-def*: (⊥::*'a option*) = *None*
definition *le-option-def*: ((*x*::*'a option*) ≤ *y*) = (*x* = *None* ∨ *x* = *y*)
definition *less-option-def*: ((*x*::*'a option*) < *y*) = (*x* ≤ *y* ∧ ¬ (*y* ≤ *x*))
definition *maximal-option-def*: *maximal* (*x*::*'a option*) = (∀ *y* . ¬ *x* < *y*)

instance

lemma [*simp*]: *None* ≤ *x*
end

context *order-bot*
begin
definition *is-lfp f x* = ((*f x* = *x*) ∧ (∀ *y* . *f y* = *y* ⟶ *x* ≤ *y*))
definition *emono f* = (∀ *x y*. *x* ≤ *y* ⟶ *f x* ≤ *f y*)

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

lemma *lfp-unique*: *is-lfp f x* ⟹ *is-lfp f y* ⟹ *x* = *y*

lemma *lfp-exists*: *is-lfp f x* ⟹ *Lfp f* = *x*

lemma *emono-a*: *emono f* ⟹ *x* ≤ *y* ⟹ *f x* ≤ *f y*

lemma *emono-fix*: *emono f* ⟹ *f y* = *y* ⟹ (*f* ^^ *n*) ⊥ ≤ *y*

lemma *emono-is-lfp*: *emono* (*f*::*'a* ⇒ *'a*) ⟹ (*f* ^^ (*n* + *1*)) ⊥ = (*f* ^^ *n*) ⊥ ⟹ *is-lfp f* ((*f* ^^ *n*) ⊥)

lemma *emono-lfp-bot*: *emono* (*f*::*'a* ⇒ *'a*) ⟹ (*f* ^^ (*n* + *1*)) ⊥ = (*f* ^^ *n*) ⊥ ⟹ *Lfp f* = ((*f* ^^ *n*) ⊥)

lemma *emono-up*: *emono f* ⟹ (*f* ^^ *n*) ⊥ ≤ (*f* ^^ (*Suc n*)) ⊥
end

context *order*
begin
definition *min-set A* = (*SOME n* . *n* ∈ *A* ∧ (∀ *x* ∈ *A* . *n* ≤ *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*: $(n::nat) \in A \Longrightarrow min\text{-}set\ A \leq n$

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

end

declare [[*show-types*]]
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

instantiation *fail-option* :: (*order-bot*) {*order-bot*, *order-top*}
begin
definition *bot-fail-option-def*: $(\bot::'a\ fail\text{-}option) = OK\ \bot$
definition *top-fail-option-def*: $(\top::'a\ fail\text{-}option) = \cdot$
definition *le-fail-option-def*: $((x::'a\ fail\text{-}option) \leq y) = ((case\ x\ of\ OK\ a \Rightarrow (case\ y\ of\ OK\ b \Rightarrow a \leq b \mid \cdot \Rightarrow True) \mid \cdot \Rightarrow y = \cdot))$
definition *less-fail-option-def*: $((x::'a\ fail\text{-}option) < y) = (x \leq y \wedge \neg\ (y \leq x))$
instance
end

lemma *maximal-prod-1*: $maximal\ (a, b) \Longrightarrow maximal\ a$

lemma *maximal-prod-2*: $maximal\ (a, b) \Longrightarrow maximal\ b$

lemma *maximal-prod*: $maximal\ (a, b) = (maximal\ a \wedge maximal\ b)$

lemma *drop-assumption*: $p \Longrightarrow True$

lemma *Sup-OO*: $(Sup\ A)\ OO\ r = Sup\ \{x\ .\ \exists\ y{\in}A\ .\ x = y\ OO\ r\}$

lemma *OO-Sup*: $r\ OO\ (Sup\ A) = Sup\ \{x\ .\ \exists\ y{\in}A\ .\ x = r\ OO\ y\}$

lemma *OO-SUP*: $r\ OO\ (SUP\ n\ .\ A\ n) = (SUP\ n\ .\ r\ OO\ (A\ n))$

lemma *SUP-OO*: $(SUP\ n\ .\ A\ n)\ OO\ r = (SUP\ n\ .\ (A\ n)\ OO\ r)$

definition *InstFeedback* $r = (\lambda\ x\ uy\ .\ case\ x\ of\ \cdot \Rightarrow uy = \cdot \mid OK\ z \Rightarrow$
$(\exists\ n\ a\ .\ (a\ 0 = \bot) \wedge (\forall\ i < n\ .\ a\ i < a\ (Suc\ i)) \wedge (\forall\ i < n\ .\ \exists\ y\ .\ r\ (OK\ (a\ i, z))\ (OK\ (a\ (Suc\ i), y))) \wedge$
$((\exists\ y\ .\ r\ (OK\ (a\ n, z))\ (OK\ (a\ (Suc\ n), y)) \wedge a\ n = a\ (Suc\ n) \wedge uy = OK\ (a\ (Suc\ n), y)) \vee$
$(r\ (OK\ (a\ n, z))\ \cdot \wedge uy = \cdot))\ ))$

lemma *InstFeedback-alt*: $InstFeedback\ r = (\lambda\ x\ uy\ .\ case\ x\ of\ \cdot \Rightarrow uy = \cdot \mid OK\ z \Rightarrow$
$(\exists\ n\ a\ .\ (a\ 0 = \bot) \wedge (\forall\ i < n\ .\ a\ i < a\ (Suc\ i) \wedge (\exists\ y\ .\ r\ (OK\ (a\ i, z))\ (OK\ (a\ (Suc\ i), y)))) \wedge$
$r\ (OK\ (a\ n, z))\ uy \wedge (\exists\ y\ .\ uy = OK\ (a\ n, y) \vee uy = \cdot))$

definition *functional* $r\ f\ g = (\forall\ u\ x\ z\ .\ r\ (OK\ (u, x))\ z = (z = OK(f\ x\ u, g\ x\ u)))$

lemma *chain-power*: $a\ 0 = b \Longrightarrow \forall i{\leq}n.\ a\ (Suc\ i) = f\ (a\ i) \Longrightarrow i \leq Suc\ n \Longrightarrow a\ i = (f\ \hat{}\hat{}\ i)\ b$

theorem *InstFeedback-constructive*: *emono* $((f\ x)::'a::$*fin-cpo* $\Rightarrow 'a) \Longrightarrow$ *functional* $r\ f\ g \Longrightarrow$
$(InstFeedback\ r\ (OK\ x)\ uy) = (uy = OK\ (Lfp\ (f\ x),\ g\ x\ (Lfp\ (f\ x))))$

definition *InstFeedback-1* $r = (\lambda\ x\ uy\ .$ *case* x *of* $\cdot \Rightarrow uy = \cdot \mid OK\ z \Rightarrow$
$(\exists\ a\ .\ \bot < a \wedge (\exists\ y\ .\ r\ (OK\ (\bot, z))\ (OK\ (a, y))) \wedge r\ (OK\ (a, z))\ uy \wedge (\exists\ y\ .\ uy = OK\ (a, y) \vee uy = \cdot)\ )$
$\vee\ (r\ (OK\ (\bot, z))\ uy \wedge (\exists\ y\ .\ uy = OK\ (\bot, y) \vee uy = \cdot)))$

lemma [*simp*]: $(\bot < (a::'a::$*order-bot*$)) = (\bot \neq a)$

definition *unkn-mono* $r = (\forall\ a\ b\ x\ .\ (a::'a::$*order-bot*$) \leq b \longrightarrow (\forall\ z\ .\ r\ (OK\ (b, x))\ (OK\ z) \longrightarrow r\ (OK\ (a, x))\ (OK\ z)))$

lemma *unkn-mono-fb-fun*: *unkn-mono* $r \Longrightarrow$ *InstFeedback-1* $r =$ *InstFeedback* r

definition *fb-begin* $= (\lambda\ x\ ux\ .\ ux = (case\ x\ of\ \cdot \Rightarrow \cdot \mid OK\ x \Rightarrow OK\ (\bot, x)))$

definition *fb-a* $r = (\lambda\ ux\ ux'\ .\ (case\ ux\ of\ \cdot \Rightarrow ux' = \cdot \mid OK\ (u, x) \Rightarrow$
$(r\ (OK\ (u, x))\ \cdot \wedge\ ux' = \cdot) \vee (\exists\ u'\ y'\ .\ r\ (OK\ (u, x))\ (OK\ (u', y')) \wedge u < u' \wedge ux' = OK\ (u', x))))$

definition *fb-b* $r = (\lambda\ ux\ uy'\ .\ (case\ ux\ of\ \cdot \Rightarrow uy' = \cdot \mid OK\ (u, x) \Rightarrow$
$(r\ (OK\ (u, x))\ \cdot \wedge\ uy' = \cdot) \vee (\exists\ y'\ .\ r\ (OK\ (u, x))\ (OK\ (u, y')) \wedge uy' = OK\ (u, y'))))$

definition *fb-end* $= (\lambda\ uy\ y'\ .\ case\ uy\ of\ \cdot \Rightarrow y' = \cdot \mid OK\ (u, y) \Rightarrow (if\ maximal\ u\ then\ y' = OK\ y\ else\ y' = \cdot))$

definition *fb-hide* $r = (InstFeedback\ r)\ OO$ *fb-end*

definition *ff* $r = r\ \cdot\ \cdot$
definition *f-f* $r = (\forall\ x\ .\ r\ \cdot\ x \longrightarrow x = \cdot)$

lemma [*simp*]:$(case\ y\ of\ \cdot \Rightarrow \cdot = \cdot \mid OK\ (u, ya) \Rightarrow (maximal\ u \longrightarrow \cdot = OK\ ya) \wedge (\neg\ maximal\ u \longrightarrow \cdot = \cdot)) = (\forall\ u\ x\ .\ y = OK\ (u, x) \longrightarrow \neg\ maximal\ u)$

lemma [*simp*]: *InstFeedback-1* $r\ \cdot\ \cdot$

lemma [*simp*]: $(case\ y\ of\ \cdot \Rightarrow \cdot = \cdot \mid OK\ (u, v, x) \Rightarrow \cdot = OK\ (v, u, x)) = (y = \cdot)$

lemma *case-b-simp*: $(case\ b\ of\ \cdot \Rightarrow OK\ y = \cdot \mid OK\ (w, u, a) \Rightarrow OK\ y = OK\ ((u, w), a)) = (b \neq \cdot \wedge (case\ b\ of\ OK\ (w, u, a) \Rightarrow y = ((u, w), a)))$

lemma [*simp*]: $(x::'a::$*order-bot*$) \leq \bot \Longrightarrow x = \bot$

definition *mono-fail* $r = (\forall\ a\ b\ x.\ a \leq b \longrightarrow r\ (OK\ (a, x))\ \cdot \longrightarrow r\ (OK\ (b, x))\ \cdot)$

lemma *sconjunctive-comp-simp*: *sconjunctive* $S \Longrightarrow S \circ (INF\ n::nat.\ T\ n) = (INF\ n\ .\ S\ o\ (T\ n))$

lemma *sconj-star-a*: *sconjunctive* $S \Longrightarrow (INF\ n::nat.\ S\ \hat{}\hat{}\ n) \leq gfp\ (\lambda X.\ Skip \sqcap (S \circ X))$

lemma *mono-comp-simp*: *mono* $S \Longrightarrow T \leq T' \Longrightarrow S\ o\ T \leq S\ o\ T'$

lemma *sconj-star-b-aux*: $mono\ S \Longrightarrow u \leq Skip \Longrightarrow u \leq S \circ u \Longrightarrow u \leq S\ \hat{}\hat{}\ n$

lemma *sconj-star-b*: $mono\ S \Longrightarrow gfp\ (\lambda X.\ Skip \sqcap (S \circ X)) \leq (INF\ n{::}nat.\ S\ \hat{}\hat{}\ n)$

lemma *sconj-star*: $sconjunctive\ S \Longrightarrow gfp\ (\lambda X.\ Skip \sqcap (S \circ X)) = (INF\ n{::}nat.\ S\ \hat{}\hat{}\ n)$

lemma [*simp*]: $(case\ ya\ of\ \cdot \Rightarrow OK\ y = \cdot \mid OK\ z \Rightarrow p\ z) = (\exists\ z\ .\ ya = OK\ z \wedge p\ z)$

lemma [*simp*]: $((p \longrightarrow q) \wedge p) = (p \wedge q)$

lemma *relpowp-chain*: $\bigwedge x\ y\ .\ (R\ \hat{}\hat{}\ n)\ x\ y = (\exists\ a\ .\ (\forall\ i < n\ .\ R\ (a\ i)\ (a\ (Suc\ i))) \wedge x = a\ 0 \wedge y = a\ n)$

lemma [*simp*]: $fb\text{-}a\ r\ \cdot\ x = (x = \cdot)$

lemma [*simp*]: $fb\text{-}a\ r\ (OK\ (u,\ x))\ (OK\ (u',\ x')) = ((\exists\ y\ .\ r\ (OK\ (u,\ x))\ (OK\ (u',\ y))) \wedge u < u' \wedge x = x')$

lemma [*simp*]: $fb\text{-}a\ r\ (OK\ ux)\ \cdot = r\ (OK\ ux)\ \cdot$

lemma *fb-a-id*: $\bigwedge u\ x\ u'\ x'\ .\ (fb\text{-}a\ r\ \hat{}\hat{}\ n)\ (OK\ (u,\ x))\ (OK\ (u',\ x')) \Longrightarrow x = x'$

lemma *fb-a-id-a*: $(\forall i < n.\ fb\text{-}a\ r\ (a\ i)\ (a\ (Suc\ i))) \longrightarrow (\forall\ i \leq n\ .\ a\ i \neq \cdot \longrightarrow (snd\ (elem\ (a\ i))) = (snd\ (elem\ (a\ 0))))$

lemma *fb-a-id-b*: $(\forall i < n.\ fb\text{-}a\ r\ (a\ i)\ (a\ (Suc\ i))) \Longrightarrow (\forall\ i \leq n\ .\ a\ i \neq \cdot \longrightarrow snd\ (elem\ (a\ i)) = snd\ (elem\ (a\ 0)))$

lemma [*simp*]: $x < y \Longrightarrow x \neq \cdot$

lemma [*simp*]: $\bigwedge x\ .\ ((fb\text{-}a\ r)\ \hat{}\hat{}\ n)\ \cdot\ x = (x = \cdot)$

lemma *chain-fail*: $\bigwedge k\ .\ \forall i < n.\ fb\text{-}a\ r\ (a\ i)\ (a\ (Suc\ i)) \Longrightarrow k < n \Longrightarrow a\ (Suc\ k) = \cdot \Longrightarrow a\ n = \cdot$

lemma [*simp*]: $OK\ x < \cdot$

lemma *chain-not-fail*: $a\ 0 \neq \cdot \Longrightarrow \forall k.\ a\ (Suc\ k) = \cdot \longrightarrow k < n \longrightarrow (\exists j \leq k.\ a\ j = \cdot) \Longrightarrow (\forall\ i \leq n\ .\ a\ i \neq \cdot)$

lemma [*simp*]: $fb\text{-}b\ r\ (OK\ (u,\ x))\ (OK\ (u',\ y)) = (r\ (OK\ (u,\ x))\ (OK\ (u',\ y)) \wedge u = u')$

lemma [*simp*]: $fb\text{-}b\ r\ (OK\ (u,\ x))\ \cdot = r\ (OK\ (u,\ x))\ \cdot$

lemma [*simp*]: $fb\text{-}b\ r\ \cdot\ x = (x = \cdot)$

lemma *chain-all-fail*: $\bigwedge i\ .\ a\ (0{::}nat) = \cdot \Longrightarrow \forall i < n.\ fb\text{-}a\ r\ (a\ i)\ (a\ (Suc\ i)) \Longrightarrow i \leq n \Longrightarrow a\ i = \cdot$

theorem *InstFeedback-simp*: $InstFeedback\ r = fb\text{-}begin\ OO\ ((fb\text{-}a\ r)\ \hat{}{**})\ OO\ (fb\text{-}b\ r)$

lemma *SUP-pointwise*: $(\forall\ n\ .\ (S{::}'a \Rightarrow 'b{::}complete\text{-}lattice)\ n \leq S'\ n) \Longrightarrow (SUP\ n\ .\ S\ n) \leq (SUP\ n\ .\ S'\ n)$

lemma *INF-pointwise*: $(\forall\ n\ .\ (S{::}'a \Rightarrow 'b{::}complete\text{-}lattice)\ n \leq S'\ n) \Longrightarrow (INF\ n\ .\ S\ n) \leq (INF\ n\ .\ S'\ n)$

definition *faila r x* = ((*r* (*OK x*) •)::*bool*)
definition *rela r x y* = (*r* (*OK x*) (*OK y*))
definition *preca r* = −*faila r*

definition *wp r* = {.*preca r*.} *o* [:*rela r*:]

lemma (*wp r* ≤ *wp r'*) = ((∀ *x* . *r'* (*OK x*) • ⟶ *r* (*OK x*) •) ∧ (∀ *x*. ¬ *r* (*OK x*) • ⟶ (∀ *y*. *r'* (*OK x*) (*OK y*) ⟶ *r* (*OK x*) (*OK y*))))

definition *Fb-a S* = [:*u*,*x*⇝ (*u'*,*x'*), *x''* . *u'* = *u* ∧ *x'* = *x* ∧ *x''* = *x*:] *o* ((*S* ‖ [:*u*,*x*⇝*v*,*y* . *u* < *v*:]) ∗∗ *Skip*) *o* [:(*v*,*y*), *x* ⇝ *v'*,*x'* . *v'* = *v* ∧ *x'* = *x*:]

thm *fusion-spec*

thm *Prod-spec-Skip*

lemma *wp* (*fb-a r*) = *Fb-a* (*wp r*)

lemma *ff r* ⟹ (*wp r* ≤ *wp r'*) = (∀ *x* . *r x* • ∨ *r' x* ≤ *r x*)

lemma [*simp*]: *preca* (*op* =) = ⊤

lemma [*simp*]: (*rela* (*op* =)) = (*op* =)

lemma [*simp*]: *wp* (*op* =) = *Skip*

lemma *mono* (*wp r*)

definition *serial r r'* = (*r OO r'*)

lemma *pred-bot-comp*: *ff r* ⟹ *ff r'* ⟹ *preca* (*r OO r'*) = (λ*x*. *preca r x* ∧ (∀ *y*. *rela r x y* ⟶ *preca r' y*))

lemma *fb-a-not-fail-fail-simp*: *fb-a r* (*OK* (*u*, *x*)) • = (*r* (*OK* (*u*, *x*)) •)

lemma *fb-b-not-fail-simp*: *fb-b r* (*OK* (*u*, *x*)) (*OK* (*u'*, *y'*)) = (*u* = *u'* ∧ *r* (*OK* (*u*, *x*)) (*OK* (*u'*, *y'*)))

lemma *fb-b-fail-simp*: *fb-b r* (*OK* (*u*, *x*)) • = *r* (*OK* (*u*, *x*)) •

lemma *refine-fba-a*: *wp r* ≤ *wp r'* ⟹ *wp* (*fb-a r*) ≤ *wp* (*fb-a r'*)

lemma *refine-fba-b'*: *wp r* ≤ *wp r'* ⟹ *wp* (*fb-b r*) ≤ *wp* (*fb-b r'*)

lemma *rel-bot-comp*: (*preca r x* ∧ *rela* (*r OO r'*) *x y*) = (*preca r x* ∧ (*rela r OO rela r'*) *x y*)

lemma *prec-demonic*: {.*p* ⊓ *q*.} *o* [:*r*:] = {.*p* ⊓ *q*.} *o* [:*x*⇝*y* . *p x* ∧ *r x y*:]

lemma *wp-refine*: (*wp r* ≤ *wp r'*) = (*preca r* ≤ *preca r'* ∧ (∀ *x* . *preca r x* ⟶ *rela r' x* ≤ *rela r x*))

lemma *wp-comp*: *ff r* ⟹ *ff r'* ⟹ *wp* (*r OO r'*) = ((*wp r*) *o* (*wp r'*))

lemma *not-maximal-prod*: (¬ *maximal* (*a*, *b*)) = (¬ *maximal a* ∨ ¬ *maximal b*)

lemma [*simp*]: *ff fb-end*

lemma *refine-left*: $S \leq S' \Longrightarrow S\ o\ T \leq S'\ o\ T$

lemma *prec-SUP*: *preca* $(SUP\ n\ .\ r\ n) = (INF\ n\ .\ preca\ (r\ n))$

lemma *rel-SUP*: *rela* $(SUP\ n\ .\ r\ n) = (SUP\ n\ .\ rela\ (r\ n))$

lemma *INF-spec*: $(INF\ n\ .\ \{.p\ n.\}\ o\ [:(r\ n)::('a \Rightarrow 'b \Rightarrow bool):]) = \{.INF\ n\ .\ p\ n.\}\ o\ [:SUP\ n\ .\ r\ n:]$

lemma *wp-SUP*: *wp* $(SUP\ n\ .\ r\ n) = (INF\ n\ .\ wp\ (r\ n))$

thm *wp-def*

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

term $(f::'a \Rightarrow 'b)\ \hat{}\hat{}\ n$

thm *funpow-times-power*

lemma *le-power*: *mono* $g \Longrightarrow (f::'a::order \Rightarrow 'a::order) \leq g \Longrightarrow f\ \hat{}\hat{}\ n \leq g\ \hat{}\hat{}\ n$

lemma [*simp*]: *mono* $(wp\ r)$

lemma [*simp*]: *ff* $r \Longrightarrow ff\ ((r::'a\ fail\text{-}option \Rightarrow 'a\ fail\text{-}option \Rightarrow bool)\ \hat{}\hat{}\ n)$

lemma *wp-power*: *ff* $r \Longrightarrow wp\ ((r::'a\ fail\text{-}option \Rightarrow 'a\ fail\text{-}option \Rightarrow bool)\ \hat{}\hat{}\ n) = (wp\ r)\ \hat{}\hat{}\ n$

lemma *wp-power-refin*: *ff* $r \Longrightarrow ff\ r' \Longrightarrow wp\ (r::'a\ fail\text{-}option \Rightarrow 'a\ fail\text{-}option \Rightarrow bool) \leq wp\ r' \Longrightarrow wp\ (r\hat{}\hat{}n) \leq wp\ (r'\hat{}\hat{}n)$

thm *INF-lower*

lemma *wp-rt-refine*: *ff* $r \Longrightarrow ff\ r' \Longrightarrow wp\ r \leq wp\ r' \Longrightarrow wp\ (r\hat{}**) \leq wp\ (r'\hat{}**)$

lemma [*simp*]: *ff fb-begin*

lemma [*simp*]: *ff* $(fb\text{-}a\ r)$

lemma [*simp*]: *ff* $(fb\text{-}b\ r)$

lemma [*simp*]: *ff* $((fb\text{-}a\ r)^{**})$

lemma [*simp*]: *ff* $r \Longrightarrow ff\ r' \Longrightarrow ff\ (r\ OO\ r')$

theorem *InstFeedback-refine*: *ff* $r \Longrightarrow ff\ r' \Longrightarrow wp\ r \leq wp\ r' \Longrightarrow wp\ (InstFeedback\ r) \leq wp\ (InstFeedback\ r')$

lemma [*simp*]: *ff* $r \Longrightarrow ff\ (InstFeedback\ r)$

theorem *fb-hide-refine*: *ff* $r \Longrightarrow ff\ r' \Longrightarrow wp\ r \leq wp\ r' \Longrightarrow wp\ (fb\text{-}hide\ r) \leq wp\ (fb\text{-}hide\ r')$

definition *cross-prod* $r\ r' = (\lambda\ ux\ vy\ .\ (case\ ux\ of\ \cdot \Rightarrow vy = \cdot \mid OK\ (u::'a::order\text{-}bot,\ x) \Rightarrow (\exists\ v\ y\ .\ vy = OK\ (v,\ y) \wedge r\ (OK\ x)\ (OK\ v) \wedge r'\ (OK\ u)\ (OK\ y))$

$$\vee\ (vy = \cdot \wedge r\ (OK\ x)\ \cdot) \vee\ \ (vy = \cdot \wedge r'\ (OK\ u)\ \cdot)\ \ ))$$

definition *InstFeedback-cross-prod* $r\ r' = (\lambda\ x\ vy.\ (case\ x\ of\ \cdot \Rightarrow vy = \cdot \mid OK\ x \Rightarrow$
$(\exists\ v\ y\ .\ vy = OK\ (v,\ y) \wedge r\ (OK\ x)\ (OK\ v) \wedge r'\ (OK\ v)\ (OK\ y)) \vee$
$(vy = \cdot \wedge r\ (OK\ x)\ \cdot) \vee (\exists\ v\ .\ vy = \cdot \wedge r\ (OK\ x)\ (OK\ v) \wedge r'\ (OK\ v)\ \cdot\ )\ \ ))$

lemma [*simp*]: $(\cdot < x) = False$

type-synonym $('a,\ 'b)$ *fail-pair* $= (('a\ option) \times ('b))$ *fail-option*
type-synonym $('a,\ 'b,\ 'c)$ *fail-pair-rel* $= ('a,'c)$ *fail-pair* $\Rightarrow$ $('a,'b)$ *fail-pair* $\Rightarrow$ *bool*

lemma [*simp*]: *op* $= \sqcup$ *fba-a r* $\sqcup$ (*op* $=$ *OO fba-a r*) *OO fba-a r* $\sqcup$ ((*op* $=$ *OO fba-a r*) *OO fba-a r*) *OO fba-a r* $\leq$ (*SUP n. fba-a r* $\hat{}\hat{}$ *n*)

lemma *all-fail*: $\forall\, i<xb.$ *fb-a r* (*a i*) (*a* (*Suc i*)) $\Longrightarrow$ *a 0* $= \cdot \Longrightarrow \forall\ i \leq xb\ .\ a\ i = \cdot$

lemma *fba-a-pair*: (*fb-a* (*r* :: $('a,'b,'c)$ *fail-pair-rel*)) $\hat{}**$ $= ((op=) \sqcup$ *fb-a r* $\sqcup$ (*fb-a r*) $\hat{}\hat{}$ (*Suc* (*Suc 0*)))

lemma [*simp*]: *ff* (*cross-prod r r'*)

lemma [*simp*]: *fb-begin* $\cdot\ x = (x = \cdot)$

lemma [*simp*]: *InstFeedback-cross-prod r r'* $\cdot\ x = (x = \cdot)$

definition *complete* $r = (\forall\ x\ .\ \exists\ y\ .\ r\ x\ y)$
definition *fail-mono* $r = (\forall\ x\ y\ .\ x \leq y \wedge r\ x\ \cdot \longrightarrow r\ y\ \cdot)$

definition *unkn-not-fail* $r = (\neg\ r\ (OK\ \bot)\ \cdot)$

lemma [*simp*]: *unkn-not-fail r'* $\Longrightarrow$ *cross-prod r r'* (*OK* $(\bot,$ *x2*)) $\cdot \Longrightarrow$ *InstFeedback-cross-prod r r'* (*OK x2*) $\cdot$

lemma [*simp*]: *cross-prod r r'* (*OK* $(ab,$ *bb*)) (*OK* $(ab,$ *c*)) $\Longrightarrow$ *InstFeedback-cross-prod r r'* (*OK bb*) (*OK* $(ab,$ *c*))

lemma [*simp*]: *OK* $(\bot, \bot) <$ *OK* $(\bot,$ *Some a*)

lemma [*simp*]: *OK* $(\bot, \bot) <$ *OK* $(Some\ a, \bot)$

lemma [*simp*]: *OK* $(\bot, \bot) <$ *OK* $(Some\ a,$ *Some b*)

lemma [*simp*]: *OK* $(None,$ *None*$) <$ *OK* $(Some\ a,\ y)$

lemma *move-down*: $p \Longrightarrow p$

lemma [*simp*]: *None* $<$ *Some a*

lemma [*simp*]: $\bot <$ *Some a*

thm *InstFeedback-cross-prod-def*

thm *unkn-not-fail-def*

thm *complete-def*

lemma *f-f-fb-begin*: *f-f fb-begin*

lemma *f-f-fb-a*: *f-f* (*fb-a r*)

lemma *f-f-fb-b*: *f-f* (*fb-b r*)

lemma *f-f-comp*: *f-f r* $\Longrightarrow$ *f-f r'* $\Longrightarrow$ *f-f* (*r OO r'*)

lemma [*simp*]: (*fb-a r*)** $\cdot$ *x* = (*x* = $\cdot$)

lemma *f-f-InstFeedback*: *f-f* (*InstFeedback r*)

lemma *InstFeedback-cross-prod-aux*: *complete r'* $\Longrightarrow$ *unkn-not-fail r'* $\Longrightarrow$ *InstFeedback-cross-prod r r' x xa* $\Longrightarrow$ *InstFeedback* (*cross-prod r r'*) *x xa*

theorem *InstFeedback-cross-prod*: *complete r'* $\Longrightarrow$ *unkn-not-fail r'* $\Longrightarrow$ *InstFeedback* (*cross-prod r r'*) = *InstFeedback-cross-prod r r'*

lemma [*simp*]: *OK* (*Some a*, *None*) < *OK* (*Some a*, *Some aa*)

thm *fb-hide-def*
thm *fb-end-def*

definition *fb-end-ukn* = (λ*uy y'*. *case uy of* $\cdot$ $\Rightarrow$ *y'* = $\cdot$ | *OK* (*u*, *y*) $\Rightarrow$ *y'* = *OK y*)

definition *fb-hide-cross-prod r r'* = (λ *x y*. (*case x of* $\cdot$ $\Rightarrow$ *y* = $\cdot$ | *OK x* $\Rightarrow$
(∃ *v* . *r* (*OK x*) (*OK* (*Some v*)) ∧ *r'* (*OK* (*Some v*)) *y*) ∨ (*y* = $\cdot$ ∧ (*r* (*OK x*) $\cdot$ ∨ *r* (*OK x*) (*OK* ⊥)))))

lemma [*simp*]: *InstFeedback-cross-prod r r'* $\cdot$ *y* = (*y* = $\cdot$)

lemma [*simp*]: *ff r* $\Longrightarrow$ *f-f r* $\Longrightarrow$ (*r* $\cdot$ *x*) = (*x* = $\cdot$)

lemma *rel-union*: *rela* (*r* ⊔ *r'*) = *rela r* ⊔ *rela r'*

lemma *prec-union*: *preca* (*r* ⊔ *r'*) = *preca r* ⊓ *preca r'*

lemma *wp* (*r* ⊔ *r'*) = *wp r* ⊓ *wp r'*

lemma *chain-OK*: ⋀ *a' b'* . ∀ *i*<*n*. *aa i* < *aa* (*Suc i*) $\Longrightarrow$ *aa 0* = *OK* (*a*, *b*) $\Longrightarrow$ *aa n* = *OK* (*a'*, *b'*) $\Longrightarrow$ (∃ *u y* . ∀ *i* ≤ *n* . *aa i* = *OK* (*u i*, *y i*))

lemma [*simp*]: *maximal* (*None*) = *False*

lemma [*simp*]: *maximal u* = (*u* ≠ *None*)

lemma [*simp*]: *OK* (⊥, ⊥) ≤ *OK* (*a*, *b*)

thm *InstFeedback-cross-prod-def*

lemma *fb-hide-cross-proda*: *complete* $r' \Longrightarrow$ *unkn-not-fail* $r' \Longrightarrow$ *fb-hide* (*cross-prod* r r') x y = *fb-hide-cross-prod* r r' x y

6.1 Examples

definition *havoc* x y = (*maximal* $x \longrightarrow$ *maximal* y)

definition *EQ* = (λ *ux vy* . *vy* = (*case ux of* $\cdot \Rightarrow \cdot$ | *OK* ((*u*::'*a option*), *x*) $\Rightarrow$ *OK* (*u*, *u*)))

lemma [*simp*]: (*a*::'*a*::*order*) < *a* = *False*

lemma *fb-hide-fun-EQ*: *InstFeedback EQ x uy* = (*uy* = (*case x of* $\cdot \Rightarrow \cdot$ | - $\Rightarrow$ *OK* ($\bot$,$\bot$)))

lemma *fb-hide EQ x y* = (*y* = $\cdot$)

definition *TRUEa* = (λ *ux vy* . (*case ux of* $\cdot \Rightarrow$ *vy* = $\cdot$ | *OK* ((*u*::'*a option*), *x*) $\Rightarrow$ ($\exists$ *v* . *vy* = *OK* (*v*, *v*) $\wedge$ (*u* $\neq$ *None* $\longrightarrow$ *v* $\neq$ *None*))))

lemma *move-assumption*: *p* $\Longrightarrow$ *p*

lemma *fb-hide-fun-TRUEa*: *InstFeedback TRUEa x uy* = (*case x of* $\cdot \Rightarrow$ *uy* = $\cdot$ | - $\Rightarrow$ ($\exists$ *u* . *uy* = *OK* (*u*, *u*)))

lemma *fb-hide TRUEa x y* = (*case x of* $\cdot \Rightarrow$ *y* = $\cdot$ | - $\Rightarrow$ (*y* = $\cdot$ $\vee$ ($\exists$ *u* . *maximal u* $\wedge$ *y* = *OK u*)))

definition *TRUE* = (λ *ux vy* . (*case ux of* $\cdot \Rightarrow$ *vy* = $\cdot$ | *OK* ((*u*::'*a option*), *x*) $\Rightarrow$ ($\exists$ *u* . *vy* = *OK* (*u*, *u*))))

lemma *fb-hide-fun-TRUE*: *InstFeedback TRUE x uy* = (*case x of* $\cdot \Rightarrow$ *uy* = $\cdot$ | - $\Rightarrow$ ($\exists$ *u* . *uy* = *OK* (*u*, *u*)))

lemma *fb-hide TRUE x y* = (*case x of* $\cdot \Rightarrow$ *y* = $\cdot$ | - $\Rightarrow$ (*y* = $\cdot$ $\vee$ ($\exists$ *u* . *maximal u* $\wedge$ *y* = *OK u*)))

definition *NEQ* = (λ *ux vy* . (*case ux of* $\cdot \Rightarrow$ *vy* = $\cdot$ | *OK* (*u*, *x*) $\Rightarrow$
($\exists$ *v* . *vy* = *OK* (*v*, *v*) $\wedge$ ((*u* = *None* $\longrightarrow$ *v* = *None*) $\wedge$ (*u* $\neq$ *None* $\longrightarrow$ *u* $\neq$ *v*)))))

definition *NEQ2* = (λ *ux vy* . (*case ux of* $\cdot \Rightarrow$ *vy* = $\cdot$ | *OK* (*u*, *x*) $\Rightarrow$
($\exists$ *v* . *vy* = *OK* (*v*, *v*) $\wedge$ ((*u* = *None* $\longrightarrow$ *v* = *None*) $\wedge$ (*u* $\neq$ *None* $\longrightarrow$ *u* $\neq$ *v* $\wedge$ *v* $\neq$ *None*)))))

lemma *fb-hide-fun-NEQ2*: *InstFeedback NEQ2 x uy* = (*case x of* $\cdot \Rightarrow$ *uy* = $\cdot$ | - $\Rightarrow$ *uy* = *OK* (*None*, *None*))

lemma *fb-hide-fun-NEQ*: *InstFeedback NEQ x uy* = (*case x of* $\cdot \Rightarrow$ *uy* = $\cdot$ | - $\Rightarrow$ *uy* = *OK* (*None*, *None*))

lemma *fb-hide NEQ x y* = (*y* = $\cdot$)

lemma *fb-hide NEQ2 x y* = (*y* = $\cdot$)

definition *rel-bot-true* $r = (\forall\ x\ y\ .\ \neg\ maximal\ x \longrightarrow r\ x\ y)$
definition *rel-maximal* $r = (\forall\ x\ y\ .\ r\ x\ y \wedge maximal\ x \longrightarrow maximal\ y)$

definition *assert-rel* $p\ x\ y = (if\ p\ x\ then\ y = x\ else\ y = \bot)$

definition *comp-rel* $r\ r'\ x\ y = (if\ r\ x\ \bot\ then\ y = \bot\ else\ (\exists\ z\ .\ r\ x\ z \wedge r'\ z\ y))$

definition *AND* $x\ y = (case\ (x,y)\ of\ (Some\ a,\ Some\ b) \Rightarrow Some\ (a \wedge b)$
$|\ (None,\ Some\ False) \Rightarrow Some\ False\ |\ (Some\ False,\ None) \Rightarrow Some\ False\ |\ \text{-} \Rightarrow None)$

definition *AND-rel ux vy* $= (case\ ux\ of\ \cdot \Rightarrow vy = \cdot\ |\ OK\ (u,\ x) \Rightarrow vy = OK\ (AND\ u\ x,\ AND\ u\ x))$

lemma [*simp*]: *ff AND-rel*

lemma [*simp*]: $((None,\ Some\ a) \leq (None,\ None)) = False$

lemma [*simp*]: *AND-rel* $(OK\ (u,\ Some\ False))\ (OK\ (v,\ y)) = ((v = Some\ False) \wedge (y = Some\ False))$

lemma [*simp*]: *AND-rel* $(OK\ (Some\ False,\ u))\ (OK\ (v,\ y)) = ((v = Some\ False) \wedge (y = Some\ False))$

lemma *AND-comute*: $AND\ x\ y = AND\ y\ x$

lemma *AND-rel-comute*: *AND-rel* $(OK\ (x,\ y)) = AND\text{-}rel\ (OK\ (y,\ x))$

lemma [*simp*]: *AND-rel* $(OK\ x)\ \cdot = False$

lemma *fb-hide-fun-AND*: *InstFeedback AND-rel x uy* $= (case\ x\ of\ \cdot \Rightarrow uy = \cdot\ |\ OK\ (Some\ False) \Rightarrow uy = OK\ (Some\ False,\ Some\ False)\ |\ \text{-} \Rightarrow (uy = OK\ (\bot,\ \bot)))$

lemma *fb-hide AND-rel x y* $=\ (case\ x\ of\ \cdot \Rightarrow y = \cdot\ |\ OK\ (Some\ False) \Rightarrow y = OK\ (Some\ False)\ |\ \text{-} \Rightarrow y = \cdot)$

definition *AND-rel2a* $= (\lambda\ ((w,\ u),x)\ ((v,\ w'),\ y)\ .\ (v = AND\ u\ x) \wedge (w = w') \wedge (v = y))$

definition *AND-rel2 wux vwy* $= (case\ wux\ of\ \cdot \Rightarrow vwy = \cdot\ |\ OK\ ((w,\ u),\ x) \Rightarrow vwy = OK\ ((AND\ u\ x,\ w),\ AND\ u\ x))$

lemma [*simp*]: *ff AND-rel2*

lemma [*simp*]: *AND-rel2* $(OK\ ((w,\ u),\ Some\ False))\ (OK\ ((v,\ w'),\ c)) = (v = Some\ False \wedge w = w' \wedge Some\ False = c)$

lemma [*simp*]: *AND-rel2* $(OK\ (a,\ Some\ False))\ (OK\ (b,\ c)) = (fst\ b = Some\ False \wedge fst\ a = snd\ b \wedge Some\ False = c)$

thm *f-f-def*

lemma [*simp*]: $\bigwedge u\ x\ .\ (\bigwedge\ u\ .\ preca\ r\ (u,\ x)) \Longrightarrow (fb\text{-}a\ r\ \hat{}\hat{}\ n)\ (OK\ (u,\ x))\ \cdot = False$

lemma [*simp*]: *preca AND-rel2 x*

lemma [*simp*]:*AND-rel2* $(OK\ x)\ \cdot = False$

lemma [*simp*]: *AND None None = None*

lemma [*simp*]: *AND* $(Some\ True)\ (Some\ True) = (Some\ True)$

lemma [*simp*]: *AND* $(Some\ False)\ x = (Some\ False)$

lemma [*simp*]: *AND x* $(Some\ False) = (Some\ False)$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ None),\ None))\ (OK\ ((v,\ w),\ y)) = (v = None \wedge v = y \wedge v = w)$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ Some\ a),\ None))\ (OK\ ((u,\ w),\ y)) = (u = AND\ (Some\ a)\ None \wedge y = AND\ (Some\ a)\ None \wedge w = None)$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ None),\ Some\ True))\ (OK\ ((v,\ w),\ y)) = (v = AND\ None\ (Some\ True) \wedge y = AND\ None\ (Some\ True) \wedge w = None)$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ None),\ Some\ False))\ (OK\ ((v,\ w),\ y)) = (v = Some\ False \wedge y = Some\ False \wedge w = None)$

lemma [*simp*]: *AND-rel2* $(OK\ ((Some\ False,\ w),\ Some\ False))\ (OK\ ((v,\ w'),\ y)) = (v = Some\ False \wedge w' = Some\ False \wedge y = Some\ False)$

lemma *AND2-simp*: *AND-rel2* $(OK\ (((u::'a\ option),\ w),\ x))\ (OK\ ((v,\ w'),\ y)) = (v = AND\ w\ x \wedge w' = u \wedge y = AND\ w\ x)$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ None),\ x))\ (OK\ ((v,\ w),\ y)) = (v = AND\ None\ x \wedge w = None \wedge y = AND\ None\ x)$

lemma *chain-triple*: $x < y \Longrightarrow y < z \Longrightarrow z < w \Longrightarrow w < OK\ ((a::'a\ option\ ,\ b::'b\ option),\ c::'c\ option) \Longrightarrow False$

lemma [*simp*]: *AND-rel2* $(OK\ ((None,\ None),\ None))\ (OK\ ((v,\ w),\ y)) = (v = None \wedge w = None \wedge y = None)$

definition *rel-and a b* $= (if\ a = None\ then\ b = None \vee b = Some\ True\ else\ a = b)$

lemma [*simp*]: $\exists\, b\ ba.\ None = AND\ b\ ba$

lemma [*simp*]:$(\exists\, b.\ None = AND\ (Some\ True)\ b)$

lemma [*simp*]: $OK\ (\bot,\ \bot) < OK\ ((Some\ False,\ Some\ False),\ Some\ False)$

lemma [*simp*]: $OK\ (\bot,\ \bot) < OK\ ((Some\ True,\ Some\ True),\ Some\ True)$

lemma [*simp*]: $\exists\, b\ ba.\ Some\ False = AND\ b\ ba$

lemma [*simp*]: $\exists\, b\ ba.\ Some\ x = AND\ b\ ba$

lemma [*simp*]: $\exists\, ba.\ Some\ True = AND\ (Some\ True)\ ba$

lemma [*simp*]: $(((\bot), \bot) < (\bot, None)) = False$
lemma [*simp*]: $\exists\, b.\ Some\ False = AND\ b\ (Some\ True)$

lemma [*simp*]: $\exists\, b.\ Some\ True = AND\ b\ (Some\ True)$

lemma *OK-less-less*: $(OK\ x < OK\ y) = (x < y)$

lemma *fba-a-chain*: $\bigwedge u'.\ n > 0 \Longrightarrow (fb\text{-}a\ r\ \hat{}\hat{}\ n)\ (OK\ (u, x))\ (OK\ (u', x')) \Longrightarrow u < (u'::'a::order)$

lemma *fb-hide-and-eq*: $InstFeedback\ (AND\text{-}rel2)\ (OK\ x)\ (OK\ ((v, w), y)) \Longrightarrow v = y$

lemma [*simp*]: $InstFeedback\ (AND\text{-}rel2)\ (OK\ None)\ (OK\ ((None, Some\ False), None)) = False$

lemma [*simp*]: $InstFeedback\ AND\text{-}rel2\ (OK\ None)\ (OK\ ((None, None), None))$

lemma [*simp*]: $InstFeedback\ \ AND\text{-}rel2\ (OK\ None)\ (OK\ ((None, Some\ True), None)) = False$

lemma [*simp*]: $InstFeedback\ AND\text{-}rel2\ (OK\ None)\ (OK\ ((Some\ False, None), Some\ False)) = False$

lemma [*simp*]: $InstFeedback\ AND\text{-}rel2\ (OK\ None)\ (OK\ ((Some\ False, Some\ True), Some\ False)) = False$

lemma [*simp*]: $InstFeedback\ (AND\text{-}rel2)\ (OK\ None)\ (OK\ ((Some\ False, Some\ False), Some\ False)) = False$

lemma [*simp*]: $InstFeedback\ (AND\text{-}rel2)\ (OK\ None)\ (OK\ ((Some\ True, Some\ True), Some\ True)) = False$

lemma [*simp*]: $InstFeedback\ (\ AND\text{-}rel2)\ (OK\ None)\ (OK\ ((Some\ True, None), Some\ True)) = False$

lemma [*simp*]: $InstFeedback\ (\ AND\text{-}rel2)\ (OK\ None)\ (OK\ ((Some\ True, Some\ False), Some\ True)) = False$

lemma *fb-and-wire-bot*: $InstFeedback\ (AND\text{-}rel2)\ (OK\ None)\ (OK\ ((v, w), y)) = (v = y \wedge v = w \wedge v = None)$

lemma *fb-and-wire-false*: $InstFeedback\ (AND\text{-}rel2)\ (OK\ (Some\ False))\ (OK\ ((v, w), y)) = (v = Some\ False \wedge w = v \wedge y = v)$

lemma [*simp*]: $InstFeedback\ (AND\text{-}rel2)\ (OK\ (Some\ True))\ (OK\ ((None, Some\ False), None)) = False$

lemma [*simp*]: $(\exists\, b.\ None = AND\ b\ (Some\ True))$

lemma [*simp*]: $InstFeedback\ (\ AND\text{-}rel2)\ (OK\ (Some\ True))\ (OK\ ((None, None), None))$

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*None*, *Some True*), *None*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some False*, *None*), *Some False*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some False*, *Some True*), *Some False*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some False*, *Some False*), *Some False*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some True*, *Some True*), *Some True*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some True*, *None*), *Some True*)) = *False*

lemma [*simp*]: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*Some True*, *Some False*), *Some True*)) = *False*

lemma *fb-and-wire-true*: *InstFeedback* (*AND-rel2*) (*OK* (*Some True*)) (*OK* ((*v*, *w*), *y*)) = (*v* = *y* ∧ *v* = *w* ∧ *v* = *None*)

thm *fb-and-wire-true*
thm *fb-and-wire-false*
thm *fb-and-wire-bot*

lemma *InstFeedback* (*AND-rel2*) *x y* = (*case x of* · ⇒ *y* = · | *OK* (*Some False*) ⇒ *y* = *OK* ((*Some False*, *Some False*), *Some False*) | - ⇒ *y* = *OK* ((*None*, *None*), *None*))

definition *NonDet ux vy* = (*case ux of* · ⇒ *vy* = · | *OK* (*Some u*, *x*) ⇒
(*if u* = *2 then vy* = · *else*
vy = *OK*(*Some* (*x* + *1*), *x* + *1*) ∨ *vy* = *OK*(*Some* (*x* + *1*), *x* + *2*) ∨
vy = *OK*(*Some* (*x* + *2*), *x* + *2*) ∨ *vy* = *OK*(*Some* (*x* + *2*), *x* + *3*) ∨
vy = *OK*(*Some 6*, *6*) ∨ *vy* = *OK*(*Some 6*, *7*))
| *OK*(*None*, *x*) ⇒
vy = *OK*(*Some* (*x* + *1*), *x* + *1*) ∨ *vy* = *OK*(*Some* (*x* + *1*), *x* + *2*) ∨
vy = *OK*(*Some* (*x* + *2*), *x* + *2*) ∨ *vy* = *OK*(*Some* (*x* + *2*), *x* + *3*) ∨
vy = *OK*(*Some 7*, *7*) ∨ *vy* = *OK*(*Some 7*, *8*))

definition *InstFeedbackNonDet x vy* = (*case x of* · ⇒ *vy* = · |
OK a ⇒ (*a* = *Suc 0* ∧ *vy* = ·) ∨ (*a* = *0* ∧ *vy* = ·) ∨
(*a* ≠ *1* ∧ (*vy* = *OK*(*Some* (*a* + *1*), *a* + *1*) ∨ *vy* = *OK*(*Some* (*a* + *1*), *a* + *2*))) ∨
(*a* ≠ *0* ∧ (*vy* = *OK*(*Some* (*a* + *2*), *a* + *2*) ∨ *vy* = *OK*(*Some* (*a* + *2*), *a* + *3*))))

lemma *InstFeedbackNonDet-a*: *InstFeedback NonDet x vy* ⟹ *InstFeedbackNonDet x vy*

lemma *InstFeedbackNonDet-b*: *InstFeedbackNonDet x vy* ⟹ *InstFeedback NonDet x vy*

lemma *InstFeedbackNonDet*: *InstFeedback NonDet* = *InstFeedbackNonDet*

6.2 Associativity of Instantaneous Feedback

definition *adapt r a b* = (*case a of* $\cdot \Rightarrow b = \cdot$ | *OK* (*u*, (*v*, *x*)) ⇒
(∃ *u′ v′ y* . *r* (*OK* ((*u*,*v*), *x*)) (*OK* (((*u′*,*v′*),*y*))) ∧ *b* = *OK* (*u′*, (*v′*, *y*))) ∨ (*r* (*OK* ((*u*,*v*), *x*)) $\cdot$ ∧ *b* = $\cdot$))

definition *adapt-b a b* = (*case a of* $\cdot \Rightarrow b = \cdot$ | *OK* (*u*, (*v*, *x*)) ⇒ *b* = *OK* (*v*, (*u*, *x*)))

definition *adapt-c x y* = (*case x of* $\cdot \Rightarrow y = \cdot$ |
OK (*w*, (*u*, *a*)) ⇒ *y* = *OK* ((*u*, *w*), *a*))

definition *adapt-a x y* = (*case x of* $\cdot \Rightarrow y = \cdot$ | *OK* (*u*, (*v*, *x*)) ⇒ *y* = *OK* ((*u*, *v*), *x*))

lemma *ff r* ⟹ *f-f r* ⟹ *adapt r* = *adapt-a OO r OO adapt-a*$^{-1-1}$

lemma [*simp*]: *unkn-mono r* ⟹ *unkn-mono* (*adapt r*)

lemma [*simp*]: (*case y of* $\cdot \Rightarrow$ *OK* (*b*, *a*, *yaa*) = $\cdot$ | *OK* (*u*, *v*, *x*) ⇒ *OK* (*b*, *a*, *yaa*) = *OK* (*v*, *u*, *x*))
= (*y* = *OK* (*a*, *b*, *yaa*))

lemma [*simp*]: (*case y of* $\cdot \Rightarrow$ *OK* ((*a*, *b*), *yaa*) = $\cdot$ | *OK* (*w*, *u*, *aa*) ⇒ *OK* ((*a*, *b*), *yaa*) = *OK* ((*u*, *w*), *aa*))
= (*y* = *OK* (*b*, *a*, *yaa*))

lemma [*simp*]: (*case y of* $\cdot \Rightarrow \cdot = \cdot$ | *OK* (*w*, *u*, *a*) ⇒ $\cdot$ = *OK* ((*u*, *w*), *a*)) = (*y* = $\cdot$)

lemma [*simp*]: *unkn-mono r* ⟹ *r* (*OK* ((*a*, *b*), *x2*)) (*OK* ((*u*, *v*), *z*)) ⟹ *r* (*OK* ((⊥, ⊥), *x2*)) (*OK* ((*u*, *v*), *z*))

lemma [*simp*]: *unkn-mono r* ⟹ *r* (*OK* ((*a*, *b*), *x2*)) (*OK* ((*u*, *v*), *z*)) ⟹ *r* (*OK* ((⊥, *b*), *x2*)) (*OK* ((*u*, *v*), *z*))

lemma [*simp*]: *unkn-mono r* ⟹ *r* (*OK* ((*a*, *b*), *x2*)) (*OK* ((*u*, *v*), *z*)) ⟹ *r* (*OK* ((*a*, ⊥), *x2*)) (*OK* ((*u*, *v*), *z*))

lemma [*simp*]: *unkn-mono r* ⟹ *unkn-mono* (*InstFeedback* (*adapt r*) *OO adapt-b*)

term *InstFeedback* (*fb-fun* (*adapt r*) *OO adapt-b*) *OO adapt-c*

lemma *fb-hide-comp-aux*: *unkn-mono* (*InstFeedback* (*adapt r*) *OO adapt-b*) ⟹ *InstFeedback* (*InstFeedback* (*adapt r*) *OO adapt-b*) = *InstFeedback-1* (*InstFeedback* (*adapt r*) *OO adapt-b*)

lemma [*simp*]: *adapt r* $\cdot$ $\cdot$

lemma [*simp*]: *adapt-b* $\cdot$ $\cdot$

lemma [*simp*]: *adapt-c* $\cdot$ $\cdot$

lemma [*simp*]: *unkn-mono r* ⟹

$r\ (OK\ ((\bot,\ \bot),\ x2))\ (OK\ ((a,\ b),\ ya)) \Longrightarrow$
$r\ (OK\ ((a,\ b),\ x2))\ (OK\ ((a,\ b),\ yaa)) \Longrightarrow$
InstFeedback-1 $(adapt\ r)\ (OK\ (\bot,\ x2))\ (OK\ (a,\ b,\ yaa))$

lemma [*simp*]: *unkn-mono* $r \Longrightarrow$
$r\ (OK\ ((\bot,\ \bot),\ x2))\ (OK\ ((a,\ b),\ ya)) \Longrightarrow$
$r\ (OK\ ((a,\ b),\ x2))\ (OK\ ((a,\ b),\ yaa)) \Longrightarrow$
$\exists\, a\ ba.$ *InstFeedback-1* $(adapt\ r)\ (OK\ (\bot,\ x2))\ (OK\ (a,\ b,\ ba))$

lemma [*simp*]: *unkn-mono* $r \Longrightarrow$
$r\ (OK\ ((\bot,\ \bot),\ x2))\ (OK\ ((a,\ b),\ ya)) \Longrightarrow$
$r\ (OK\ ((a,\ b),\ x2))\ (OK\ ((a,\ b),\ yaa)) \Longrightarrow$
InstFeedback-1 $(adapt\ r)\ (OK\ (b,\ x2))\ (OK\ (a,\ b,\ yaa))$

definition *indep* $r = (\forall\ x\ y\ z\ z'\ .\ r\ (OK\ ((\bot,\bot),\ z))\ (OK\ ((x,\ y),\ z')) \longrightarrow$
$((\exists\ a\ .\ r\ (OK\ ((x,\ \bot),\ z))\ (OK\ ((x,\ y),\ a))) \wedge ((\exists\ a\ .\ r\ (OK\ ((\bot,\ y),\ z))\ (OK\ ((x,\ y),\ a))))))$

lemma *InstFeedback-assoc-fail-a*: *indep* $r \Longrightarrow$ *unkn-mono* $r \Longrightarrow$ *InstFeedback* $r\ x\ \cdot \Longrightarrow ((InstFeedback\ (InstFeedback\ (adapt\ r)\ OO\ adapt\text{-}b))\ OO\ adapt\text{-}c)\ x\ \cdot$

definition *indep-a* $r = (\forall\ x\ y\ y'\ a\ b\ a'\ b'\ .\ r\ (OK\ ((\bot,\ \bot),\ x))\ (OK\ ((a,\ b),\ y)) \wedge r\ (OK\ ((\bot,\ \bot),\ x))\ (OK\ ((a',\ b'),\ y'))$
$\longrightarrow (\exists\ z\ .\ r\ (OK\ ((\bot,\ \bot),x))\ (OK\ ((a,\ b'),\ z))))$

lemma *InstFeedback-assoc-fail-b*: *indep-a* $r \Longrightarrow$ *mono-fail* $r \Longrightarrow$ *unkn-mono* $r \Longrightarrow ((InstFeedback\ (InstFeedback\ (adapt\ r)\ OO\ adapt\text{-}b))\ OO\ adapt\text{-}c)\ x\ \cdot \Longrightarrow InstFeedback\ r\ x\ \cdot$

lemma *InstFeedback-assoc-OK*: *unkn-mono* $r \Longrightarrow InstFeedback\ r\ x\ (OK\ y) = ((InstFeedback\ (InstFeedback\ (adapt\ r)\ OO\ adapt\text{-}b))\ OO\ adapt\text{-}c)\ x\ (OK\ y)$

theorem *InstFeedback-assoc*: *indep* $r \Longrightarrow$ *indep-a* $r \Longrightarrow$ *mono-fail* $r \Longrightarrow$ *unkn-mono* $r \Longrightarrow$
$(InstFeedback\ (InstFeedback\ (adapt\ r)\ OO\ adapt\text{-}b))\ OO\ adapt\text{-}c = InstFeedback\ r$

definition *unkn-mono-up* $r = (\forall\ a\ b\ x\ u\ y.\ a \leq b \wedge r\ (OK\ (a,\ x))\ (OK\ (u,\ y)) \longrightarrow ((\exists\ v\ .\ u \leq v \wedge r\ (OK\ (b,\ x))\ (OK\ (v,\ y))) \vee r\ (OK\ (b,\ x))\ \cdot))$
lemma *unkn-mono-up-A*: *unkn-mono-up* $r \Longrightarrow a \leq b \Longrightarrow r\ (OK\ (a,\ x))\ (OK\ (u,\ y)) \Longrightarrow ((\exists\ v\ .\ u \leq v \wedge r\ (OK\ (b,\ x))\ (OK\ (v,\ y))) \vee r\ (OK\ (b,\ x))\ \cdot)$
lemma *unkn-mono-a-A*: *unkn-mono* $r \Longrightarrow a \leq b \Longrightarrow r\ (OK\ (b,\ x))\ (OK\ z) \Longrightarrow r\ (OK\ (a,\ x))\ (OK\ z)$

lemma *feedback-comp-fail-Z*: *mono-fail* $(r::((\,'a\ option \times\ 'b\ option) \times\ 'c)\ fail\text{-}option \Rightarrow (((\,'a\ option \times\ 'b\ option) \times\ 'd)\ fail\text{-}option) \Rightarrow bool)$
$\Longrightarrow$ *unkn-mono* $r \Longrightarrow$ *unkn-mono-up* $r \Longrightarrow InstFeedback\ r\ x\ \cdot \Longrightarrow ((InstFeedback\ (InstFeedback\ (adapt\ r)\ OO\ adapt\text{-}b))\ OO\ adapt\text{-}c)\ x\ \cdot$

end

7 Formalizing Simulink in RCRS

7.1 Types for Simulink Modeling Elements

theory *SimulinkTypes* **imports** *Real Transcendental*
begin

instantiation *bool*::*zero*

begin
definition *zero-bool-def*[*simp*]: *0* = *False*
instance
end

instantiation *bool*::*one*
begin
definition *one-bool-def*[*simp*]: *1* = *True*
instance
end

instantiation *bool*::*plus*
begin
definition *plus-bool-def*[*simp*]: (*a*::*bool*) + *b* = (*a* ∨ *b*)
instance
end

instance *bool*::*semigroup-add*

instantiation *bool*::*numeral*
begin
instance
lemma [*simp*]: *numeral a* = *True*
end

instantiation *bool*::*divide*
begin
definition *divide-bool-def*[*simp*]: (*a*::*bool*) *div b* = (*a* ∧ *b*)
instance
end

instantiation *bool*::*inverse*
begin
definition *inverse-bool-def*[*simp*]: *inverse* (*a*::*bool*) = *a*
instance
end

class *s-pi* =
fixes *s-pi*::*'a*

instantiation *real*::*s-pi*
begin
definition *s-pi-real-def*[*simp*]: *s-pi* = *pi*
instance
end

class *s-sqrt* =
fixes *s-sqrt*:: *'a* ⇒ *'a*

instantiation *real*::*s-sqrt*
begin
definition *s-sqrt-real-def*[*simp*]: *s-sqrt* = *sqrt*
instance
end

class *s-abs* =
fixes *s-abs*:: *'a* ⇒ *'a*

instantiation *real*::*s-abs*
begin
definition *s-abs-real-def*[*simp*]: *s-abs* = (*abs*::*real* ⇒ *real*)
instance
end

class *s-exp* =
fixes *s-exp*:: *'a* ⇒ *'a*

instantiation *real*::*s-exp*
begin
definition *s-exp-real-def*[*simp*]: *s-exp* = (*exp* :: *real* ⇒ *real*)
instance
end

class *s-ln* =
fixes *s-ln*:: *'a* ⇒ *'a*

instantiation *real*::*s-ln*
begin
definition *s-ln-real-def*[*simp*]: *s-ln* = (*ln*::*real* ⇒ *real*)
instance
end

class *s-sin* =
fixes *s-sin*:: *'a* ⇒ *'a*

class *s-cos* =
fixes *s-cos*:: *'a* ⇒ *'a*

instantiation *real*::*s-sin*
begin

definition *s-sin-real-def*[*simp*]: *s-sin* = (*sin* :: *real* ⇒ *real*)
instance
end

instantiation *real*::*s-cos*
begin

definition *s-cos-real-def*[*simp*]: *s-cos* = (*cos* :: *real* ⇒ *real*)
instance
end

definition *MyIf*:: *bool* ⇒ *'a* ⇒ *'a* ⇒ *'a* ((*If* (-)/ *Then* (-)/ *Else* (-)) [*0*, *0*, *10*] *10*) **where**
(*If b Then x Else y*) = (*if b then x else y*)

lemma *If-prod*: (*If b Then* (*x*, *y*) *Else* (*u*, *v*)) = ((*If b Then x Else u*), (*If b Then y Else v*))

lemma *If-eq*: $(If\ b\ Then\ x\ Else\ x) = x$

class *simulink* = *minus* + *uminus* + *numeral* + *power* + *zero* + *ord* + *s-sqrt* + *s-abs* + *s-exp* + *s-ln* + *s-sin* + *s-cos* + *s-pi* + *inverse* +
assumes *numeral-nzero*[*simp*]: *numeral* $n \neq 0$
begin
lemma [*simp*]: $(1 = 0) = False$
lemma [*simp*]: $(0 = 1) = False$

lemma [*simp*]: $((if\ b\ then\ (1::'a)\ else\ 0) = 0) = (\neg\ b)$

lemma [*simp*]: $((if\ b\ then\ (1::'a)\ else\ 0) = 1) = b$

end

lemma [*simp*]: $(if\ b\ then\ True\ else\ False) = b$

instantiation *real*::*simulink*
begin
instance
end

instantiation *nat*::*simulink*
begin
instance
end

instantiation *bool*::*simulink*
begin
instance
end

definition *is-eq-num* $x\ y = (if\ x = y\ then\ 1\ else\ 0)$
lemma *is-eq-num-a*: $((is\text{-}eq\text{-}num\ x\ y)::bool) = (x = y)$
lemmas *is-eq-num-simp* [*simp*] = *is-eq-num-a is-eq-num-def*

definition *is-neq-num* $x\ y = (if\ x \neq y\ then\ 1\ else\ 0)$
lemma *is-neq-num-a*: $((is\text{-}neq\text{-}num\ x\ y)::bool) = (x \neq y)$
lemmas *is-neq-num-simp* [*simp*] = *is-neq-num-a is-neq-num-def*

definition *is-less-num* $x\ y = (if\ x < y\ then\ 1\ else\ 0)$
lemma *is-less-num-a*: $((is\text{-}less\text{-}num\ x\ y)::bool) = (x < y)$
lemmas *is-less-num-simp* [*simp*] = *is-less-num-a is-less-num-def*

definition *is-less-eq-num* $x\ y = (if\ x \leq y\ then\ 1\ else\ 0)$
lemma *is-less-eq-num-a*: $((is\text{-}less\text{-}eq\text{-}num\ x\ y)::bool) = (x \leq y)$
lemmas *is-less-eq-num-simp* [*simp*] = *is-less-eq-num-a is-less-eq-num-def*

definition *is-gt-num* $x\ y = (if\ x > y\ then\ 1\ else\ 0)$
lemma *is-gt-num-a*: $((is\text{-}gt\text{-}num\ x\ y)::bool) = (x > y)$

lemmas *is-gt-num-simp* [*simp*] = *is-gt-num-a is-gt-num-def*

definition *is-ge-num x y* = (*if x* ≥ *y then 1 else 0*)
lemma *is-ge-num-a*: ((*is-ge-num x y*)::*bool*) = (*x* ≥ *y*)

lemmas *is-ge-num-simp* [*simp*] = *is-ge-num-a is-ge-num-def*

consts *conversion* :: *'a* ⇒ *'b*

overloading
conversion-id ≡ *conversion*:: *'a* ⇒ *'a* (**unchecked**)
conversion-bool-real ≡ *conversion*:: *bool* ⇒ *real* (**unchecked**)
conversion-bool-nat ≡ *conversion*:: *bool* ⇒ *nat* (**unchecked**)
conversion-real-bool ≡ *conversion*:: *real* ⇒ *bool* (**unchecked**)
begin
definition [*simp*]: *conversion-id a* = *a*
definition [*simp*]: *conversion-bool-real* (*b*::*bool*) = (*if b then* (*1*::*real*) *else 0*)
definition [*simp*]: *conversion-bool-nat* (*b*::*bool*) = (*if b then* (*1*::*nat*) *else 0*)
definition [*simp*]: *conversion-real-bool* (*x*::*real*) = (*x* ≠ *0*)
end

end

7.2 Formalization of Simulink Blocks as Predicate Transformers

theory *Simulink*
imports *Complex-Main ../Feedback/TransitionFeedback SimulinkTypes*
begin

declare *comp-skip* [*simp del*]
declare *skip-comp* [*simp del*]
declare *prod-skip-skip* [*simp del*]
declare *fail-comp* [*simp del*]

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

definition *UnitVal* = ()

definition *Constant c* = [: *x*::*unit* ⇝ *y*. *y* = *c*:]

lemma *Constant-func*: *Constant c* = [− *x* ⇝ *c*−]

definition *Inport* = *Skip*

definition *Gain k* = [:*x* ⇝ *y*. *y* = *x* ∗ *k*:]

lemma *Gain-func*: *Gain k* = [− *x* ⇝ *x* ∗ *k*−]

definition *Square* = [:*x* ⇝ *y*. *y* = *x* ∗ *x*:]

lemma *Square-func*: *Square* = [− *x* ⇝ *x* ∗ *x*−]

definition *Power* = [: (*x*, *y*) ⇝ *z*. *z* = *x* ^ *y*:]

lemma *Power-func*: *Power* = [− *x*, *y* ⇝ *x* ^ *y*−]

definition *Power10* = [: *x* ⇝ *y*. *y* = *10* ^ *x*:]

lemma *Power10-func*: *Power10* = [− *x* ⇝ *10* ^ *x*−]

definition *Exp* = [: *x* ⇝ *y*. *y* = *s-exp x* :]

lemma *Exp-func*: *Exp* = [− *x* ⇝ *s-exp x*−]

definition *Ln* = [: *x* ⇝ *y*. *y* = *s-ln x*:]

lemma *Ln-func*: *Ln* = [− *x* ⇝ *s-ln x*−]

definition *Sqrt* = {. *x*. *x* ≥ *0* .} ∘ [:*x* ⇝ *y*. *y* = *s-sqrt x*:]

lemma *Sqrt-func*: *Sqrt* = {. *x*. *x* ≥ *0* .} ∘ [− *x* ⇝ *s-sqrt x*−]

definition *Outport* = *Skip*

definition *Scope* = *Skip*

definition *Terminator* = [: $x \rightsquigarrow (u::unit).\ True$:]

lemma *Terminator-func*: *Terminator* = [− $x \rightsquigarrow ()$−]

definition *Integrator dt* = [:$(x,s) \rightsquigarrow (y,\ s').\ y = s \wedge s' = s + x * dt$:]

lemma *Integrator-func*: *Integrator dt* = [−$x,\ s \rightsquigarrow s,\ s + x * dt$−]

definition *IntegratorA* = [:$s \rightsquigarrow y.\ y = s$:]

lemma *IntegratorA-func*: *IntegratorA* = [−*id*−]

definition *IntegratorB dt* = [:$(x,s) \rightsquigarrow s'.\ s' = s + x * dt$:]

lemma *IntegratorB-func*: *IntegratorB dt* = [− $x,\ s \rightsquigarrow s + x * dt$−]

definition *IntegratorLimit high low dt* = [: $(x,\ s) \rightsquigarrow (y,\ s').\ y = s \wedge s' = (If\ s + x * dt > high\ Then\ high\ Else\ If\ s + x * dt < low\ Then\ low\ Else\ s + x * dt)$:]

lemma *IntegratorLimit-func* : *IntegratorLimit high low dt* = [− $x,\ s \rightsquigarrow s,\ If\ s + x * dt > high\ Then\ high\ Else\ If\ s + x * dt < low\ Then\ low\ Else\ s + x * dt$ −]

definition *IntegratorLimitA* = [:$s \rightsquigarrow y.\ y = s$:]

lemma *IntegratorLimitA-func*: *IntegratorLimitA* = [− *id* −]

definition *IntegratorLimitB high low dt* = [: $(x,\ s) \rightsquigarrow y.\ y = (If\ s + x * dt > high\ Then\ high\ Else\ If\ s + x * dt < low\ Then\ low\ Else\ s + x * dt)$:]

lemma *IntegratorLimitB-func*: *IntegratorLimitB high low dt* = [− $x,\ s \rightsquigarrow If\ s + x * dt > high\ Then\ high\ Else\ If\ s + x * dt < low\ Then\ low\ Else\ s + x * dt$ −]

definition *Saturation low-limit high-limit* = [: $x \rightsquigarrow y.$
y = (*If x* < *low-limit Then low-limit*
Else If x > *high-limit Then high-limit*
Else x):]

lemma *Saturation-func*: *Saturation low-limit high-limit* =
[− $x \rightsquigarrow$ *If* $x <$ *low-limit Then low-limit*
Else If $x >$ *high-limit Then high-limit*
Else x−]

definition *Relay low-limit high-limit value-low value-high* = [: x, $s \rightsquigarrow y$, s' .
y = (*If high-limit* $\leq x$ *Then value-high*
Else (*If* $x \leq$ *low-limit Then value-low Else* s))
$\wedge$ $s' = y$:]

lemma *Relay-func*: *Relay low-limit high-limit value-low value-high* =
[− x, $s \rightsquigarrow$ *If high-limit* $\leq x$ *Then value-high Else If* $x \leq$ *low-limit Then value-low Else* s,
If high-limit $\leq x$ *Then value-high Else If* $x \leq$ *low-limit Then value-low Else* s−]

definition *RelayA low-limit high-limit value-low value-high* = [: x, $s \rightsquigarrow y$.
y = (*If high-limit* $\leq x$ *Then value-high*
Else (*If* $x \leq$ *low-limit Then value-low Else* s)) :]

lemma *RelayA-func*: *RelayA low-limit high-limit value-low value-high* =
[− x, $s \rightsquigarrow$ *If high-limit* $\leq x$ *Then value-high Else If* $x \leq$ *low-limit Then value-low Else* s−]

definition *RelayB low-limit high-limit value-low value-high* = [: x, $s \rightsquigarrow s'$.
s' = (*If high-limit* $\leq x$ *Then value-high*
Else (*If* $x \leq$ *low-limit Then value-low Else* s)) :]

lemma *RelayB-func*: *RelayB low-limit high-limit value-low value-high* =
[− x, $s \rightsquigarrow$ *If high-limit* $\leq x$ *Then value-high Else If* $x \leq$ *low-limit Then value-low Else* s−]

definition *PulseGenerator period phase-delay pulse-width amplitude dt* = [: $(i,c) \rightsquigarrow y$, i',c'. i'=i+1
$\wedge$
(*If* $(i * dt <$ *phase-delay*) *Then* $(y = 0 \wedge c' = 0)$ *Else*
If $(i * dt \geq$ *phase-delay* $\wedge (c * dt) <$ (*pulse-width* $*$ *period*) $\wedge$ (*pulse-width* $*$ *period*) $<$ *period*)
Then (y=*amplitude* $\wedge$ $(c' = c + 1)$) *Else*
If $(i * dt \geq$ *phase-delay* $\wedge (c * dt) \geq$ (*pulse-width* $*$ *period*) $\wedge (c * dt) <$ (*period* $-$ *dt*) $\wedge$ (*pulse-width*
$*$ *period*) $<$ *period*) *Then* (y=0 $\wedge$ $(c' = c + 1)$)
Else $(c' = 0 \wedge y = 0)$):]

lemma *PulseGenerator-func*: *PulseGenerator period phase-delay pulse-width amplitude dt* =
[− i, $c \rightsquigarrow$
If $(i * dt <$ *phase-delay*) *Then* $(0, i + 1, 0)$ *Else*
If $(i * dt \geq$ *phase-delay* $\wedge (c * dt) <$ (*pulse-width* $*$ *period*) $\wedge$ (*pulse-width* $*$ *period*) $<$ *period*)
Then (*amplitude*, i+1, $c + 1$) *Else*
If $(i * dt \geq$ *phase-delay* $\wedge (c * dt) \geq$ (*pulse-width* $*$ *period*) $\wedge (c * dt) <$ (*period* $-$ *dt*) $\wedge$ (*pulse-width*
$*$ *period*) $<$ *period*) *Then* $(0, i$+1, $c + 1)$

Else $(0,\ i{+}1,\ 0)-]$

definition *PulseGeneratorA period phase-delay pulse-width amplitude dt* = [: $(i,c) \rightsquigarrow y$.
(*If* $(i * dt < \text{phase-delay})$ *Then* $y = 0$ *Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) < (\text{pulse-width} * \text{period}) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then* $y = \text{amplitude}$
Else $y = 0$):]

lemma *PulseGeneratorA-func* : *PulseGeneratorA period phase-delay pulse-width amplitude dt* =
[− $i,\ c \rightsquigarrow$
If $(i * dt < \text{phase-delay})$ *Then 0 Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) < (\text{pulse-width} * \text{period}) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then amplitude Else 0* −]

definition *PulseGeneratorB* = [: $i \rightsquigarrow i'.\ i' = i + 1$:]

lemma *PulseGeneratorB-func*: *PulseGeneratorB* = [− $i \rightsquigarrow i + 1$ −]

definition *PulseGeneratorC period phase-delay pulse-width dt* = [: $(i,c) \rightsquigarrow c'$.
(*If* $(i * dt < \text{phase-delay})$ *Then* $c' = 0$ *Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) < (\text{pulse-width} * \text{period}) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then* $c' = c + 1$ *Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) \geq (\text{pulse-width} * \text{period}) \wedge (c * dt) < (\text{period} - dt) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then* $c' = c + 1$
Else $c' = 0$) :]

lemma *PulseGeneratorC-func*: *PulseGeneratorC period phase-delay pulse-width dt* =
[− $i,\ c \rightsquigarrow$
If $(i * dt < \text{phase-delay})$ *Then 0 Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) < (\text{pulse-width} * \text{period}) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then* $c + 1$ *Else*
If $(i * dt \geq \text{phase-delay} \wedge (c * dt) \geq (\text{pulse-width} * \text{period}) \wedge (c * dt) < (\text{period} - dt) \wedge (\text{pulse-width} * \text{period}) < \text{period})$ *Then* $c + 1$
Else 0−]

definition *PulseGeneratorS period phase-delay pulse-width amplitude dt* = [: $t \rightsquigarrow y,\ t'$.
(*If* $(t < \text{phase-delay})$ *Then* $(y = 0 \wedge t' = t + dt)$ *Else*
If $t - \text{phase-delay} < \text{period} * \text{pulse-width} / 100$ *Then* $(y = \text{amplitude} \wedge t' = t + dt)$ *Else*
If $t - \text{phase-delay} < \text{period}$ *Then* $(y = 0 \wedge t' = t + dt)$
Else $(y = \text{amplitude} \wedge t' = t + dt - \text{period})$):]

lemma *PulseGeneratorS-func*: *PulseGeneratorS period phase-delay pulse-width amplitude dt* = [− $t \rightsquigarrow$
If $(t < \text{phase-delay})$ *Then* $(0,\ t + dt)$ *Else*
If $t - \text{phase-delay} < \text{period} * \text{pulse-width} / 100$ *Then* $(\text{amplitude},\ t + dt)$ *Else*
If $t - \text{phase-delay} < \text{period}$ *Then* $(0,\ t + dt)$
Else $(\text{amplitude},\ t + dt - \text{period})$ −]

definition *PulseGeneratorSA period phase-delay pulse-width amplitude dt = PulseGeneratorS period phase-delay pulse-width amplitude dt o* [:*y*, *t* ⇝ *y′* . *y* = *y′*:]

lemma *PulseGeneratorSA-func*: *PulseGeneratorSA period phase-delay pulse-width amplitude dt* = [− *t* ⇝
If (*t* < *phase-delay*) *Then 0 Else*
If t − *phase-delay* < *period* ∗ *pulse-width* / *100 Then amplitude Else*
If t − *phase-delay* < *period Then 0*
Else amplitude −]

thm *PulseGeneratorS-def*

definition *PulseGeneratorSB period phase-delay pulse-width dt* = [: *t* ⇝ *t′*.
(*If* (*t* < *phase-delay*) *Then t′* = *t* + *dt Else*
If t − *phase-delay* < *period* ∗ *pulse-width* / *100 Then t′* = *t* + *dt Else*
If t − *phase-delay* < *period Then t′* = *t* + *dt*
Else t′ = *t* + *dt* − *period*) :]

lemma *PulseGeneratorSB-func*: *PulseGeneratorSB period phase-delay pulse-width dt* = [− λ *t* .
(*If* (*t* < *phase-delay*) *Then t* + *dt Else*
If t − *phase-delay* < *period* ∗ *pulse-width* / *100 Then t* + *dt Else*
If t − *phase-delay* < *period Then t* + *dt*
Else t + *dt* − *period*) −]

lemma *PulseGeneratorSB-func-real*[*simp*]: *0* ≤ *phase-delay* ⟹ *0* < *period* ⟹ *0* < *pulse-width* ⟹ *pulse-width* < *100* ⟹
(λ (*t*::*real*) .
(*If* (*t* < *phase-delay*) *Then t* + *dt Else*
If t − *phase-delay* < *period* ∗ *pulse-width* / *100 Then t* + *dt Else*
If t − *phase-delay* < *period Then t* + *dt*
Else t + *dt* − *period*))
= (λ (*t*::*real*) . (*If t* − *phase-delay* < *period Then t* + *dt Else t* + *dt* − *period*))

definition *Step step-time initial-value final-value dt* = [:*i* ⇝ *y*, *i′*. *i′* = *i* + *1* ∧
y = (*If* (*i* ∗ *dt*) < *step-time Then initial-value Else final-value*):]

lemma *Step-func*: *Step step-time initial-value final-value dt* = [− *i* ⇝ *If* (*i* ∗ *dt*) < *step-time Then initial-value Else final-value*, *i*+*1*−]

definition *StepA step-time initial-value final-value dt* = [:*i* ⇝ *y*.
y = (*If* (*i* ∗ *dt*) < *step-time Then initial-value Else final-value*):]

lemma *StepA-func*: *StepA step-time initial-value final-value dt* = [− *i* ⇝ *If* (*i* ∗ *dt*) < *step-time Then initial-value Else final-value*−]

definition *StepB* = [:$i \rightsquigarrow i'.\ i' = i + 1$:]

lemma *StepB-func*: *StepB* = [− $i \rightsquigarrow i+1$−]

definition *StepT step-time initial-value final-value dt* = [:$t \rightsquigarrow y,\ t'.\ t' = t + dt \wedge$
y = (*If* $t <$ *step-time Then initial-value Else final-value*):]

lemma *StepT-func*: *StepT step-time initial-value final-value dt* = [− $t \rightsquigarrow$ *If* $t <$ *step-time Then initial-value Else final-value*, $t + dt$−]

definition *StepTA step-time initial-value final-value dt* = [:$t \rightsquigarrow y.$
y = (*If* $t <$ *step-time Then initial-value Else final-value*):]

lemma *StepTA-func*: *StepTA step-time initial-value final-value dt* = [− $t \rightsquigarrow$ *If* $t <$ *step-time Then initial-value Else final-value* −]

definition *StepTB dt* = [:$t \rightsquigarrow t'.\ t' = t + dt$:]

lemma *StepTB-func*: *StepTB dt* = [− $t \rightsquigarrow t + dt$−]

definition *TransferFcn k a dt* = [:$(x,\ i,\ s) \rightsquigarrow (y,\ i',\ s').\ y = (s * s\text{-}exp(a * i * dt) + k * x * s\text{-}exp(a * (i + 1) * dt) * dt)\ /\ s\text{-}exp(a * (i + 1) * dt) \wedge$
$i' = i + 1 \wedge s' = y$:]

lemma *TransferFcn-func*: *TransferFcn k a dt* = [− $x,\ i,\ s \rightsquigarrow (s * s\text{-}exp(a * i * dt) + k * x * s\text{-}exp(a * (i + 1) * dt) * dt)\ /\ s\text{-}exp(a * (i + 1) * dt)$,
$i+1,\ (s * s\text{-}exp(a * i * dt) + k * x * s\text{-}exp(a * (i + 1) * dt) * dt)\ /\ s\text{-}exp(a * (i + 1) * dt)$−]

definition *TransferFcnA k a dt* = [: $(x,\ i,\ s) \rightsquigarrow y.\ y = (s * s\text{-}exp(a * i * dt) + k * x * s\text{-}exp(a * (i + 1) * dt) * dt)\ /\ s\text{-}exp(a * (i + 1) * dt)$:]

lemma *TransferFcnA-func*: *TransferFcnA k a dt* = [− $x,\ i,\ s \rightsquigarrow (s * s\text{-}exp(a * i * dt) + k * x * s\text{-}exp(a * (i + 1) * dt) * dt)\ /\ s\text{-}exp(a * (i + 1) * dt)$−]

definition *TransferFcnB* = [: $i \rightsquigarrow i'.\ i' = i + 1$:]

lemma *TransferFcnB-func*: *TransferFcnB* = [− $i \rightsquigarrow i+ 1$−]

definition *TransferTFcn k a dt* = [:(*x*, *t*, *s*) ⇝ (*y*, *t′*, *s′*). *y* = (*s* ∗ *s-exp*(*a* ∗ *t*) + *k* ∗ *x* ∗ *s-exp*(*a* ∗ (*t* + *dt*)) ∗ *dt*) / *s-exp*(*a* ∗ (*t* + *dt*)) ∧
t′ = *t* + *dt* ∧ *s′* = *y*:]

lemma *TransferTFcn-func*: *TransferTFcn k a dt* = [− *x*, *t*, *s* ⇝ (*s* ∗ *s-exp*(*a* ∗ *t*) + *k* ∗ *x* ∗ *s-exp*(*a* ∗ (*t* + *dt*)) ∗ *dt*) / *s-exp*(*a* ∗ (*t* + *dt*)),
t + *dt*, (*s* ∗ *s-exp*(*a* ∗ *t*) + *k* ∗ *x* ∗ *s-exp*(*a* ∗ (*t* + *dt*)) ∗ *dt*) / *s-exp*(*a* ∗ (*t* + *dt*))−]

definition *TransferTFcnA k a dt* = [: (*x*, *t*, *s*) ⇝ *y*. *y* = (*s* ∗ *s-exp*(*a* ∗ *t*) + *k* ∗ *x* ∗ *s-exp*(*a* ∗ (*t* + *dt*)) ∗ *dt*) / *s-exp*(*a* ∗ (*t* + *dt*)) :]

lemma *TransferTFcnA-func*: *TransferTFcnA k a dt* = [− *x*, *t*, *s* ⇝ (*s* ∗ *s-exp*(*a* ∗ *t*) + *k* ∗ *x* ∗ *s-exp*(*a* ∗ (*t* + *dt*)) ∗ *dt*) / *s-exp*(*a* ∗ (*t* + *dt*))−]

definition *TransferTFcnB dt* = [: *t* ⇝ *t′*. *t′* = *t* + *dt*:]

lemma *TransferTFcnB-func*: *TransferTFcnB dt* = [− *t* ⇝ *t* + *dt*−]

definition *SinWave amplitude frequency phase bias dt* = [: *i* ⇝ (*y*, *i′*). *y* = *amplitude* ∗ *s-sin*(*frequency* ∗ *i* ∗ *dt* + *phase*) + *bias* ∧ *i′* = *i* + *1* :]

lemma *SinWave-func*: *SinWave amplitude frequency phase bias dt* = [− *i* ⇝ *amplitude* ∗ *s-sin*(*frequency* ∗ *i* ∗ *dt* + *phase*) + *bias*, *i*+*1*−]

definition *SinWaveA amplitude frequency phase bias dt* = [: *i* ⇝ *y*. *y* = *amplitude* ∗ *s-sin*(*frequency* ∗ *i* ∗ *dt* + *phase*) + *bias* :]

lemma *SinWaveA-func* : *SinWaveA amplitude frequency phase bias dt* = [− *i* ⇝ *amplitude* ∗ *s-sin*(*frequency* ∗ *i* ∗ *dt* + *phase*) + *bias* −]

definition *SinWaveB* = [: *i* ⇝ *i′*. *i′* = *i* + *1* :]

lemma *SinWaveB-func* : *SinWaveB* = [− *i* ⇝ *i* + *1* −]

definition *SinWaveT amplitude frequency phase bias dt* = [: *t* ⇝ (*y*, *t′*). *y* = *amplitude* ∗ *s-sin*(*frequency* ∗ *t* + *phase*) + *bias* ∧ *t′* = *t* + *dt* :]

lemma *SinWaveT-func*: *SinWaveT amplitude frequency phase bias dt* = [− *t* ⇝ *amplitude* ∗ *s-sin*(*frequency* ∗ *t* + *phase*) + *bias*, *t* + *dt*−]

definition *SinWaveTA amplitude frequency phase bias dt* = [: *t* ⇝ *y*. *y* = *amplitude* ∗ *s-sin*(*frequency*

$* \ t \ + \ phase) \ + \ bias \ :]$

lemma *SinWaveTA-func* : *SinWaveTA amplitude frequency phase bias dt* = $[-$ *t* $\rightsquigarrow$ *amplitude* $*$ *s-sin*(*frequency* $*$ *t* + *phase*) + *bias* $-]$

definition *SinWaveTB dt* = $[:$ *t* $\rightsquigarrow$ *t*$'$. *t*$'$ = *t* + *dt* $:]$

lemma *SinWaveTB-func* : *SinWaveTB dt* = $[-$ *t* $\rightsquigarrow$ *t* + *dt* $-]$

fun *MIN*:: *'a*::*ord list* $\Rightarrow$ *'a* **where**
MIN [] = *Eps* $\top$ |
MIN [*x*] = *x* |
MIN (*x* # *xs*) = *min x* (*MIN xs*)

fun *MAX*:: *'a*::*ord list* $\Rightarrow$ *'a* **where**
MAX [] = *Eps* $\top$ |
MAX [*x*] = *x* |
MAX (*x* # *xs*) = *max x* (*MAX xs*)

definition *slope-val x xi xj yi yj* = (*yj* − *yi*) $*$ (*x* − *xi*) / (*xj* − *xi*) + *yi*

definition *siggen-square x* = (*If s-sin x* < *0 Then* (−*1*::*'a*::*simulink*) *Else* (*1*::*'a*::*simulink*))

lemmas *additional-simps* =

slope-val-def siggen-square-def MIN.simps MAX.simps

lemmas *basic-block-rel-simps* =
Gain-def Square-def Power-def Power10-def Exp-def Ln-def Sqrt-def Constant-def Saturation-def

Relay-def Integrator-def
PulseGenerator-def Step-def TransferFcn-def
Scope-def Outport-def Inport-def
IntegratorA-def IntegratorB-def Terminator-def SinWave-def SinWaveA-def SinWaveB-def IntegratorLimit-def IntegratorLimitA-def IntegratorLimitB-def

lemmas *basic-block-func-simps* =
Gain-func Square-func Power-func Power10-func Exp-func Ln-func Sqrt-func Constant-func Saturation-func

Relay-func RelayA-func RelayB-func
Integrator-func IntegratorA-func IntegratorB-func
PulseGenerator-func PulseGeneratorA-func PulseGeneratorB-func PulseGeneratorC-func

PulseGeneratorS-func PulseGeneratorSA-func PulseGeneratorSB-func
TransferFcn-func TransferFcnA-func TransferFcnB-func
TransferTFcn-func TransferTFcnA-func TransferTFcnB-func
Scope-def Outport-def Inport-def
Step-func StepA-func StepB-func
StepT-func StepTA-func StepTB-func
Terminator-func
SinWave-func SinWaveA-func SinWaveB-func
SinWaveT-func SinWaveTA-func SinWaveTB-func
IntegratorLimit-func IntegratorLimitA-func IntegratorLimitB-func

lemmas *comp-rel-simps* = *Prod-spec-Skip Prod-Skip-spec Prod-demonic-skip Prod-skip-demonic Prod-demonic Prod-spec-demonic Prod-demonic-spec*
comp-assoc [*THEN sym*] *demonic-demonic comp-demonic-demonic assert-assert-comp comp-demonic-assert demonic-assert-comp*
OO-def Prod-spec Fail-assert fail-assert-demonic fail-comp
prod-skip-skip skip-comp comp-skip prod-fail fail-prod
update-demonic-comp demonic-update-comp comp-update-demonic comp-demonic-update

lemmas *comp-func-simps* =

prod-update prod-update-skip prod-skip-update
prod-assert-update-skip prod-skip-assert-update
Prod-assert-skip Prod-skip-assert prod-assert-update
prod-assert-assert-update prod-assert-update-assert
prod-update-assert-update prod-assert-update-update
comp-update-update comp-update-assert update-assert-comp
assert-assert-comp-pred
update-comp comp-assoc [*THEN sym*]
Fail-def fail-comp update-fail assert-fail prod-fail fail-prod
prod-skip-skip skip-comp comp-skip

lemmas *refinement-simps* = *assert-demonic-refinement spec-demonic-refinement*

lemmas *simulink-simps* = *basic-block-func-simps comp-func-simps*

lemmas *comp-var-simps* = *demonic-def assert-def le-fun-def Prod-spec-Skip Prod-Skip-spec Prod-demonic-skip Prod-skip-demonic Prod-demonic Prod-spec-demonic Prod-demonic-spec*
comp-assoc [*THEN sym*] *demonic-demonic comp-demonic-demonic assert-assert-comp comp-demonic-assert demonic-assert-comp OO-def Prod-spec Fail-assert*

lemmas *fail-simps* = *fail-def demonic-def Prod-spec-Skip Prod-Skip-spec Prod-demonic-skip Prod-skip-demonic assert-def le-fun-def Prod-demonic Prod-spec-demonic Prod-demonic-spec*
comp-assoc [*THEN sym*] *demonic-demonic comp-demonic-demonic assert-assert-comp comp-demonic-assert demonic-assert-comp OO-def Prod-spec Fail-assert*

lemmas *prec-simps* = *prec-def fail-def demonic-def Prod-spec-Skip Prod-Skip-spec Prod-demonic-skip Prod-skip-demonic assert-def le-fun-def Prod-spec-demonic Prod-demonic-spec*
comp-assoc [*THEN sym*] *demonic-demonic comp-demonic-demonic assert-assert-comp comp-demonic-assert demonic-assert-comp OO-def Prod-demonic Prod-spec Fail-assert*

lemmas *rel-simps* = *rel-def demonic-def Prod-spec-Skip Prod-Skip-spec Prod-demonic-skip Prod-skip-demonic assert-def le-fun-def Prod-demonic Prod-spec-demonic Prod-demonic-spec*
comp-assoc [*THEN sym*] *demonic-demonic comp-demonic-demonic assert-assert-comp comp-demonic-assert demonic-assert-comp OO-def Prod-spec Fail-assert*

lemmas *sconjunctive-simps* = *sconjunctive-simp-a sconjunctive-simp-b sconjunctive-simp-c*

lemmas *feedback-rel-simps* = *feedback-simp-a feedback-simp-b feedback-simp-bot*

lemmas *feedback-func-simps* = *feedback-update-simp-aaa feedback-update-simp-bbb feedback-simp-bot*

lemmas *feedbackless-func-simps* = *feedbackless-update-simp-aaa feedbackless-update-simp-bbb feedback-simp-bot*

lemma [*simp*]: $(\exists\ x\ y\ z\ .\ x = f\ y\ z)$

lemma [*simp*]: $(\exists\ x\ y\ z\ .\ f\ y\ z = x)$

lemma [*simp*]: $(\exists\ x\ y\ \ .\ x = f\ y\ )$

lemma [*simp*]: $(\exists\ x\ y\ \ .\ f\ y = x)$

lemma [*simp*]: $(\forall\, x{::}real.\ \neg\ 0 \leq x) = False$

lemma [*simp*]: $Ex\ (op \leq (0{::}real)) = True$

lemma [*simp*]: $(\exists\ a\ b\ .\ a + b = (x{::}'a{::}group\text{-}add)) = True$

lemma *common-imp-right-a*[*simp*]: $((p \longrightarrow (a \wedge b)) \wedge (\neg\ p \longrightarrow (c \wedge b))) = (((p \longrightarrow a) \wedge (\neg\ p \longrightarrow c)) \wedge b)$

lemma *common-imp-right-b*[*simp*]: $((\neg\ p \longrightarrow (a \wedge b)) \wedge (p \longrightarrow (c \wedge b))) = (((\neg\ p \longrightarrow a) \wedge (p \longrightarrow c)) \wedge b)$

lemma *common-imp-left-a* [*simp*]: $((p \longrightarrow b \wedge a) \wedge (\neg\ p \longrightarrow b \wedge c)) = (b \wedge (p \longrightarrow a) \wedge (\neg\ p \longrightarrow c))$

lemma *common-imp-left-b* [*simp*]: $((\neg\ p \longrightarrow b \wedge a) \wedge (p \longrightarrow b \wedge c)) = (b \wedge (\neg\ p \longrightarrow a) \wedge (p \longrightarrow c))$

lemma *common-dimp*: $((p \longrightarrow (q \longrightarrow a)) \wedge (r \longrightarrow (q \longrightarrow b))) = (q \longrightarrow ((p \longrightarrow a) \wedge (r \longrightarrow b)))$

lemma *fst-case-prod-eqa*: $(\bigwedge x\ y\ .\ fst\ (f1\ x\ y) = fst\ (f2\ x\ y)) \Longrightarrow fst\ (case\text{-}prod\ f1\ p) = fst\ (case\text{-}prod\ f2\ p)$

lemma *fst-case-prod-eqa-x*: $(\bigwedge x\ y\ .\ f\ (f1\ x\ y) = f\ (f2\ x\ y)) \Longrightarrow f\ (case\text{-}prod\ f1\ p) = f\ (case\text{-}prod\ f2\ p)$

lemma *fst-case-prod-eq*: $fst\ (f1\ (fst\ p1)\ (snd\ p1)) = fst\ (f2\ (fst\ p2)\ (snd\ p2)) \Longrightarrow fst\ (case\text{-}prod\ f1\ p1) = fst\ (case\text{-}prod\ f2\ p2)$

lemma *fst-case-prod-eqc*: $(\bigwedge z\ .\ fst\ (f1\ u\ z) = fst\ (f2\ u'\ z)) \Longrightarrow fst\ (case\text{-}prod\ f1\ (u,\ x)) = fst\ (case\text{-}prod\ f2\ (u',\ x))$

lemma *fst-case-prod-eqd*: $(\bigwedge y\ z\ .\ fst\ (f1\ y\ z) = fst\ (f2\ y\ z)) \Longrightarrow fst\ (case\text{-}prod\ f1\ x) = fst\ (case\text{-}prod\ f2\ x)$

definition $Snd = snd$

lemma *fst-case-prod-eqb*: $(fst\ (case\text{-}prod\ f1\ p1) = fst\ (case\text{-}prod\ f2\ p2)) = (fst\ (f1\ (fst\ p1)\ (Snd\ p1)) = fst\ (f2\ (fst\ p2)\ (Snd\ p2)))$

lemma *fst-case-prod-eqb-a*: $(fst\ (case\text{-}prod\ f1\ (u,\ x)) = fst\ (case\text{-}prod\ f2\ (v,\ x))) = (fst\ (f1\ u\ x) = fst\ (f2\ v\ x))$

lemma *fst-case-prod-eqb-b*: $(fst\ (case\text{-}prod\ f1\ p) = fst\ (case\text{-}prod\ f2\ p)) = (fst\ (f1\ (fst\ p)\ (Snd\ p)) = fst\ (f2\ (fst\ p)\ (Snd\ p)))$

definition $FstA = fst$

lemma *Fst-simp*: $FstA\ (x,y) = x$

lemma *fst-case-prod-eqc-a*: $(fst\ (case\text{-}prod\ f1\ (u,\ x)) = fst\ (case\text{-}prod\ f2\ (v,\ x))) = (FstA\ (f1\ u\ x) = FstA\ (f2\ v\ x))$

lemma *fst-case-prod-eqc-b*: $(FstA\ (case\text{-}prod\ f1\ p) = FstA\ (case\text{-}prod\ f2\ q)) = (FstA\ (f1\ (fst\ p)\ (Snd\ p)) = FstA\ (f2\ (fst\ q)\ (Snd\ q)))$

lemma *Snd-simp*: $Snd\ (x,\ y) = y$

lemma *fst-case-prod-eqb-x*: $(f\ (case\text{-}prod\ f1\ p1) = f\ (case\text{-}prod\ f2\ p2)) = (f\ (f1\ (fst\ p1)\ (Snd\ p1)) = f\ (f2\ (fst\ p2)\ (Snd\ p2)))$

lemma *fst-case-prod-eqba*: $(\forall\ x\ .\ fst\ (case\text{-}prod\ f1\ x) = fst\ (case\text{-}prod\ f2\ x)) = (\forall\ x\ y\ .\ fst\ (f1\ x\ y) = fst\ (f2\ x\ y))$

lemma [*simp*]: $(p \wedge (p \longrightarrow q)) = (p \wedge q)$

lemma [*simp*]: $(\forall\ x.\ x \neq y) = False$

lemma [*simp*]: $(\forall\ x.\ y \neq x) = False$

lemma [*simp*]: $(\exists\ x{::}real.\ y \neq x) = True$

lemma [*simp*]: $(\exists\ x{::}real.\ x \neq y) = True$

lemma *rel-if-expr-1*: $p\ x\ z \Longrightarrow p\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (b \vee\ p\ y\ z)$

lemma *rel-if-expr-2*: $p\ y\ z \Longrightarrow p\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (\neg\ b \vee p\ x\ z)$

lemma *rel-if-not-expr-1*: $\neg\ p\ x\ z \Longrightarrow p\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (\neg\ b \wedge p\ y\ z)$

lemma *rel-if-not-expr-2*: $\neg\ p\ y\ z \Longrightarrow p\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (b \wedge p\ x\ z)$

lemma *rel-expr-if-1*: $p\ z\ x \Longrightarrow p\ z\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (b \vee\ p\ z\ y)$

lemma *rel-expr-if-2*: $p\ z\ y \Longrightarrow p\ z\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\neg\ b \vee p\ z\ x)$

lemma *rel-expr-if-not-1*: $\neg\ p\ z\ x \Longrightarrow p\ z\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\neg\ b \wedge p\ z\ y)$

lemma *rel-expr-if-not-2*: $\neg\ p\ z\ y \Longrightarrow p\ z\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (b \wedge p\ z\ x)$

lemma *if-not*: $(\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ b\ \mathit{then}\ y\ \mathit{else}\ x)$

lemma *rel-not-if-expr-1*: $p\ y\ z \Longrightarrow p\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (b \vee p\ x\ z)$

lemma *rel-not-if-expr-2*: $p\ x\ z \Longrightarrow p\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (\neg b \vee\ p\ y\ z)$

lemma *rel-not-if-not-expr-1*: $\neg\ p\ y\ z \Longrightarrow p\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (\neg\ b \wedge p\ x\ z)$

lemma *rel-not-if-not-expr-2*: $\neg\ p\ x\ z \Longrightarrow p\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y)\ z = (b \wedge p\ y\ z)$

lemma *rel-expr-not-if-1*: $p\ z\ y \Longrightarrow p\ z\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (b \vee p\ z\ x)$

lemma *rel-expr-not-if-2*: $p\ z\ x \Longrightarrow p\ z\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\neg\ b \vee p\ z\ y)$

lemma *rel-expr-not-if-not-1*: $\neg\ p\ z\ y \Longrightarrow p\ z\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\neg\ b \wedge p\ z\ x)$

lemma *rel-expr-not-if-not-2*: $\neg\ p\ z\ x \Longrightarrow p\ z\ (\mathit{if}\ \neg\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (b \wedge p\ z\ y)$

lemma *not-inf*: $(\neg\ (x{::}\mathit{real}) < y) = (y \leq x)$

lemmas *if-simps* = *rel-if-expr-1 rel-if-expr-2 rel-if-not-expr-1 rel-if-not-expr-2 rel-expr-if-1 rel-expr-if-2 rel-expr-if-not-1 rel-expr-if-not-2*
rel-not-if-expr-1 rel-not-if-expr-2 rel-not-if-not-expr-1 rel-not-if-not-expr-2 rel-expr-not-if-1 rel-expr-not-if-2 rel-expr-not-if-not-1 rel-expr-not-if-not-2
if-not not-inf MyIf-def

end

7.3 Automated Simplification

theory *SimplifyRCRS* **imports** *Simulink*
keywords *simplify-RCRS simplify-RCRS-f* :: *thy-decl*
begin

thm *update-assert-comp*

definition *prod-fun f g* $= (\lambda\ (x,\ y)\ .\ (f\ x,\ g\ y))$
definition *prod-prec p q* $= (\lambda\ (x,\ y)\ .\ p\ x \wedge q\ y)$

lemma *asseert-update-comp*: $(\bigwedge x\ .\ let\ y = f\ x\ in\ p''\ x = (p\ x \wedge p'\ y) \wedge f''\ x = f'\ y) \Longrightarrow (\{.p.\}\ o\ [-f-])\ o\ (\{.p'.\}\ o\ [-f'-]) = \{.p''.\}\ o\ [-f''-]$

lemma *asseert-update-comp-abs-aux*: $p'' = p \sqcap (p'\ o\ f) \Longrightarrow\ f'' = f'\ o\ f \Longrightarrow (\{.p.\}\ o\ [-f-])\ o\ (\{.p'.\}\ o\ [-f'-]) = \{.p''.\}\ o\ [-f''-]$

lemma *asseert-update-comp-abs*: $p \sqcap (p'\ o\ f) \equiv p'' \Longrightarrow\ f'\ o\ f \equiv f'' \Longrightarrow (\{.p.\}\ o\ [-f-])\ o\ (\{.p'.\}\ o\ [-f'-]) = \{.p''.\}\ o\ [-f''-]$

lemma *asseert-update-prod-abs*: *prod-prec p p'* $\equiv p'' \Longrightarrow$ *prod-fun f f'* $\equiv f'' \Longrightarrow (\{.p.\}\ o\ [-f-])\ **\ (\{.p'.\}\ o\ [-f'-]) = \{.p''.\}\ o\ [-f''-]$

thm *If-prod*

term *Product-Type.prod.case-prod*

lemma *case-prod f* $(a,\ b) = f\ a\ b$

thm *Product-Type.case-prod-conv*

declare [[*show-sorts*]]

lemma *case-prod-eta-eq-sym*: $f \equiv (\lambda\ (x,\ y)\ .\ f\ (x,\ y))$

thm *Product-Type.case-prod-eta*

term $T\ ((x,y)\ ,z) = (x+y,x+z)$

definition *TtestTerm* $x \equiv x + 3$

definition *TTtestTerm* $\equiv (\lambda\ (x,\ (u,v),\ y)\ .\ (x,\ x+y,\ u+v))$

lemma *TT-simp*: *TTtestTerm* $(x,\ (u,v),\ y) \equiv (x,\ x\ +\ y,\ u+v)$

lemma *TTa-simp*: $(G \equiv$ *TTtestTerm*$) \Longrightarrow (G\ (x,\ (u,v),\ y) \equiv (x,\ x\ +\ y,\ u+v))$

thm *TtestTerm-def* [*of x*]

lemmas *T-inst* = *TtestTerm-def* [*of x*]

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

thm *cond-case-prod-eta*

thm *case-prod-eta*

thm *eta-contract-eq*

lemma *remove-aux-var*: $(\bigwedge X\ .\ X \equiv A \Longrightarrow X \equiv B) \Longrightarrow (A \equiv B)$

thm *Product-Type.case-prod-eta*

thm *cond-case-prod-eta*

declare [[*eta-contract=false*]]

lemma $(\{.(x,y).\ y{\neq}0.\}\ o\ [-\lambda(x,y).\ x/y-])\ o\ (\{.z.\ z{\geq}0.\}\ o\ [-\lambda z.\ sqrt\ z-]) = \{.\ (\lambda(x,\ y).\ y \neq 0) \sqcap ((\lambda z.\ z{\geq}0) \circ (\lambda(x,\ y).\ x\ /\ y))\ .\} \circ [-(\lambda z.\ sqrt\ z)\ o\ (\lambda(x,\ y).\ x\ /\ y)-]$

definition $dup\ y = (y,y)$

lemma $(snd\ o\ f\ o\ Pair\ (g\ x\ y))\ y = (snd\ o\ f\ o\ (prod\text{-}fun\ (g\ x)\ id)\ o\ dup)\ y$

lemma *feedback-asseert-update-abs-aux*: $g = (\lambda\ x\ .\ fst\ o\ f\ o\ Pair\ x) \Longrightarrow (\bigwedge x\ x'\ .\ g\ x = g\ x') \Longrightarrow\ snd\ o\ (f\ o\ (prod\text{-}fun\ (g\ x)\ id\ o\ dup)) = f' \Longrightarrow$
$p\ o\ (prod\text{-}fun\ (g\ x)\ id\ o\ dup) = p' \Longrightarrow feedback\ (\{.p.\}\ o\ [-f-]) = \{.p'.\}\ o\ [-f'-]$

lemma *feedback-asseert-update-abs*: $(\lambda\ x\ .\ fst\ o\ f\ o\ Pair\ x) \equiv g \Longrightarrow (\bigwedge x\ x'\ .\ g\ x \equiv g\ x') \Longrightarrow\ snd\ o\ (f\ o\ (prod\text{-}fun\ (g\ x)\ id\ o\ dup)) \equiv f' \Longrightarrow$
$p\ o\ (prod\text{-}fun\ (g\ x)\ id\ o\ dup) \equiv p' \Longrightarrow feedback\ (\{.p.\}\ o\ [-f-]) = \{.p'.\}\ o\ [-f'-]$

declare [[*eta-contract = false*]]

thm *eta-contract-eq*

thm *transitive*

lemma *Skip-th*: $\top \equiv p \Longrightarrow id \equiv f \Longrightarrow Skip = \{.p.\}\ o\ [-f-]$

lemma *Fail-th*: $\bot \equiv p \Longrightarrow f \equiv f \Longrightarrow \bot = \{.p.\}\ o\ [-f-]$

lemma *assert-th*: $p \equiv p' \Longrightarrow id \equiv f \Longrightarrow \{.p.\} = \{.p'.\}\ o\ [-f-]$

lemma *update-eq*: $\top \equiv p \Longrightarrow f \equiv g \Longrightarrow [-f-] = \ \{.p.\}\ o\ [-g-]$

lemma *demonic-eq*: $\top \equiv p \Longrightarrow r \equiv r' \Longrightarrow [:r:] = \ \{.p.\}\ o\ [:r':]$

lemma *assert-update-eq*: $p \equiv q \Longrightarrow f \equiv g \Longrightarrow \{.p.\}\ o\ [-f-] = \ \{.q.\}\ o\ [-g-]$

lemma *assert-demonic-eq*: $p \equiv q \Longrightarrow r \equiv r' \Longrightarrow \{.p.\}\ o\ [:r:] = \ \{.q.\}\ o\ [:r':]$

lemma *prec-simp-rel*: $((p \Longrightarrow r) \equiv (p \Longrightarrow r')) \Longrightarrow p \wedge r \equiv p \wedge r'$

lemma $((p \Longrightarrow r) \equiv Trueprop\ True) \Longrightarrow p \wedge r \equiv p$

definition *inter-pre-rel p r x y* = $(p\ x \wedge r\ x\ y)$

lemma *prop-eq-true*: $X \equiv True \Longrightarrow X$

lemma *inter-pre-rel-sym*: $(p\ x \wedge r\ x\ y) = inter\text{-}pre\text{-}rel\ p\ r\ x\ y$

theorem *assert-simp-demonic-eq*: $p \equiv p' \Longrightarrow inter\text{-}pre\text{-}rel\ p'\ r \equiv inter\text{-}pre\text{-}rel\ p'\ r' \Longrightarrow \{.p.\}\ o\ [:r:] = \{.p'.\}\ o\ [:r':]$

lemma *feedback-cong*: $B = A \Longrightarrow feedback\ A = F \Longrightarrow feedback\ B = F$

lemma *comp-cong*: $S = A \Longrightarrow T = B \Longrightarrow A\ o\ B = F \Longrightarrow S\ o\ T = F$

lemma *prod-cong*: $S = A \Longrightarrow T = B \Longrightarrow A\ **\ B = F \Longrightarrow S\ **\ T = F$

lemma *eq-eq-tran*: $a = b \Longrightarrow b \equiv c \Longrightarrow c = d \Longrightarrow a = d$

lemma *rename-vars*: $Skip = A \Longrightarrow A\ o\ B = C \Longrightarrow M = B \Longrightarrow M = C$

lemma *simp-to-fail*: $A = \{.p.\}\ o\ T \Longrightarrow (\bigwedge x\ .\ p\ x = False) \Longrightarrow A = \bot$

lemma *assert-true-comp*: $A = \{.p.\}\ o\ T \Longrightarrow (\bigwedge x\ .\ p\ x = True) \Longrightarrow A = T$

lemma *test-types*: $(a::real) = a \wedge b + 0 = b + 0 \wedge (c :: 'a \Rightarrow 'b) = c$

declare [[*show-types*]]

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

end

7.4 Python Simulation Code Generation

theory *PythonSimulation* **imports** *Real Transcendental SimulinkTypes*
begin

definition *PI-PY* = $(\lambda x::nat.\ s\text{-}pi)$

lemma *PI-PY-gen-simp*: $s\text{-}pi = PI\text{-}PY(0)$

lemma *PI-PY-simp*: $pi = PI\text{-}PY(0)$

definition *NOT-PY* = *Not*

lemma *NOT-PY-simp*: *Not* x = *NOT-PY*(x)

definition *AND-PY* = $(\lambda\ (x,\ y).\ x \wedge y)$

lemma *AND-PY-simp*: $(x \wedge y)$ = *AND-PY*(x,y)

definition *OR-PY* = $(\lambda\ (x,y).\ x \vee y)$

lemma *OR-PY-simp*: $(x \vee y)$ = *OR-PY*(x,y)

definition *LESS-PY* = $(\lambda\ (x,\ y)\ .\ x < y)$

lemma *LESS-PY-simp*: $(x < y)$ = *LESS-PY* $(x,\ y)$

definition *LE-PY* = $(\lambda\ (x,\ y)\ .\ x \leq y)$

lemma *LE-PY-simp*: $(x \leq y)$ = *LE-PY* $(x,\ y)$

definition *EQ-PY* = $(\lambda\ (x,\ y)\ .\ x = y)$

lemma *EQ-PY-simp*: $(x = y)$ = *EQ-PY* $(x,\ y)$

definition *ADD-PY* = $(\lambda\ (x,\ y).\ x + y)$

lemma *ADD-PY-simp*: $(x + y)$ = *ADD-PY* $(x,\ y)$

definition *SUBS-PY* = $(\lambda\ (x,\ y).\ x - y)$

lemma *SUBS-PY-simp*: $(x - y)$ = *SUBS-PY* $(x,\ y)$

definition *MULT-PY* = (λ (*x*, *y*). *x* ∗ *y*)

lemma *MULT-PY-simp*: (*x* ∗ *y*) = *MULT-PY* (*x*, *y*)

definition *DIV-PY* = (λ (*x*,*y*) . *x* / *y*)

lemma *DIV-PY-simp*: *x* / *y* = *DIV-PY* (*x*, *y*)

definition *ABS-PY* = (λ *x*. *s-abs x*)

lemma *ABS-PY-gen-simp*: *s-abs x* = *ABS-PY x*

lemma *ABS-PY-simp*: *abs* (*x*::*real*) = *ABS-PY x*

definition *POW-PY* = (λ(*x*,*y*). *power x y*)

lemma *POW-PY-simp*: (*x* ^ *y*) = *POW-PY* (*x*, *y*)

definition *SQRT-PY* = *s-sqrt*

lemma *SQRT-PY-gen-simp*: *s-sqrt x* = *SQRT-PY*(*x*)

lemma *SQRT-PY-simp*: *sqrt x* = *SQRT-PY*(*x*)

definition *EXP-PY* = *s-exp*

lemma *EXP-PY-gen-simp*: *s-exp x* = *EXP-PY*(*x*)

lemma *EXP-PY-simp*: *exp* (*x*::*real*) = *EXP-PY*(*x*)

definition *SIN-PY* = *s-sin*

lemma *SIN-PY-gen-simp*: *s-sin x = SIN-PY(x)*

lemma *SIN-PY-simp*: *sin (x::real) = SIN-PY(x)*

definition *FST-PY = fst*

lemma *FST-PY-simp*: *fst x = FST-PY (x)*

definition *SND-PY = snd*

lemma *SND-PY-simp*: *snd x = SND-PY (x)*

definition *IF-PY = (λ (b, x, y) . If b Then x Else y)*

lemma *IF-PY-gen-simp*: *(If b Then x Else y) = IF-PY (b, x, y)*

lemma *IF-PY-simp*: *(if b then x else y) = IF-PY (b, x, y)*

definition *IMP-PY = (λ (x, y) . x ⟶ y)*

lemma *IMP-PY-simp*: *(x ⟶ y) = IMP-PY (x, y)*

definition *CONVERSION-PY = (λ (x, y::nat) . conversion x)*

lemma *CONVERSION-PY-simp*: *conversion x = CONVERSION-PY (x,0)*

lemmas *python-simps = PI-PY-simp PI-PY-gen-simp NOT-PY-simp AND-PY-simp OR-PY-simp LESS-PY-simp LE-PY-simp EQ-PY-simp*
ADD-PY-simp SUBS-PY-simp MULT-PY-simp DIV-PY-simp ABS-PY-gen-simp ABS-PY-simp
POW-PY-simp SQRT-PY-gen-simp SQRT-PY-simp
EXP-PY-gen-simp EXP-PY-simp SIN-PY-gen-simp SIN-PY-simp
FST-PY-simp SND-PY-simp
IF-PY-simp IF-PY-gen-simp IMP-PY-simp
CONVERSION-PY-simp

end

8 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*]: $\bigwedge v\ .\ distinct\ u \Longrightarrow length\ u = length\ v \Longrightarrow Subst\ u\ v\ u = v$

lemma *Subst-inex*[*simp*]: $\bigwedge b.\ set\ a \cap set\ x = \{\} \Longrightarrow length\ a = length\ b \Longrightarrow Subst\ a\ b\ x = x$

lemma *set-Subst*: $set\ (Subst\ [a]\ [b]\ x) = (if\ a \in set\ x\ then\ (set\ x - \{a\}) \cup \{b\}\ else\ set\ x)$

lemma *distinct-Subst*: $distinct\ (b\#x) \Longrightarrow distinct\ (Subst\ [a]\ [b]\ x)$

lemma *inter-Subst*: $distinct(b\#y) \Longrightarrow set\ x \cap set\ y = \{\} \Longrightarrow b \notin set\ x \Longrightarrow set\ x \cap set\ (Subst\ [a]\ [b]\ y) = \{\}$

lemma *incl-Subst*: $distinct(b\#x) \Longrightarrow set\ y \subseteq set\ x \Longrightarrow set\ (Subst\ [a]\ [b]\ y) \subseteq set\ (Subst\ [a]\ [b]\ x)$

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

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

lemma *subst-not-in*: $\bigwedge y\ .\ a \notin set\ x' \Longrightarrow length\ x = length\ y \Longrightarrow length\ x' = length\ y' \Longrightarrow subst\ (x\ @\ x')\ (y\ @\ y')\ a = subst\ x\ y\ a$

lemma *subst-not-in-b*: $\bigwedge y\ .\ a \notin set\ x \Longrightarrow length\ x = length\ y \Longrightarrow length\ x' = length\ y' \Longrightarrow subst\ (x\ @\ x')\ (y\ @\ y')\ a = subst\ x'\ y'\ a$

lemma *Subst-not-in*: $set\ x' \cap set\ z = \{\} \Longrightarrow length\ x = length\ y \Longrightarrow length\ x' = length\ y' \Longrightarrow Subst\ (x\ @\ x')\ (y\ @\ y')\ z = Subst\ x\ y\ z$

lemma *Subst-not-in-a*: $set\ x \cap set\ z = \{\} \Longrightarrow length\ x = length\ y \Longrightarrow length\ x' = length\ y' \Longrightarrow Subst\ (x\ @\ x')\ (y\ @\ y')\ z = Subst\ x'\ y'\ z$

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

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

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

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

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

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

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

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

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

lemma *length-Subst*[*simp*]: *length* (*Subst x y z*) = *length z*

lemma *subst-Subst*: ⋀ *y y′* . *length y* = *length y′* ⟹ *a* ∈ *set w* ⟹ *subst w* (*Subst y y′ w*) *a* = *subst y y′ a*

lemma *Subst-Subst*: *length y* = *length y′* ⟹ *set z* ⊆ *set w* ⟹ *Subst w* (*Subst y y′ w*) *z* = *Subst y y′ z*

primrec *listinter* :: *′a list* ⇒ *′a list* ⇒ *′a list* (**infixl** ⊗ *60*) **where**
[] ⊗ *y* = [] |
(*a* # *x*) ⊗ *y* = (*if a* ∈ *set y then a* # (*x* ⊗ *y*) *else x* ⊗ *y*)

lemma *inter-filter*: *x* ⊗ *y* = *filter* (λ *a* . *a* ∈ *set y*) *x*

lemma *inter-append*: *set y* ∩ *set z* = {} ⟹ *perm* (*x* ⊗ (*y* @ *z*)) ((*x* ⊗ *y*) @ (*x* ⊗ *z*))

lemma *append-inter*: (*x* @ *y*) ⊗ *z* = (*x* ⊗ *z*) @ (*y* ⊗ *z*)

lemma *notin-inter* [*simp*]: *a* ∉ *set x* ⟹ *a* ∉ *set* (*x* ⊗ *y*)

lemma *distinct-inter*: *distinct x* ⟹ *distinct* (*x* ⊗ *y*)

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

primrec *diff* :: *′a list* ⇒ *′a list* ⇒ *′a list* (**infixl** ⊖ *52*) **where**
[] ⊖ *y* = [] |
(*a* # *x*) ⊖ *y* = (*if a* ∈ *set y then x* ⊖ *y else a* # (*x* ⊖ *y*))

lemma *diff-filter*: *x* ⊖ *y* = *filter* (λ *a* . *a* ∉ *set y*) *x*

lemma *diff-distinct*: *set x* ∩ *set y* = {} ⟹ (*y* ⊖ *x*) = *y*

lemma *set-diff*: *set* (*x* ⊖ *y*) = *set x* − *set y*

lemma *distinct-diff*: *distinct x* ⟹ *distinct* (*x* ⊖ *y*)

definition *addvars* :: *′a list* ⇒ *′a list* ⇒ *′a list* (**infixl** ⊕ *55*) **where**
addvars x y = *x* @ (*y* ⊖ *x*)

lemma *addvars-distinct*: *set x* ∩ *set y* = {} ⟹ *x* ⊕ *y* = *x* @ *y*

lemma *set-addvars*: *set* (*x* ⊕ *y*) = *set x* ∪ *set y*

lemma *distinct-addvars*: *distinct x* ⟹ *distinct y* ⟹ *distinct* (*x* ⊕ *y*)

lemma *mset-inter-diff*: *mset oa* = *mset* (*oa* ⊗ *ia*) + *mset* (*oa* ⊖ (*oa* ⊗ *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*: $set\ x \cap set\ z = \{\} \Longrightarrow x \otimes y \mathbin{@} z = x \otimes y$

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*: $x \ominus (y \mathbin{@} z) = ((x \ominus y) \otimes (x \ominus z))$

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*: $perm\ (x \ominus y)\ (((x \ominus y) \otimes z) \mathbin{@} ((x \ominus y) \ominus z))$

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

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

9 Translation of Hierarchical Block Diagrams

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

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

locale *BaseOperationFeedbackless* =

fixes *TI TO* :: $'a \Rightarrow 'tp\ list$

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

fixes *comp* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *oo 70*)
assumes *TI-comp*[*simp*]: $TI\ S' = TO\ S \Longrightarrow TI\ (S\ oo\ S') = TI\ S$
assumes *TO-comp*[*simp*]: $TI\ S' = TO\ S \Longrightarrow TO\ (S\ oo\ S') = TO\ S'$
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*: $TI\ T = TO\ S \Longrightarrow TI\ R = TO\ T \Longrightarrow S\ oo\ T\ oo\ R = S\ oo\ (T\ oo\ R)$

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

assumes *comp-parallel-distrib*: $TO\ S = TI\ S' \Longrightarrow TO\ T = TI\ T' \Longrightarrow (S \parallel T)\ oo\ (S' \parallel T') = (S\ oo\ S') \parallel (T\ oo\ T')$

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* ∥ *ID ts* = *ID ts*

assumes *Split-Switch*[*simp*]: *Split ts oo Switch ts ts* = *Split ts*
assumes *Split-assoc*: *Split ts oo ID ts* ∥ *Split ts* = *Split ts oo Split ts* ∥ *ID ts*

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

assumes *switch-par-no-vars*: *TI A* = *ti* ⟹ *TO A* = *to* ⟹ *TI B* = *ti'* ⟹ *TO B* = *to'* ⟹ *Switch ti ti' oo B* ∥ *A oo Switch to' to* = *A* ∥ *B*

fixes *fb* :: *'a ⇒ 'a*
assumes *TI-fb*: *TI S* = *t* # *ts* ⟹ *TO S* = *t* # *ts'* ⟹ *TI* (*fb S*) = *ts*
assumes *TO-fb*: *TI S* = *t* # *ts* ⟹ *TO S* = *t* # *ts'* ⟹ *TO* (*fb S*) = *ts'*
assumes *fb-comp*: *TI S* = *t* # *TO A* ⟹ *TO S* = *t* # *TI B* ⟹ *fb* (*ID* [*t*] ∥ *A oo S oo ID* [*t*] ∥ *B*) = *A oo fb S oo B*
assumes *fb-par-indep*: *TI S* = *t* # *ts* ⟹ *TO S* = *t* # *ts'* ⟹ *fb* (*S* ∥ *T*) = *fb S* ∥ *T*

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

begin
definition *fbtype S tsa ts ts'* = (*TI S* = *tsa* @ *ts* ∧ *TO S* = *tsa* @ *ts'*)

lemma *fb-comp-fbtype*: *fbtype S* [*t*] (*TO A*) (*TI B*)
⟹ *fb* ((*ID* [*t*] ∥ *A*) *oo S oo* (*ID* [*t*] ∥ *B*)) = *A oo fb S oo B*

lemma *fb-serial-no-vars*: *TO A* = *t* # *ts* ⟹ *TI B* = *t* # *ts*
⟹ *fb* (*ID* [*t*] ∥ *A oo Switch* [*t*] [*t*] ∥ *ID ts oo ID* [*t*] ∥ *B*) = *A oo B*

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

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

lemma *fb-par-indep-fbtype*: *fbtype S* [*t*] *ts ts'* ⟹ *fb* (*S* ∥ *T*) = *fb S* ∥ *T*

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

lemma *comp-id-right-simp* [*simp*]: *TO S* = *ts* ⟹ *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*: $TI\ S2 = TO\ S1 \Longrightarrow TI\ S3 = TO\ S2 \Longrightarrow TI\ S4 = TO\ S3 \Longrightarrow TI\ S5 = TO\ S4 \Longrightarrow$
$S1\ oo\ (S2\ oo\ S3\ oo\ S4)\ oo\ S5 = (S1\ oo\ S2)\ oo\ S3\ oo\ (S4\ oo\ S5)$

lemma *fb-gen-parallel*: $\bigwedge\ S\ .\ fbtype\ S\ tsa\ ts\ ts' \Longrightarrow (fb\hat{}\hat{}(length\ tsa))\ (S \parallel T) = ((fb\hat{}\hat{}(length\ tsa))\ (S)) \parallel T$

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

lemma *fb-indep*: $\bigwedge S.\ fbtype\ S\ tsa\ (TO\ A)\ (TI\ B) \Longrightarrow (fb\hat{}\hat{}(length\ tsa))\ ((ID\ tsa \parallel A)\ oo\ S\ oo\ (ID\ tsa \parallel B)) = A\ oo\ (fb\hat{}\hat{}(length\ tsa))\ S\ oo\ B$

lemma *fb-indep-a*: $\bigwedge S.$ *fbtype S tsa* (*TO A*) (*TI B*) $\Longrightarrow$ *length tsa* = *n* $\Longrightarrow$ (*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*) $\Longrightarrow$ *fb* (*S oo* (*ID* [*t*] ∥ *B*)) = *fb S oo B*

lemma *fb-comp-left*: *fbtype S* [*t*] (*TO A*) *ts* $\Longrightarrow$ *fb* ((*ID* [*t*] ∥ *A*) *oo S*) = *A oo fb S*

lemma *fb-indep-right*: $\bigwedge S.$ *fbtype S tsa ts* (*TI B*) $\Longrightarrow$ (*fb*^^(*length tsa*)) (*S oo* (*ID tsa* ∥ *B*)) = (*fb*^^(*length tsa*)) *S oo B*

lemma *fb-indep-left*: $\bigwedge S.$ *fbtype S tsa* (*TO A*) *ts* $\Longrightarrow$ (*fb*^^(*length tsa*)) ((*ID tsa* ∥ *A*) *oo S*) = *A oo* (*fb*^^(*length tsa*)) *S*

lemma *TI-fb-fbtype-n*: $\bigwedge S.$ *fbtype S t ts ts'* $\Longrightarrow$ *TI* ((*fb*^^(*length t*)) *S*) = *ts*
and *TO-fb-fbtype-n*: $\bigwedge S.$ *fbtype S t ts ts'* $\Longrightarrow$ *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* ⇒ *'var* ⇒ *'a* **where**
set-var [] *b* = *Arb* (*TV b*) |
set-var (*a* # *x*) *b* = (*if a* = *b then ID* [*TV a*] ∥ *Sink* (*TVs x*) *else Sink* [*TV a*] ∥ *set-var x b*)

lemma *TO-set-var*[*simp*]: *TO* (*set-var x a*) = [*TV a*]

lemma *TI-set-var*[*simp*]: *TI* (*set-var x a*) = *TVs x*

primrec *switch* :: *'var list* ⇒ *'var list* ⇒ *'a* ([- ⇝ -]) **where**
[*x* ⇝ []] = *Sink* (*TVs x*) |
[*x* ⇝ *a* # *y*] = *Split* (*TVs x*) *oo set-var x a* ∥ [*x* ⇝ *y*]

lemma *TI-switch*[*simp*]: *TI* [*x* ⇝ *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*: $\bigwedge A\ B\ v\ x\ .\ distinct\ (u\ @\ v\ @\ x) \Longrightarrow TO\ A = TVs\ (v@x) \Longrightarrow TI\ B = TVs\ (u\ @\ x) \Longrightarrow\ TVs\ u = TVs\ v$
$\Longrightarrow (fb\ \hat{}\hat{}\ length\ u)\ (([u \rightsquigarrow u] \parallel A)\ oo\ [u\ @\ v\ @\ x \rightsquigarrow v\ @\ u\ @\ x]\ oo\ ([v \rightsquigarrow v] \parallel B)) = A\ oo\ B$

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

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*: $distinct\ x \Longrightarrow distinct\ y \Longrightarrow distinct\ z \Longrightarrow set\ y \subseteq set\ x \Longrightarrow set\ z \subseteq set\ x$
$\Longrightarrow set\ y' \subseteq set\ y \Longrightarrow set\ z' \subseteq set\ z \Longrightarrow [x \rightsquigarrow y\ @\ z]\ oo\ [y \rightsquigarrow y'] \parallel [z \rightsquigarrow z'] = [x \rightsquigarrow y'\ @\ z']$

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

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

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

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

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

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

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

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*: $distinct\ a \Longrightarrow distinct\ b \Longrightarrow TVs\ a = TVs\ b \Longrightarrow [a \rightsquigarrow a\ @\ a] = [b \rightsquigarrow b\ @\ b]$

9.1.1 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*: $deterministic\ A \Longrightarrow distinct\ x \Longrightarrow distinct\ y \Longrightarrow TI\ A = TVs\ x \Longrightarrow TO\ A = TVs\ y$
$\Longrightarrow [x \rightsquigarrow x\ @\ x]\ oo\ (A \parallel A) = A\ oo\ [y \rightsquigarrow y\ @\ y]$

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

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

9.2 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* $\Longrightarrow$ *TO S* = *t'* # *t* # *ts'*
$\Longrightarrow$ (*fb* ^^ (*2*::*nat*)) (*Switch* [*t*] [*t'*] $\parallel$ *ID ts oo S oo Switch* [*t'*] [*t*] $\parallel$ *ID ts'*) = (*fb* ^^ (*2*:: *nat*)) *S*

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

lemma *fb-switch-a*: $\bigwedge$ *S* . *distinct* (*a* # *z* @ *x*) $\Longrightarrow$ *distinct* (*a* # *z* @ *y*) $\Longrightarrow$ *TI S* = *TVs* (*z* @ *a* # *x*) $\Longrightarrow$ *TO S* = *TVs* (*z* @ *a* # *y*)
$\Longrightarrow$ (*fb* ^^ (*Suc* (*length z*))) ([*a* # *z* @ *x* $\rightsquigarrow$ *z* @ *a* # *x*] *oo S oo* [*z* @ *a* # *y* $\rightsquigarrow$ *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*: $\bigwedge$ *v x y S* . *distinct* (*u* @ *v* @ *x*) $\Longrightarrow$ *distinct* (*u* @ *v* @ *y*) $\Longrightarrow$ *TI S* = *TVs* (*v* @ *u* @ *x*) $\Longrightarrow$ *TO S* = *TVs* (*v* @ *u* @ *y*)
$\Longrightarrow$ (*fb* ^^ (*length* (*u* @ *v*))) ([*u* @ *v* @ *x* $\rightsquigarrow$ *v* @ *u* @ *x*] *oo S oo* [*v* @ *u* @ *y* $\rightsquigarrow$ *u* @ *v* @ *y*]) = (*fb* ^^ (*length* (*u* @ *v*))) *S*

theorem *fb-perm*: $\bigwedge$ *v S* . *perm u v* $\Longrightarrow$ *distinct* (*u* @ *x*) $\Longrightarrow$ *distinct* (*u* @ *y*) $\Longrightarrow$ *fbtype S* (*TVs u*) (*TVs x*) (*TVs y*)
$\Longrightarrow$ (*fb* ^^ (*length u*)) ([*v* @ *x* $\rightsquigarrow$ *u* @ *x*] *oo S oo* [*u* @ *y* $\rightsquigarrow$ *v* @ *y*]) = (*fb* ^^ (*length u*)) *S*

end

end

9.3 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 $Var\ A\ B = (Out\ A) \otimes (In\ B)$

definition $io\text{-}diagram\ A = (TVs\ (In\ A) = TI\ (Trs\ A) \wedge TVs\ (Out\ A) = TO\ (Trs\ A) \wedge distinct\ (In\ A) \wedge distinct\ (Out\ A))$

definition $Comp :: ('var, 'a)\ Dgr \Rightarrow ('var, 'a)\ Dgr \Rightarrow ('var, 'a)\ Dgr$ (**infixl** ;; *70*) **where**
$A ;; B = (let\ I = In\ B \ominus Var\ A\ B\ in\ let\ O' = Out\ A \ominus Var\ A\ B\ in$
$(\!|In = (In\ A) \oplus I,\ Out = O' @ Out\ B,$
$Trs = [(In\ A) \oplus I \rightsquigarrow In\ A @ I\ ]\ oo\ Trs\ A \parallel [I \rightsquigarrow I]\ oo\ [Out\ A @ I \rightsquigarrow O' @ In\ B]\ oo\ ([O' \rightsquigarrow O'] \parallel Trs\ B)\ |\!))$

lemma *io-diagram-Comp*: $io\text{-}diagram\ A \Longrightarrow io\text{-}diagram\ B \Longrightarrow set\ (Out\ A \ominus In\ B) \cap set\ (Out\ B) = \{\} \Longrightarrow io\text{-}diagram\ (A ;; B)$

lemma *Comp-in-disjoint*:
assumes *io-diagram A*
and *io-diagram B*
and $set\ (In\ A) \cap set\ (In\ B) = \{\}$
shows $A ;; B = (let\ I = In\ B \ominus Var\ A\ B\ in\ let\ O' = Out\ A \ominus Var\ A\ B\ in$
$(\!|In = (In\ A) @ I,\ Out = O' @ Out\ B,\ Trs = Trs\ A \parallel [I \rightsquigarrow I]\ oo\ [Out\ A @ I \rightsquigarrow O' @ In\ B]\ oo\ ([O' \rightsquigarrow O'] \parallel Trs\ B)\ |\!))$

lemma *Comp-full*: $io\text{-}diagram\ A \Longrightarrow io\text{-}diagram\ B \Longrightarrow Out\ A = In\ B \Longrightarrow$
$A ;; B = (\!|In = In\ A,\ Out = Out\ B,\ Trs = Trs\ A\ oo\ Trs\ B\ |\!)$

lemma *Comp-in-out*: $io\text{-}diagram\ A \Longrightarrow io\text{-}diagram\ B \Longrightarrow set\ (Out\ A) \subseteq set\ (In\ B) \Longrightarrow$
$A ;; B = (let\ I = diff\ (In\ B)\ (Var\ A\ B)\ in\ let\ O' = diff\ (Out\ A)\ (Var\ A\ B)\ in$
$(\!|In = In\ A \oplus I,\ Out = Out\ B,\ Trs = [In\ A \oplus I \rightsquigarrow In\ A @ I\ ]\ oo\ Trs\ A \parallel [I \rightsquigarrow I]\ oo\ [Out\ A @ I \rightsquigarrow In\ B]\ oo\ Trs\ B\ |\!))$

lemma *Comp-assoc-new*: $io\text{-}diagram\ A \Longrightarrow io\text{-}diagram\ B \Longrightarrow io\text{-}diagram\ C \Longrightarrow$
$set\ (Out\ A \ominus In\ B) \cap set\ (Out\ B) = \{\} \Longrightarrow set\ (Out\ A \otimes In\ B) \cap set\ (In\ C) = \{\}$
$\Longrightarrow A ;; B ;; C = A ;; (B ;; C)$

lemma *Comp-assoc-a*: $io\text{-}diagram\ A \Longrightarrow io\text{-}diagram\ B \Longrightarrow io\text{-}diagram\ C \Longrightarrow$
$set\ (In\ B) \cap set\ (In\ C) = \{\} \Longrightarrow$
$set\ (Out\ A) \cap set\ (Out\ B) = \{\} \Longrightarrow$
$A ;; B ;; C = A ;; (B ;; C)$

definition $Parallel :: ('var, 'a)\ Dgr \Rightarrow ('var, 'a)\ Dgr \Rightarrow ('var, 'a)\ Dgr$ (**infixl** ||| *80*) **where**
$A ||| B = (\!|In = In\ A \oplus In\ B,\ Out = Out\ A @ Out\ B,\ Trs = [In\ A \oplus In\ B \rightsquigarrow In\ A @ In\ B]\ oo\ (Trs\ A \parallel Trs\ B)\ |\!)$

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

lemma *Parallel-indep*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*In A*) $\cap$ *set* (*In B*) = {} $\Longrightarrow$ *A* ||| *B* = (|*In* = *In A* @ *In B*, *Out* = *Out A* @ *Out B*, *Trs* = (*Trs A* ‖ *Trs B*) |)

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

definition *VarFB A* = *Var A A*
definition *InFB A*= *In A* $\ominus$ *VarFB A*
definition *OutFB A* = *Out A* $\ominus$ *VarFB A*

definition *FB* :: ('*var*, '*a*) *Dgr* $\Rightarrow$ ('*var*, '*a*) *Dgr* **where**
FB A = (*let I* = *In A* $\ominus$ *Var A A in let O'* = *Out A* $\ominus$ *Var A A in*
(|*In* = *I*, *Out* = *O'*, *Trs* = (*fb* ^^ (*length* (*Var A A*))) ([*Var A A* @ *I* $\rightsquigarrow$ *In A*] *oo Trs A oo* [*Out A* $\rightsquigarrow$ *Var A A* @ *O'*]) |))

lemma *Type-ok-FB*: *io-diagram A* $\Longrightarrow$ *io-diagram* (*FB A*)

lemma *perm-var-Par*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*In A*) $\cap$ *set* (*In B*) = {} $\Longrightarrow$ *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* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *set* (*Out A*) $\cap$ *set* (*Out B*) = {} $\Longrightarrow$ *distinct* (*Var* (*A* ||| *B*) (*A* ||| *B*))

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

lemma *drop-assumption*: *p* $\Longrightarrow$ *True*

lemma *Dgr-eq*: *In A* = *x* $\Longrightarrow$ *Out A* = *y* $\Longrightarrow$ *Trs A* = *S* $\Longrightarrow$ (|*In* = *x*, *Out* = *y*, *Trs* = *S*|) = *A*

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

theorem *FB-idemp*: *io-diagram A* $\Longrightarrow$ *FB* (*FB A*) = *FB A*

definition *VarSwitch* :: '*var list* $\Rightarrow$ '*var list* $\Rightarrow$ ('*var*, '*a*) *Dgr* ([[- $\rightsquigarrow$ -]]) **where**
VarSwitch x y = (|*In* = *x*, *Out* = *y*, *Trs* = [*x* $\rightsquigarrow$ *y*]|)

definition *in-equiv A B* = (*perm* (*In A*) (*In B*) $\wedge$ *Trs A* = [*In A* $\rightsquigarrow$ *In B*] *oo Trs B* $\wedge$ *Out A* = *Out B*)
definition *out-equiv A B* = (*perm* (*Out A*) (*Out B*) $\wedge$ *Trs A* = *Trs B oo* [*Out B* $\rightsquigarrow$ *Out A*] $\wedge$ *In A* = *In B*)

definition *in-out-equiv A B* = (*perm* (*In A*) (*In B*) $\wedge$ *perm* (*Out A*) (*Out B*) $\wedge$ *Trs A* = [*In A* $\rightsquigarrow$ *In B*] *oo Trs B oo* [*Out B* $\rightsquigarrow$ *Out A*])

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

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

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

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

lemma [*simp*]: *io-diagram A* $\Longrightarrow$ [*In A* $\rightsquigarrow$ *In A*] *oo Trs A oo* [*Out A* $\rightsquigarrow$ *Out A*] = *Trs A*

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

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

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

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

lemma [*simp*]: *distinct* (*Out A*) $\Longrightarrow$ *distinct* (*Var A B*)

lemma [*simp*]: *set* (*Var A B*) $\subseteq$ *set* (*Out A*)
lemma [*simp*]: *set* (*Var A B*) $\subseteq$ *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* $\in$ *X then x* # *inter-set xs X else inter-set xs X*)

lemma *list-inter-set*: *x* $\otimes$ *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* $\wedge$ *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* = ⋃ (*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*) ⟹ *Out A* = [*out A*]

lemma *concat-map-Out-get-other-out*: *Type-OK As* ⟹ *concat* (*map Out* (*get-other-out a As*)) = (*concat* (*map Out As*) ⊖ [*a*])

thm *Out-out*

lemma *VarFB-cons-out*: *Type-OK As* ⟹ *VarFB* (*Parallel-list As*) = *a* # *L* ⟹ ∃ *A* ∈ *set As* . *out A* = *a*

lemma *VarFB-cons-out-In*: *Type-OK As* ⟹ *VarFB* (*Parallel-list As*) = *a* # *L* ⟹ ∃ *B* ∈ *set As* . *a* ∈ *set* (*In B*)

lemma *AAA-a*: *Type-OK* (*A* # *As*) ⟹ *A* ∉ *set As*

lemma *AAA-b*: (∀ *A* ∈ *set As*. *a* ∉ *set* (*Out A*)) ⟹ *get-other-out a As* = *As*

lemma *AAA-d*: *Type-OK* (*A* # *As*) ⟹ ∀ *Aa*∈*set As*. *out A* ≠ *out Aa*

lemma *mem-get-other-out*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *get-other-out* (*out A*) *As* = (*As* ⊖ [*A*])

lemma *In-CompA*: *In* (*A* ▷ *B*) = (*if out A* ∈ *set* (*In B*) *then In A* ⊕ (*In B* ⊖ *Out A*) *else In B*)

lemma *union-set-In-CompA*: ⋀ *B* . *length* (*Out A*) = *1* ⟹ *B* ∈ *set As* ⟹ *out A* ∈ *set* (*In B*) ⟹ (⋃*x*∈*set As*. *set* (*In* (*CompA A x*))) = *set* (*In A*) ∪ ((⋃ *B* ∈ *set As* . *set* (*In B*)) − {*out A*})

lemma *BBBB-e*: *Type-OK As* ⟹ *VarFB* (*Parallel-list As*) = *out A* # *L* ⟹ *A* ∈ *set As* ⟹ *out A* ∉ *set L*

lemma *BBBB-f*: *loop-free As* ⟹
Type-OK As ⟹ *A* ∈ *set As* ⟹ *B* ∈ *set As* ⟹ *out A* ∈ *set* (*In B*) ⟹ *B* ≠ *A*

thm *union-set-In-CompA*

lemma [*simp*]: *x* ∈ *set* (*Out* (*get-comp-out x As*))

lemma *comp-out-in*: *A* ∈ *set As* ⟹ *a* ∈ *set* (*Out A*) ⟹ (*get-comp-out a As*) ∈ *set As*

lemma [*simp*]: *a* ∈ *internal As* ⟹ *get-comp-out a As* ∈ *set As*

lemma *out-CompA*: *length* (*Out A*) = *1* ⟹ *out* (*CompA A B*) = *out B*

lemma *Type-OK-loop-free-elem*: *Type-OK As* ⟹ *loop-free As* ⟹ *A* ∈ *set As* ⟹ *out A* ∉ *set* (*In A*)

lemma *BBB-a*: *length* (*Out A*) = *1* ⟹ *Out* (*CompA A B*) = *Out B*

lemma *BBB-b*: *length* (*Out A*) = *1* ⟹ *map* (*Out* ∘ *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* ∈ *internal As*
shows *VarFB* (*Parallel-list* (*fb-out-less-step a As*)) = (*VarFB* (*Parallel-list As*)) ⊖ [*a*]

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

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

lemma *Parallel-list-cons*:*Parallel-list* (*a* # *As*) = *a* ||| *Parallel-list As*

lemma *io-diagram-parallel-list*: *Type-OK As* ⟹ *io-diagram* (*Parallel-list As*)

lemma *BBB-c*: *distinct* (*map f As*) ⟹ *distinct* (*map f* (*As* ⊖ *Bs*))

lemma *io-diagram-CompA*: *io-diagram A* ⟹ *length* (*Out A*) = *1* ⟹ *io-diagram B* ⟹ *io-diagram* (*CompA A B*)

lemma *Type-OK-fb-out-less-step-aux*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *Type-OK* (*fb-less-step A* (*As* ⊖ [*A*]))

thm *VarFB-cons-out*

theorem *Type-OK-fb-out-less-step-new*: *Type-OK As* ⟹
a ∈ *internal As* ⟹
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*: $(\forall\ A \in set\ As\ .\ io\text{-}diagram\ A) \Longrightarrow TO\ (Trs\ (Parallel\text{-}list\ As)) = TVs\ (concat\ (map\ Out\ As))$

lemma *fbtype-aux*: $(Type\text{-}OK\ As) \Longrightarrow loop\text{-}free\ As \Longrightarrow VarFB\ (Parallel\text{-}list\ As) = a\ \#\ L \Longrightarrow$
$fbtype\ ([L\ @\ (In\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) \ominus L) \rightsquigarrow In\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As))]\ oo\ Trs\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As))\ oo$
$[Out\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) \rightsquigarrow L\ @\ (Out\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) \ominus L)])$
$(TVs\ L)\ (TO\ [In\ (Parallel\text{-}list\ As) \ominus a\ \#\ L \rightsquigarrow In\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) \ominus L])\ (TVs\ (Out\ (Parallel\text{-}list\ (fb\text{-}out\text{-}less\text{-}step\ a\ As)) \ominus L))$

lemma *fb-indep-left-a*: $fbtype\ S\ tsa\ (TO\ A)\ ts \Longrightarrow A\ oo\ (fb\hat{}\hat{}(length\ tsa))\ S = (fb\hat{}\hat{}(length\ tsa))\ ((ID\ tsa \parallel A)\ oo\ S)$

lemma *parallel-list-cons*: $parallel\text{-}list\ (A\ \#\ As) = A \parallel parallel\text{-}list\ As$

lemma *TI-parallel-list*: $(\forall\ A \in set\ As\ .\ io\text{-}diagram\ A) \Longrightarrow TI\ (parallel\text{-}list\ (map\ Trs\ As)) = TVs\ (concat\ (map\ In\ As))$

lemma *TO-parallel-list*: $(\forall\ A \in set\ As\ .\ io\text{-}diagram\ A) \Longrightarrow TO\ (parallel\text{-}list\ (map\ Trs\ As)) = TVs\ (concat\ (map\ Out\ As))$

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

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

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

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

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* $\Longrightarrow$ $[x \oplus y \rightsquigarrow x \mathbin{@} y]$ *oo* $[x \rightsquigarrow u] \parallel [y \rightsquigarrow v] = [x \oplus y \rightsquigarrow u \mathbin{@} v]$

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* $(Out\ A) = 1 \Longrightarrow$ *map* $(out \circ CompA\ A)\ As = map\ out\ As$

lemma *CompA-in*[*simp*]: *out A* $\in$ *set* $(In\ B) \Longrightarrow A \rhd B = A \mathbin{;;} B$

lemma *CompA-not-in*[*simp*]: *out A* $\notin$ *set* $(In\ B) \Longrightarrow A \rhd B = B$

lemma *in-equiv-CompA-Parallel-a*: *deterministic* $(Trs\ A) \Longrightarrow$ *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* $\in$ *set* $(In\ C)$
$\Longrightarrow$ *in-equiv* $((A \rhd B) \mid\mid\mid (A \rhd C))\ (A \rhd (B \mid\mid\mid C))$

lemma *in-equiv-CompA-Parallel-c*: *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* $\in$ *set* $(In\ C) \Longrightarrow$
in-equiv $(CompA\ A\ B \mid\mid\mid CompA\ A\ C)\ (CompA\ A\ (B \mid\mid\mid C))$

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'* $\cap$ *set z* $= \{\} \Longrightarrow$ *TVs x* $=$ *TVs y* $\Longrightarrow$ *TVs x'* $=$ *TVs y'* $\Longrightarrow$ *Subst* $(x \mathbin{@} x')$ $(y \mathbin{@} y')\ z =$ *Subst x y z*

lemma [*simp*]: $set\ x \cap set\ z = \{\} \Longrightarrow TVs\ x = TVs\ y \Longrightarrow TVs\ x' = TVs\ y' \Longrightarrow Subst\ (x\ @\ x')\ (y\ @\ y')\ z = Subst\ x'\ y'\ z$

lemma [*simp*]: $set\ x \cap set\ z = \{\} \Longrightarrow TVs\ x = TVs\ y \Longrightarrow Subst\ x\ y\ z = z$

lemma [*simp*]: $distinct\ x \Longrightarrow TVs\ x = TVs\ y \Longrightarrow Subst\ x\ y\ x = y$

lemma $TVs\ x = TVs\ y \Longrightarrow length\ x = length\ y$

thm *length-TVs*

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

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

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*: $io\text{-}diagram\ A' \Longrightarrow io\text{-}diagram\ B' \Longrightarrow in\text{-}equiv\ A\ A' \Longrightarrow in\text{-}equiv\ B\ B' \Longrightarrow in\text{-}equiv\ (A\ ;;\ B)\ (A'\ ;;\ B')$

lemma $io\text{-}diagram\ A' \Longrightarrow io\text{-}diagram\ B' \Longrightarrow in\text{-}equiv\ A\ A' \Longrightarrow in\text{-}equiv\ B\ B' \Longrightarrow in\text{-}equiv\ (CompA\ A\ \ B)\ (CompA\ A'\ B')$

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 $z = ts \Longrightarrow [x \rightsquigarrow z \mathbin{@} y] \mathbin{oo} (ID\ ts) \parallel [y' \rightsquigarrow u] = [x \rightsquigarrow z \mathbin{@} Subst\ y'\ y\ u]$

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* $z =$ *TVs* ts
$\Longrightarrow [x \rightsquigarrow y \mathbin{@} z] \mathbin{oo} [y \rightsquigarrow u] \parallel (ID\ ts) = [x \rightsquigarrow u \mathbin{@} z]$

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* $y =$ *TVs* $ts \Longrightarrow [x \rightsquigarrow y \mathbin{@} z] \mathbin{oo} (ID\ ts) \parallel [z \rightsquigarrow u] = [x \rightsquigarrow y \mathbin{@} u]$

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-list* (*fb-less-step* $A\ As$)) (*CompA* A (*Parallel-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* $\parallel$ *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-OK As* $\Longrightarrow$ *distinct As*

lemma *TI-parallel-list-a*: *TI* (*parallel-list As*) = *concat* (*map TI As*)

lemma *fb-CompA-aux*: *Type-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** $- \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow - \Longrightarrow fb\ ?Ta = ?Tb$)

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

lemma *fb-CompA*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *out A* = *a* ⟹ *a* ∉ *set* (*In A*) ⟹ *C* = *A* ▷ (*Parallel-list* (*As* ⊖ [*A*])) ⟹
OutAs = *Out* (*Parallel-list As*) ⟹ *perm y* (*In C*) ⟹ *perm z* (*Out C*) ⟹ *B* ∈ *set As* − {*A*} ⟹ *a* ∈ *set* (*In B*) ⟹
fb ([*a* # *y* ⇝ *concat* (*map In As*)] *oo parallel-list* (*map Trs As*) *oo* [*OutAs* ⇝ *a* # *z*]) = [*y* ⇝ *In C*] *oo Trs C oo* [*Out C* ⇝ *z*]

definition *Deterministic As* = (∀ *A* ∈ *set As* . *deterministic* (*Trs A*))

lemma *Deterministic-fb-out-less-step*: *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *a* = *out A* ⟹ *Deterministic As* ⟹ *Deterministic* (*fb-out-less-step a As*)

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

lemma *io-diagram-FB-Parallel-list*: *Type-OK As* ⟹ *io-diagram* (*FB* (*Parallel-list As*))

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

thm *loop-free-def*

lemma *io-rel-compA*: *length* (*Out A*) = *1* ⟹ *io-rel* (*CompA A B*) ⊆ *io-rel B* ∪ (*io-rel B O io-rel A*)

theorem *loop-free-fb-out-less-step*: *loop-free As* ⟹ *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *out A* = *a* ⟹ *loop-free* (*fb-out-less-step a As*)

theorem *in-equiv-FB-fb-less-delete*: ⋀ *As* . *Deterministic As* ⟹ *loop-free As* ⟹ *Type-OK As* ⟹ *VarFB* (*Parallel-list As*) = *L* ⟹
in-equiv (*FB* (*Parallel-list As*)) (*Parallel-list* (*fb-less L As*)) ∧ *io-diagram* (*Parallel-list* (*fb-less L As*))

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

lemma [*simp*]: ⋀ *x* . *distinct x* ⟹ *distinct y* ⟹ *perm* (((*y* ⊗ *x*) @ (*x* ⊖ *y* ⊗ *x*))) *x*

lemma [*simp*]: *io-diagram X* ⟹ *perm* (*VarFB X* @ (*In X* ⊖ *VarFB X*)) (*In X*)

lemma *Type-OK-diff* [*simp*]: *Type-OK As* ⟹ *Type-OK* (*As* ⊖ *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* . ($\exists$ *A* $\in$ *set As* . *out A* = *a* $\wedge$ ($\exists$ *B* $\in$ *set As*. *a* $\in$ *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* $\subseteq$ *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*: $a \in$ *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* $\rightsquigarrow$ *In X*] *oo Trs X oo* [*Out X* $\rightsquigarrow$ *a* # *z*]) = [*y* $\rightsquigarrow$ *In Y*] *oo Trs Y oo* [*Out Y* $\rightsquigarrow$ *z*]
and *perm* (*a* # *In Y*) (*In X*)

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

lemma *internal-Out-Parallel-list*: $a \in$ *internal As* $\Longrightarrow$ $a \in$ *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* $\subseteq$ *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* $\ominus$ *L*) $\rightsquigarrow$ *In X*] *oo Trs X oo* [*Out X* $\rightsquigarrow$ *L* @ (*Out X* $\ominus$ *L*)]) = [*In X* $\ominus$ *L* $\rightsquigarrow$ *In Y*] *oo Trs Y*
$\wedge$ *perm* (*In X* $\ominus$ *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* ^^ *length* (*L*)) ([*L* @ *InFB X* ⇝ *In X*] *oo Trs X oo* [*Out X* ⇝ *L* @ *OutFB X*]) = [*InFB X* ⇝ *In Y*] *oo Trs Y*
and *B*: *perm* (*InFB X*) (*In Y*)

definition *fb-perm-eq A* = (∀ *x*. *perm x* (*VarFB A*) ⟶
(*fb* ^^ *length* (*VarFB A*)) ([*VarFB A* @ *InFB A* ⇝ *In A*] *oo Trs A oo* [*Out A* ⇝ *VarFB A* @ *OutFB A*]) =
(*fb* ^^ *length* (*VarFB A*)) ([*x* @ *InFB A* ⇝ *In A*] *oo Trs A oo* [*Out A* ⇝ *x* @ *OutFB A*]))

lemma *fb-perm-eq-simp*: *fb-perm-eq A* = (∀ *x*. *perm x* (*VarFB A*) ⟶
Trs (*FB A*) = (*fb* ^^ *length* (*VarFB A*)) ([*x* @ *InFB A* ⇝ *In A*] *oo Trs A oo* [*Out A* ⇝ *x* @ *OutFB A*]))

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

lemma [*simp*]: *distinct* (*concat* (*map f As*)) ⟹ *distinct* (*concat* (*map f* (*As* ∈ [*A*])))

lemma *set-op-list-addvars*: *set* (*op-list* [] *op* ⊕ *x*) = (⋃ *a* ∈ *set x* . *set a*)

end

context *BaseOperationFeedbacklessVars*

begin

lemma [*simp*]: *set* (*Out A*) ⊆ *set* (*In B*) ⟹ *Out* ((*A* ;; *B*)) = *Out B*

lemma [*simp*]: *set* (*Out A*) ⊆ *set* (*In B*) ⟹ *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* ⊆ *set x*
and [*simp*]: *set z* ⊆ *set x*
and [*simp*]: *set u* ⊆ *set x*
and [*simp*]: *set y′* ⊆ *set y*
and [*simp*]: *set z′* ⊆ *set z*
and [*simp*]: *set u′* ⊆ *set u*

shows $[x \rightsquigarrow y \mathbin{@} z \mathbin{@} u] \mathbin{oo} [y \rightsquigarrow y'] \parallel [z \rightsquigarrow z'] \parallel [u \rightsquigarrow u'] = [x \rightsquigarrow y' \mathbin{@} z' \mathbin{@} u']$

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* $\subseteq$ *set x* **and** [*simp*]: *set z* $\subseteq$ *set x* **and** [*simp*]: *set t* $\subseteq$ *set x*
and [*simp*]: *set u* $\subseteq$ *set y'* **and** [*simp*]: *set v* $\subseteq$ *set z'* **and** [*simp*]: *set w* $\subseteq$ *set t'*
and [*simp*]: *TVs y* = *TVs y'* **and** [*simp*]: *TVs z* = *TVs z'* **and** [*simp*]: *TVs t* = *TVs t'*

shows $[x \rightsquigarrow y \mathbin{@} z \mathbin{@} t] \mathbin{oo} [y' \rightsquigarrow u] \parallel [z' \rightsquigarrow v] \parallel [t' \rightsquigarrow w] = [x \rightsquigarrow$ *Subst y' y u* @ *Subst z' z v* @ *Subst t' t w*$]$

lemma *Comp-assoc-single*: *length* (*Out A*) = *1* $\Longrightarrow$ *length* (*Out B*) = *1* $\Longrightarrow$ *out A* $\neq$ *out B* $\Longrightarrow$ *io-diagram A*
$\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *io-diagram C* $\Longrightarrow$ *out B* $\notin$ *set* (*In A*) $\Longrightarrow$
deterministic (*Trs A*) $\Longrightarrow$
out A $\in$ *set* (*In B*) $\Longrightarrow$ *out A* $\in$ *set* (*In C*) $\Longrightarrow$ *out B* $\in$ *set* (*In C*) $\Longrightarrow$ (*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* $\notin$ *set* (*In A*)
and [*simp*]: *out A* $\notin$ *set* (*In B*)
and [*simp*]: *out A* $\in$ *set* (*In C*)
and [*simp*]: *out B* $\in$ *set* (*In C*)
and *Diff*: *out A* $\neq$ *out B*

shows *Trs* (*A* ;; (*B* ;; *C*)) =
[*In A* $\oplus$ *In B* $\oplus$ (*In C* $\ominus$ [*out A*] $\ominus$ [*out B*]) $\rightsquigarrow$ *In A* @ *In B* @ (*In C* $\ominus$ [*out A*] $\ominus$ [*out B*])]
oo Trs A $\parallel$ *Trs B* $\parallel$ [*In C* $\ominus$ [*out A*] $\ominus$ [*out B*] $\rightsquigarrow$ *In C* $\ominus$ [*out A*] $\ominus$ [*out B*]]
oo [*out A* # *out B* # (*In C* $\ominus$ [*out A*] $\ominus$ [*out B*]) $\rightsquigarrow$ *In C*]
oo Trs C
and *In* (*A* ;; (*B* ;; *C*)) = *In A* $\oplus$ *In B* $\oplus$ (*In C* $\ominus$ [*out A*] $\ominus$ [*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* $\notin$ *set* (*In A*)
and [*simp*]: *out A* $\notin$ *set* (*In B*)
and [*simp*]: *out A* $\in$ *set* (*In C*)
and [*simp*]: *out B* $\in$ *set* (*In C*)
and *Diff*: *out A* $\neq$ *out B*
shows *in-equiv* (*A* ;; (*B* ;; *C*)) (*B* ;; (*A* ;; *C*))

lemma *CompA-commute-aux-a*: *io-diagram A* $\Longrightarrow$ *io-diagram B* $\Longrightarrow$ *io-diagram C* $\Longrightarrow$ *length* (*Out A*) = *1* $\Longrightarrow$ *length* (*Out B*) = *1*

$\Longrightarrow$ *out A* $\notin$ *set* (*Out C*) $\Longrightarrow$ *out B* $\notin$ *set* (*Out C*)
$\Longrightarrow$ *out A* $\neq$ *out B* $\Longrightarrow$ *out A* $\in$ *set* (*In B*) $\Longrightarrow$ *out B* $\notin$ *set* (*In A*)
$\Longrightarrow$ *deterministic* (*Trs A*)
$\Longrightarrow$ (*CompA* (*CompA B A*) (*CompA B C*)) = (*CompA* (*CompA A B*) (*CompA A C*))

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

fun *In-Equiv* :: (('*var*, '*a*) *Dgr*) *list* $\Rightarrow$ (('*var*, '*a*) *Dgr*) *list* $\Rightarrow$ *bool* **where**
In-Equiv [] [] = *True* |
In-Equiv (*A* # *As*) (*B*# *Bs*) = (*in-equiv A B* $\wedge$ *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* $\notin$ *set* (*Out C*)
and [*simp*]: *out B* $\notin$ *set* (*Out C*)
and [*simp*]: *out A* $\neq$ *out B*
and [*simp*]: *deterministic* (*Trs A*)
and [*simp*]: *deterministic* (*Trs B*)
and *A*: (*out A* $\in$ *set* (*In B*) $\Longrightarrow$ *out B* $\notin$ *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*: ($\bigwedge$ *C* . *C* $\in$ *set As* $\Longrightarrow$ *io-diagram C* $\wedge$ *out A* $\notin$ *set* (*Out C*) $\wedge$ *out B* $\notin$ *set* (*Out C*)) $\Longrightarrow$ *io-diagram A* $\Longrightarrow$ *io-diagram B*
$\Longrightarrow$ *length* (*Out A*) = *1* $\Longrightarrow$ *length* (*Out B*) = *1* $\Longrightarrow$ *out A* $\neq$ *out B*
$\Longrightarrow$ *deterministic* (*Trs A*) $\Longrightarrow$ *deterministic* (*Trs B*)
$\Longrightarrow$ (*out A* $\in$ *set* (*In B*) $\Longrightarrow$ *out B* $\notin$ *set* (*In A*))
$\Longrightarrow$ *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*: $(\bigwedge b\ .\ b \in set\ x \Longrightarrow b \neq a \Longrightarrow f\ b \neq f\ a) \Longrightarrow map\ f\ x \ominus [f\ a] = map\ f\ (x \ominus [a])$

lemma *In-Equiv-fb-out-less-step-commute*: *Type-OK As* $\Longrightarrow$ *Deterministic As* $\Longrightarrow$ $x \in internal\ As \Longrightarrow$ $y \in internal\ As \Longrightarrow x \neq y \Longrightarrow$ *loop-free As*
$\Longrightarrow$ *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* $\Longrightarrow$ *In-Equiv As As*

lemma *fb-less-append*: $\bigwedge$ *As* . *fb-less* (*x* @ *y*) *As* = *fb-less y* (*fb-less x As*)

thm *in-equiv-tran*

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

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

9.4 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

9.5 HBD Translation Algorithms that use Feedback Composition

theory *HBDTranslationsUsingFeedback* **imports** *HBDTranslationProperties ../RefinementReactive/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* $\Longrightarrow$
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* $\Longrightarrow$ *distinct* (*concat* (*map Out As*)) $\Longrightarrow$ *io-diagram* (*Parallel-list As*)

lemma *io-diagram-Parallel-list-a*: *io-distinct As* $\Longrightarrow$ *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*: $\forall\ A \in set\ As\ .\ io\text{-}diagram\ A \Longrightarrow distinct\ (concat\ (map\ Out\ As)) \Longrightarrow \forall\ A \in set\ Bs\ .\ io\text{-}diagram\ A$
$\Longrightarrow distinct\ (concat\ (map\ Out\ Bs)) \Longrightarrow$
$Parallel\text{-}list\ (As\ @\ Bs) = Parallel\text{-}list\ As\ |||\ Parallel\text{-}list\ Bs$

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* ⟹ *io-diagram C* ⟹ *set* (*Out A*) ∩ *set* (*Out B*) = {} ⟹ *in-out-equiv B C*
⟹ *in-out-equiv* (*A* ||| *B*) (*A* ||| *C*)

lemma *perm-map*: *perm x y* ⟹ *perm* (*map f x*) (*map f y*)

lemma *distinct-concat-perm*: ⋀ *Y* . *distinct* (*concat X*) ⟹ *perm X Y* ⟹ *distinct* (*concat Y*)

lemma *distinct-Par-equiv-a*: ⋀ *Bs* . ∀ *A* ∈ *set As* . *io-diagram A* ⟹ *distinct* (*concat* (*map Out As*)) ⟹ *perm As Bs* ⟹
in-out-equiv (*Parallel-list As*) (*Parallel-list Bs*)

thm *distinct-concat-perm*
thm *perm-map*

lemma *distinct-FB*: *distinct* (*In A*) ⟹ *distinct* (*In* (*FB A*))

lemma *io-distinct-FB-cat*: *io-distinct* (*A* # *Cs*) ⟹ *io-distinct* (*FB A* # *Cs*)

lemma *io-distinct-perm*: *io-distinct As* ⟹ *perm As Bs* ⟹ *io-distinct Bs*

lemma [*simp*]: *distinct* (*concat X*) ⟹ *op-list* [] *op* ⊕ (*X*) = *concat X*

lemma [*simp*]: *io-distinct As* ⟹ *perm As* (*Bs* @ *Cs*) ⟹ *io-distinct* (*FB* (*Parallel-list Bs*) # *Cs*)

lemma *io-distinct-append-a*: *io-distinct As* ⟹ *perm As* (*Bs* @ *Cs*) ⟹ *io-distinct Bs*

lemma *io-distinct-append-b*: *io-distinct As* ⟹ *perm As* (*Bs* @ *Cs*) ⟹ *io-distinct Cs*

lemma [*simp*]: *io-distinct As* ⟹ *perm As* (*Bs* @ *Cs*) ⟹ *io-diagram* (*FB* (*FB* (*Parallel-list Bs*) ||| *Parallel-list Cs*))

lemma [*simp*]: *io-distinct As* ⟹ *io-diagram* (*FB* (*Parallel-list As*))

lemma *io-distinct-set-In*[*simp*]: *io-distinct x* ⟹ *perm x* (*A* # *B* # *Bs*) ⟹ *set* (*In A*) ∩ *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

9.6 Feedbackless HBD Translation

theory *FeedbacklessHBDTranslation* **imports** *Diagrams ../RefinementReactive/Refinement*
begin
context *BaseOperationFeedbacklessVars*
begin
definition *WhileFeedbackless* =
while-stm (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*]):]

definition *TranslateHBDFeedbackless* = *WhileFeedbackless o* [−(λ *As* . *Parallel-list As*)−]

definition *ok-fbless As* = (*Deterministic As* ∧ *loop-free As* ∧ *Type-OK As*)

definition *TranslateHBDRec* = {. *ok-fbless* .}
o [:*As* ⇝ *As'* . ∃ *L* . *perm* (*VarFB* (*Parallel-list As*)) *L* ∧ *As'* = *fb-less L As* :]

lemma [*simp*]:{.*As*. *length* (*VarFB* (*Parallel-list As*)) = *w*.} (*TranslateHBDRec x*) *y* ⟹ [. − (λ*As*. *internal As* ≠ {}) .] *x y*

lemma *internal-fb-less-step*: *loop-free As* ⟹ *Type-OK As* ⟹ *A* ∈ *set As* ⟹ *out A* ∈ *internal As* ⟹ *internal* (*fb-less-step A* (*As* ⊖ [*A*])) = *internal As* − {*out A*}

lemma *ok-fbless-fb-less-step*: *ok-fbless As* ⟹ *A* ∈ *set As* ⟹ *out A* ∈ *internal As* ⟹ *ok-fbless* (*fb-less-step A* (*As* ⊖ [*A*]))

lemma *map-CompA-fb-out-less-step*: *Deterministic As* ⟹
loop-free As ⟹
Type-OK As ⟹ *A* ∈ *set As* ⟹ *out A* ∈ *internal As* ⟹ *map* (*CompA A*) (*As* ⊖ [*A*]) = *fb-out-less-step* (*out A*) *As*

lemma *length-diff*: *a* ∈ *set x* ⟹ *length* (*x* ⊖ [*a*]) < *length x*

thm *perm-cons*

lemma *perm-cons-a*: ⋀ *y* . *a* ∈ *set x* ⟹ *distinct x* ⟹ *perm* (*x* ⊖ [*a*]) *y* ⟹ *perm x* (*a* # *y*)

lemma [*simp*]: {.*As*. *length* (*VarFB* (*Parallel-list As*)) = *w*.} (*TranslateHBDRec x*) *y* ⟹
[. λ*As*. *internal As* ≠ {} .]
([:*As*⇝*As'*.∃ *A*. *A* ∈ *set As* ∧ *out A* ∈ *internal As* ∧ *As'* = *map* (*CompA A*) (*As* ⊖ [*A*]):]
({.*As*. *length* (*VarFB* (*Parallel-list As*)) < *w*.} (*TranslateHBDRec x*))) *y*

lemma *Feedbackless-Rec-While-refinement*: *TranslateHBDRec* ≤ *WhileFeedbackless*

lemma [*simp*]: *TranslateHBDRec o* [−(λ *As* . *Parallel-list As*)−] ≤ *TranslateHBDFeedbackless*

thm *FB-fb-less*(*1*)

lemma *Out-Parallel-fb-less*: ⋀ *As* . *Type-OK As* ⟹ *loop-free As* ⟹ *distinct L* ⟹ *set L* ⊆ *internal As* ⟹
Out (*Parallel-list* (*fb-less L As*)) = *concat* (*map Out As*) ⊖ *L*

lemma *io-diagram-distinct-VarFB*: *io-diagram A* ⟹ *distinct* (*VarFB A*)

theorem *fbless-correctness*: *ok-fbless As* ⟹ *perm* (*VarFB* (*Parallel-list As*)) *L* ⟹
in-equiv (*FB* (*Parallel-list As*)) (*Parallel-list* (*fb-less L As*))

lemma *Hoare-TranslateHBDRec*: *Hoare* (λ *As* . *As* = *As-init* ∧ *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* ∧ *ok-fbless As*)
TranslateHBDFeedbackless
(λ *A* . *in-equiv* (*FB* (*Parallel-list As-init*)) *A*)

end

end

9.7 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* ⇒ *bool*
assumes *maximal-def*: *maximal x* = (∀ *y* . ¬ *x* < *y*)
assumes [*simp*]: ¬ *maximal* ⊥
begin
lemma *ex-not-le-bot*[*simp*]: ∃ *a*. ¬ *a* ≤ ⊥
end

instantiation *option* :: (*type*) *order-bot-max*
begin
definition *bot-option-def*: (⊥::*'a option*) = *None*
definition *le-option-def*: ((*x*::*'a option*) ≤ *y*) = (*x* = *None* ∨ *x* = *y*)
definition *less-option-def*: ((*x*::*'a option*) < *y*) = (*x* ≤ *y* ∧ ¬ (*y* ≤ *x*))
definition *maximal-option-def*: *maximal* (*x*::*'a option*) = (∀ *y* . ¬ *x* < *y*)

instance

lemma [*simp*]: *None* ≤ *x*
end

context *order-bot*
begin
definition *is-lfp f x* = ((*f x* = *x*) ∧ (∀ *y* . *f y* = *y* ⟶ *x* ≤ *y*))
definition *emono f* = (∀ *x y*. *x* ≤ *y* ⟶ *f x* ≤ *f y*)

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

lemma *lfp-unique*: *is-lfp f x* ⟹ *is-lfp f y* ⟹ *x* = *y*

lemma *lfp-exists*: *is-lfp f x* ⟹ *Lfp f* = *x*

lemma *emono-a*: *emono f* ⟹ *x* ≤ *y* ⟹ *f x* ≤ *f y*

lemma *emono-fix*: *emono f* ⟹ *f y* = *y* ⟹ (*f* ^^ *n*) ⊥ ≤ *y*

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

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

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

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-set A* $\in A \wedge (\forall\ x \in A\ .$ *min-set A* $\leq x)$

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

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

lemma *min-set-less*: $(n::nat) \in A \Longrightarrow$ *min-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-lfp f* $((f \hat{\ }\hat{\ } n) \perp)$

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

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

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

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

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

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

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

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

9.8 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* ⟹ *mono B* ⟹ *mono C* ⟹ *mono-a f* ⟹ *mono-a* (λ*a b*. *C* (*f* (*A a*) (*B b*)))

lemma *fb-t-commute*: *mono-a f* ⟹ *mono-a g*
⟹ *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*: (⋀ *a* . *tv a* = *t* ⟹ *f a* = *g a*) ⟹ *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* ⟹ *f* ∈ *has-in-out-type x y* ⟹ *tp v* = *x*

lemma *has-type-type-in-a*: *f v* = *None* ⟹ *f* ∈ *has-in-out-type x y* ⟹ *tp v* ≠ *x*

lemma *has-type-defined*: *f* ∈ *has-in-out-type x y* ⟹ *tp v* = *x* ⟹ ∃ *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)) =
($\lambda\ v$. (*if* $tp\ v = t \# t' \#$ *ts then case* v *of* $a \# b \# v' \Rightarrow$ (*case* $f\ (b \# a \# v')$ *of Some* $(c \# d \# u) \Rightarrow$ *Some* $(d \# c \# u)$) *else None*))

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

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

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

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

thm *has-type-out-type*

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

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

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

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

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

thm *monoD*

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

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

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

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

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

lemma *mono-fb-t-aux*: $f \in \textit{mono-f} \Longrightarrow$
$\textit{le-list}\ x\ y \Longrightarrow (\bigwedge x\ y.\ \textit{tp}\ x = \textit{tp}\ y \Longrightarrow f\ x = \textit{None} \Longrightarrow f\ y = \textit{None}) \Longrightarrow \textit{mono}\ (A::'a::\textit{order} \Rightarrow 'b::\textit{fin-cpo}) \Longrightarrow \textit{mono}\ B$
$\Longrightarrow B\ (\textit{Lfp}\ (\lambda a.\ A\ (\textit{hd}\ (\textit{the}\ (f\ (B\ a\ \#\ x))))))) \leq B\ (\textit{Lfp}\ (\lambda a.\ A\ (\textit{hd}\ (\textit{the}\ (f\ (B\ a\ \#\ y))))))$

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

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

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

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

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

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

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

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

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

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

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* $\in$ *typed-func*

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

lemma *par-typed-func*[*simp*]: $f \in$ *typed-func* $\Longrightarrow$ $g \in$ *typed-func* $\Longrightarrow$ *par-f f g* $\in$ *typed-func*

lemma *fb-typed-func*[*simp*]: $f \in$ *typed-func* $\Longrightarrow$ *TI-f f* $= t \# x \Longrightarrow$ *TO-f f* $= t \# y \Longrightarrow$ *fb-f f* $\in$ *typed-func*

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

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

lemma [*simp*]: *Dup-f x* $\in$ *typed-func*

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

lemma [*simp*]: *Sink-f x* $\in$ *typed-func*

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

lemma *map-comp-assoc*: $(f \circ_m g) \circ_m h = f \circ_m (g \circ_m h)$

lemma *map-comp-id*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ $(f \circ_m$ *ID-f x*$) = f$

lemma *id-map-comp*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ $($*ID-f y* $\circ_m f) = f$

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

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

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

lemma *par-f-assoc*: $f \in$ *has-in-out-type x y* $\Longrightarrow$ $g \in$ *has-in-out-type* x' $y' \Longrightarrow$ $h \in$ *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*: $f \in has\text{-}in\text{-}out\text{-}type\ x\ y \Longrightarrow g \in has\text{-}in\text{-}out\text{-}type\ y\ z \Longrightarrow$
$f' \in has\text{-}in\text{-}out\text{-}type\ x'\ y' \Longrightarrow g' \in has\text{-}in\text{-}out\text{-}type\ y'\ z' \Longrightarrow$
$par\text{-}f\ g\ g' \circ_m par\text{-}f\ f\ f' = (par\text{-}f\ (g \circ_m f)\ (g' \circ_m f'))$

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

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

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

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

lemma [*simp*]: $TI\text{-}f\ (Sink\text{-}f\ ts) = ts$

lemma [*simp*]: $TO\text{-}f\ (Sink\text{-}f\ ts) = []$

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

lemma [*simp*]: $TI\text{-}f\ (Dup\text{-}f\ x) = x$

lemma [*simp*]: $TO\text{-}f\ (Dup\text{-}f\ x) = (x\ @\ x)$

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

lemma [*simp*]: $TO\text{-}f\ (Switch\text{-}f\ x\ y) = (y\ @\ x)$

lemma [*simp*]: $TO\text{-}f\ (ID\text{-}f\ x) = x$

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

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

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

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

lemma *AAA*: $S\ x \neq None \Longrightarrow tv\ a = t \Longrightarrow tp\ x = TI\text{-}f\ S \Longrightarrow the\ ((par\text{-}f\ (ID\text{-}f\ [t])\ S)\ (a\ \#\ x)) = a\ \#\ the\ (S\ x)$

lemma *AAAb*: $S\ x \neq None \Longrightarrow tv\ a = t \Longrightarrow tp\ x = TI\text{-}f\ S \Longrightarrow ((par\text{-}f\ (ID\text{-}f\ [t])\ S)\ (a\ \#\ x)) = Some\ (a\ \#\ the\ (S\ x))$

lemma *pref-suff-append*: $\bigwedge ts\ .\ tp\ x = ts\ @\ ts' \Longrightarrow pref\ x\ ts\ @\ suff\ x\ ts = x$

lemma [*simp*]: $Lfp\ (\lambda\ b.\ a) = a$

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

References

[1] Viorel Preoteasa, Iulia Dragomir, and Stavros Tripakis. The refinement calculus of reactive systems. *CoRR*, abs/1710.03979, 2017.

[2] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. The refinement calculus of reactive systems toolset. *CoRR*, abs/1710.08195, 2017.

[3] Viorel Preoteasa, Iulia Dragomir, and Stavros Tripakis. Type Inference of Simulink Hierarchical Block Diagrams in Isabelle. In *37th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, 2017.

[4] Viorel Preoteasa and Stavros Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

[5] Viorel Preoteasa, Iulia Dragomir, and Stavros Tripakis. A Nondeterministic and Abstract Algorithm for Translating Hierarchical Block Diagrams. *CoRR*, abs/1611.01337, November 2016.

[6] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *23rd International SPIN Symposium on Model Checking of Software (SPIN 2016)*, volume 9641 of *LNCS*, pages 38–56. Springer, April 2016.

[7] Viorel Preoteasa. Formalization of refinement calculus for reactive systems. *Archive of Formal Proofs*, October 2014. http://afp.sf.net/entries/RefinementReactive.shtml, Formal proof development.

[8] Viorel Preoteasa and Stavros Tripakis. Refinement calculus of reactive systems. In *Embedded Software (EMSOFT)*. ACM, 2014.

[9] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14:1–14:41, July 2011.

[10] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[11] Viorel Preoteasa and Ralph-Johan Back. Semantics and data refinement of invariant based programs. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/DataRefinementIBP.shtml, May 2010. Formal proof development.

[12] Ralph-Johan Back and Michael Butler. Exploring summation and product operators in the refinement calculus. In Bernhard Möller, editor, *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 128–158. Springer Berlin Heidelberg, 1995.