HOL-CSP_RS: CSP Semantics over Restriction Spaces

Benoît Ballenghien Burkhart Wolff

November 9, 2026

Abstract

We use the `Restriction_Spaces` library as a semantic foundation for the process algebra framework `HOL-CSP`, offering a complementary backend to the existing `HOLCF` infrastructure. The type of processes is instantiated as a restriction space, and we prove that it is complete in this setting. This enables the construction of fixed points for recursive process definitions without having to rely exclusively on a pointed complete partial order. Notably, some operators are constructive without being Scott-continuous, and vice versa, illustrating the genuine complementarity between the two approaches. We also show that key CSP operators are either constructive or non-destructive, and verify the admissibility of several predicates, thereby supporting automated reasoning over recursive specifications.

Contents

1 Depth Operator **1**
- 1.1 Definition . . . 1
- 1.2 Projections . . . 3
- 1.3 Proof obligation . . . 7
- 1.4 Compatibility with Refinements . . . 10
- 1.5 First Laws . . . 11
- 1.6 Monotony . . . 13
  - 1.6.1 $P \downarrow n$ is an Approximation of the P . . . 13
  - 1.6.2 Monotony of $(\downarrow)$ . . . 13
  - 1.6.3 Interpretations of Refinements . . . 15
- 1.7 Continuity . . . 16
- 1.8 Completeness . . . 17

2 Constructiveness of Prefixes **20**
- 2.1 Equality . . . 20
- 2.2 Constructiveness . . . 22

3 Non Destructiveness of Choices **22**
- 3.1 Equality . . . 22
- 3.2 Non Destructiveness . . . 24

4 Non Destructiveness of Renaming **24**
4.1 Equality . . . 24
4.2 Non Destructiveness . . . 25

5 Non Destructiveness of Sequential Composition **26**
5.1 Refinement . . . 26
5.2 Non Destructiveness . . . 28

6 Non Destructiveness of Synchronization Product **28**
6.1 Preliminaries . . . 28
6.2 Refinement . . . 35
6.3 Non Destructiveness . . . 38

7 Non Destructiveness of Throw **39**
7.1 Equality . . . 39
7.2 Refinement . . . 41
7.3 Non Destructiveness . . . 44

8 Non Destructiveness of Interrupt **47**
8.1 Refinement . . . 47
8.2 Non Destructiveness . . . 48

9 Non too Destructiveness of After **49**
9.1 Equality . . . 49
9.2 Non too Destructiveness . . . 50

10 Destructiveness of Hiding **52**
10.1 Refinement . . . 52
10.2 Destructiveness . . . 54

11 Admissibility **54**
11.1 Belonging . . . 54
11.2 Refining . . . 55
11.2.1 Transitions . . . 56

12 Higher-Order Rules **59**
12.1 Prefixes . . . 59
12.2 Choices . . . 60
12.3 Renaming . . . 62
12.4 Sequential Composition . . . 62
12.5 Synchronization Product . . . 62
12.6 Throw . . . 63
12.7 Interrupt . . . 64
12.8 After . . . 65
12.9 Illustration . . . 65

I Instantiation of the Generic Fixrec Package for Restriction-Spaces **66**
12.10 Example setup . . . 67

II Conceptual Tests: the Restriction-Space Instance of `Fixrec` 70

13 Language Selection 71

14 Unshifted Equations 71

15 Generic Shift over HOL Application 72

16 Domain-Specific Shift 72

17 Mutual Recursion 72

18 Polymorphism 73

19 Generated Theorems 74

20 Semantic Properties of the Instance 74

21 Local Contexts: Locales, Contexts, Interpretation 74

III Tests Moved from the Instance Theory (verbatim) 75

22 Test 76

23 Simulation and Tests 77

IV Annex: Running Example with Buffer over infinite Alphabet 78

24 Defining the Copy-Buffer Example 78

25 Unique Fixpoints and Least Fixpoints 79

26 The Standard Proof 80
26.1 Channels and Synchronization Sets 80
26.2 Setup for Rewriting 80
26.3 Various Samples of Refinement Proofs 80
26.4 An Example for a Highly Automated Proof. 81
26.5 Deadlock Freeness Proof 82

V Annex: A Note on State-based Processes in HOL-CSP (Restriction Spaces) 83

27 The 'Random' Key-Generator 84
27.1 Example for the basic Construction with v 84
27.2 Example in High-level Notation 85

28 Cauchy-Series as Processes **85**
28.1 The Construction in basic Steps (Again) 85
28.2 The Example again in High-level Notation 85

29 The Collatz-Function as Process **86**

30 True Concurrent Evolution of States. **86**

VI Example: Dining Philosophers **88**

31 Classic version **89**

32 Formalization with the Fixrec package **90**

VII Example: Plain Old Telephone System **91**

33 The Alphabet and Basic Types of POTS **92**

34 Auxilliaries to Substructure the Specification **93**

35 A Telephone **94**

36 A Connector with the Network **95**

37 Combining NETWORK and TELEPHONES to a SYSTEM **96**

38 A simple Model of a User **96**

39 Toplevel Proof-Goals **97**

40 Auxiliaries: Traces and Hiding in the Traces Model **100**

41 APPLICATION THREAD Methods **106**

42 WORKER THREAD METHODS **108**

43 ABSTRACT MAP SPECIFICATION **110**

44 The General Dali Model Instantiated **114**
44.1 Component Traces 115

45 Assertions **117**

46 Check: Dali is Accepted in Restriction Spaces **124**

1 Depth Operator

1.1 Definition

instantiation $process_{ptick}$ :: (*type*, *type*) *order-restriction-space*
begin

lift-definition *restriction-process*$_{ptick}$ ::
‹[('*a*, '*r*) *process*$_{ptick}$, *nat*] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
is ‹λ*P n*. ($\mathcal{F}$ *P* ∪ {(*t* @ *u*, *X*) |*t u X*. *t* ∈ $\mathcal{T}$ *P* ∧ *length t* = *n* ∧ *tF t* ∧ *ftF u*},
$\mathcal{D}$ *P* ∪ { *t* @ *u* |*t u*. *t* ∈ $\mathcal{T}$ *P* ∧ *length t* = *n* ∧ *tF t* ∧ *ftF u*})›
proof −
show ‹*?thesis P n*› (**is** ‹*is-process* (*?f*, *?d*)›) **for** *P* **and** *n*
proof (*unfold is-process-def FAILURES-def fst-conv DIVERGENCES-def snd-conv*, *intro conjI impI allI*)
show ‹([], {}) ∈ *?f*› **by** (*simp add*: *process-charn*)
next
show ‹(*t*, *X*) ∈ *?f* ⟹ *ftF t*› **for** *t X*
by *simp* (*meson ftF-append is-processT*)
next
fix *t u*
assume ‹(*t* @ *u*, {}) ∈ *?f*›
then consider ‹(*t* @ *u*, {}) ∈ $\mathcal{F}$ *P*›
| *t' u'* **where** ‹*t* @ *u* = *t'* @ *u'*› ‹*t'* ∈ $\mathcal{T}$ *P*› ‹*length t'* = *n*› ‹*tF t'*› ‹*ftF u'*› **by** *blast*
thus ‹(*t*, {}) ∈ *?f*›
proof *cases*
assume ‹(*t* @ *u*, {}) ∈ $\mathcal{F}$ *P*›
with *is-processT3* **have** ‹(*t*, {}) ∈ $\mathcal{F}$ *P*› **by** *auto*
thus ‹(*t*, {}) ∈ *?f*› **by** *fast*
next
fix *t' u'* **assume** ∗ : ‹*t* @ *u* = *t'* @ *u'*› ‹*t'* ∈ $\mathcal{T}$ *P*› ‹*length t'* = *n*› ‹*tF t'*› ‹*ftF u'*›
show ‹(*t*, {}) ∈ *?f*›
proof (*cases* ‹*t* ≤ *t'*›)
assume ‹*t* ≤ *t'*›
with ∗(*2*) *is-processT3-TR* **have** ‹*t* ∈ $\mathcal{T}$ *P*› **by** *auto*
thus ‹(*t*, {}) ∈ *?f*› **by** (*simp add*: *T-F*)
next
assume ‹¬ *t* ≤ *t'*›
with ∗(*1*) **have** ‹*t* = *t'* @ *take* (*length t* − *length t'*) *u'*›
by (*metis* (*no-types*, *lifting*) *Prefix-Order.prefixI append-Nil2 diff-is-0-eq nle-le take-all take-append take-eq-Nil*)
with ∗(*2*, *3*, *4*, *5*) **show** ‹(*t*, {}) ∈ *?f*›
by *simp* (*metis append-take-drop-id ftF-dw-closed*)
qed
qed
next

show ‹$(t,\ Y) \in ?f \wedge X \subseteq Y \Longrightarrow (t,\ X) \in ?f$› **for** $t\ X\ Y$
by *simp* (*meson is-processT4*)
next
show ‹$(t,\ X) \in ?f \wedge (\forall\, c.\ c \in Y \longrightarrow (t\ @\ [c],\ \{\}) \notin ?f) \Longrightarrow (t,\ X \cup Y) \in ?f$› **for** $t\ X\ Y$
by (*auto simp add*: *is-processT5*)
next
show ‹$(t\ @\ [\checkmark(r)],\ \{\}) \in ?f \Longrightarrow (t,\ X - \{\checkmark(r)\}) \in ?f$› **for** $t\ r\ X$
by (*simp*, *elim disjE exE*, *solves* ‹*simp add*: *is-processT6*›)
(*metis butlast-append* $event_{ptick}$*.disc*(*2*) *ftF-iff-tF-butlast*
snoc-eq-iff-butlast tF-append-iff tF-butlast)
next
from *ftF-append is-processT7 tF-append-iff*
show ‹$t \in ?d \wedge tF\ t \wedge ftF\ u \Longrightarrow t\ @\ u \in ?d$› **for** $t\ u$ **by** *fastforce*
next
from *D-F* **show** ‹$t \in ?d \Longrightarrow (t,\ X) \in ?f$› **for** $t\ X$ **by** *blast*
next
show ‹$t\ @\ [\checkmark(r)] \in ?d \Longrightarrow t \in ?d$› **for** $t\ r$
by *simp* (*metis append.right-neutral butlast-append* $event_{ptick}$*.disc*(*2*)
ftF-dw-closed is-processT9 snoc-eq-iff-butlast tF-butlast)
qed
qed

1.2 Projections

context fixes P :: ‹$('a,\ 'r)\ process_{ptick}$› **begin**

lemma $F\text{-}restriction\text{-}process_{ptick}$:
‹$\mathcal{F}\ (P \downarrow n) = \mathcal{F}\ P \cup \{(t\ @\ u,\ X)\ |t\ u\ X.\ t \in \mathcal{T}\ P \wedge length\ t = n \wedge tF\ t \wedge ftF\ u\}$›
by (*simp add*: *Failures-def FAILURES-def* $restriction\text{-}process_{ptick}$*.rep-eq*)

lemma $D\text{-}restriction\text{-}process_{ptick}$:
‹$\mathcal{D}\ (P \downarrow n) = \mathcal{D}\ P \cup \{t\ @\ u\ |t\ u.\ t \in \mathcal{T}\ P \wedge length\ t = n \wedge tF\ t \wedge ftF\ u\}$›
by (*simp add*: *Divergences-def DIVERGENCES-def* $restriction\text{-}process_{ptick}$*.rep-eq*)

lemma $T\text{-}restriction\text{-}process_{ptick}$:
‹$\mathcal{T}\ (P \downarrow n) = \mathcal{T}\ P \cup \{t\ @\ u\ |t\ u.\ t \in \mathcal{T}\ P \wedge length\ t = n \wedge tF\ t \wedge ftF\ u\}$›
using $F\text{-}restriction\text{-}process_{ptick}$ **by** (*auto simp add*: *Failures-def Traces-def TRACES-def*)

lemmas $restriction\text{-}process_{ptick}\text{-}projs = F\text{-}restriction\text{-}process_{ptick}\ D\text{-}restriction\text{-}process_{ptick}\ T\text{-}restriction\text{-}process_{ptick}$

lemma $D\text{-}restriction\text{-}process_{ptick}E$:

assumes ‹$t \in \mathcal{D}\ (P \downarrow n)$›
obtains ‹$t \in \mathcal{D}\ P$› **and** ‹*length* $t \leq n$›
| $u\ v$ **where** ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ P$› ‹*length* $u = n$› ‹*tF* u› ‹*ftF* v›
proof −
note *assms* = *that*
from ‹$t \in \mathcal{D}\ (P \downarrow n)$› **have** ‹*ftF* t› **by** (*simp add*: *D-imp-ftF*)
from ‹$t \in \mathcal{D}\ (P \downarrow n)$› **consider** ‹$t \in \mathcal{D}\ P$›
| $u\ v$ **where** ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ P$› ‹*length* $u = n$› ‹*tF* u› ‹*ftF* v›
by (*simp add*: *D-restriction-process*$_{ptick}$) *blast*
thus *thesis*
proof *cases*
show ‹$t = u\ @\ v \Longrightarrow u \in \mathcal{T}\ P \Longrightarrow$ *length* $u = n \Longrightarrow$ *tF* $u \Longrightarrow$ *ftF* $v \Longrightarrow$ *thesis*› **for** $u\ v$
by (*fact assms*(*2*))
next
show *thesis* **if** ‹$t \in \mathcal{D}\ P$›
proof (*cases* ‹*length* $t \leq n$›)
from ‹$t \in \mathcal{D}\ P$› **show** ‹*length* $t \leq n \Longrightarrow$ *thesis*› **by** (*rule assms*(*1*))
next
show *thesis* **if** ‹$\neg$ *length* $t \leq n$›
proof (*intro assms*(*2*) *exI*)
show ‹$t =$ *take* $n\ t\ @$ *drop* $n\ t$› **by** *simp*
next
show ‹*take* $n\ t \in \mathcal{T}\ P$› **by** (*metis D-T* ‹$t \in \mathcal{D}\ P$› *append-take-drop-id is-processT3-TR-append*)
next
show ‹*length* (*take* $n\ t$) $= n$› **by** (*simp add*: *min-def* ‹$\neg$ *length* $t \leq n$›)
next
show ‹*tF* (*take* $n\ t$)› **by** (*metis* ‹*ftF* t› *append-take-drop-id drop-eq-Nil2*
ftF-append-iff ‹$\neg$ *length* $t \leq n$›)
next
show ‹*ftF* (*drop* $n\ t$)› **by** (*metis* ‹*ftF* t› *append-take-drop-id drop-eq-Nil*
ftF-append-iff that)
qed
qed
qed
qed

lemma *F-restriction-process*$_{ptick}$*E* :
assumes ‹$(t,\ X) \in \mathcal{F}\ (P \downarrow n)$›
obtains ‹$(t,\ X) \in \mathcal{F}\ P$› **and** ‹*length* $t \leq n$›
| $u\ v$ **where** ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ P$› ‹*length* $u = n$› ‹*tF* u› ‹*ftF* v›
proof −
from ‹$(t,\ X) \in \mathcal{F}\ (P \downarrow n)$› **consider** ‹$(t,\ X) \in \mathcal{F}\ P$› | ‹$t \in \mathcal{D}\ (P \downarrow n)$›

unfolding *restriction-process$_{ptick}$-projs* **by** *blast*
thus *thesis*
proof *cases*
show ‹$(t, X) \in \mathcal{F}\ P \Longrightarrow$ *thesis*›
by (*metis F-T F-imp-ftF append-take-drop-id drop-eq-Nil ftF-nonempty-append-imp is-processT3-TR-append length-take min-def that*)
next
show ‹$t \in \mathcal{D}\ (P \downarrow n) \Longrightarrow$ *thesis*› **by** (*meson D-restriction-process$_{ptick}$E is-processT8 that*)
qed
qed

lemma *T-restriction-process$_{ptick}$E* :
‹$[\![t \in \mathcal{T}\ (P \downarrow n);\ t \in \mathcal{T}\ P \Longrightarrow$ *length* $t \leq n \Longrightarrow$ *thesis*;
$\bigwedge u\ v.\ t = u\ @\ v \Longrightarrow u \in \mathcal{T}\ P \Longrightarrow$ *length* $u = n \Longrightarrow$ *tF* $u \Longrightarrow$ *ftF* $v \Longrightarrow$ *thesis*$]\!] \Longrightarrow$ *thesis*›
by (*fold T-F-spec, elim F-restriction-process$_{ptick}$E*) (*simp-all add*: *T-F*)

lemmas *restriction-process$_{ptick}$-elims* =
F-restriction-process$_{ptick}$E D-restriction-process$_{ptick}$E T-restriction-process$_{ptick}$E

lemma *D-restriction-process$_{ptick}$I* :
‹$t \in \mathcal{D}\ P \vee t \in \mathcal{T}\ P \wedge$ (*length* $t = n \wedge$ *tF* $t \vee n <$ *length* $t) \Longrightarrow t \in \mathcal{D}\ (P \downarrow n)$›
by (*simp add*: *D-restriction-process$_{ptick}$*, *elim disjE conjE*)
(*solves simp*, *use ftF-Nil* **in** *blast*,
metis (*no-types*) *T-imp-ftF append-self-conv ftF-nonempty-append-imp id-take-nth-drop is-processT3-TR-append leD length-take min.absorb4 take-all-iff*)

lemma *F-restriction-process$_{ptick}$I* :
‹$(t, X) \in \mathcal{F}\ P \vee t \in \mathcal{T}\ P \wedge$ (*length* $t = n \wedge$ *tF* $t \vee n <$ *length* $t) \Longrightarrow (t, X) \in \mathcal{F}\ (P \downarrow n)$›
by (*metis* (*mono-tags*, *lifting*) *D-restriction-process$_{ptick}$I F-restriction-process$_{ptick}$ Un-iff is-processT8*)

lemma *T-restriction-process$_{ptick}$I* :
‹$t \in \mathcal{T}\ P \vee t \in \mathcal{T}\ P \wedge$ (*length* $t = n \wedge$ *tF* $t \vee n <$ *length* $t) \Longrightarrow t \in \mathcal{T}\ (P \downarrow n)$›
using *F-restriction-process$_{ptick}$I T-F-spec* **by** *blast*

lemma *F-restriction-process$_{ptick}$-Suc-length-iff-F* :

‹$(t,\ X) \in \mathcal{F}\ (P \downarrow Suc\ (length\ t)) \longleftrightarrow (t,\ X) \in \mathcal{F}\ P$›
and *D-restriction-process$_{ptick}$-Suc-length-iff-D* :
‹$t \in \mathcal{D}\ (P \downarrow Suc\ (length\ t)) \longleftrightarrow t \in \mathcal{D}\ P$›
and *T-restriction-process$_{ptick}$-Suc-length-iff-T* :
‹$t \in \mathcal{T}\ (P \downarrow Suc\ (length\ t)) \longleftrightarrow t \in \mathcal{T}\ P$›
by (*auto simp add*: *restriction-process$_{ptick}$-projs*)

lemma *length-less-in-F-restriction-process$_{ptick}$* :
‹*length* $t < n \Longrightarrow (t,\ X) \in \mathcal{F}\ (P \downarrow n) \Longrightarrow (t,\ X) \in \mathcal{F}\ P$›
by (*auto elim*: *F-restriction-process$_{ptick}$E*)

lemma *length-le-in-T-restriction-process$_{ptick}$* :
‹*length* $t \leq n \Longrightarrow t \in \mathcal{T}\ (P \downarrow n) \Longrightarrow t \in \mathcal{T}\ P$›
by (*auto elim*: *T-restriction-process$_{ptick}$E*)

lemma *length-less-in-D-restriction-process$_{ptick}$* :
‹*length* $t < n \Longrightarrow t \in \mathcal{D}\ (P \downarrow n) \Longrightarrow t \in \mathcal{D}\ P$›
by (*auto elim*: *D-restriction-process$_{ptick}$E*)

lemma *not-tF-in-F-restriction-process$_{ptick}$-iff* :
‹*length* $t \leq n \Longrightarrow \neg\ tF\ t \Longrightarrow (t,\ X) \in \mathcal{F}\ (P \downarrow n) \longleftrightarrow (t,\ X) \in \mathcal{F}$
P›
by (*auto simp add*: *F-restriction-process$_{ptick}$*)

lemma *not-tF-in-D-restriction-process$_{ptick}$-iff* :
‹*length* $t \leq n \Longrightarrow \neg\ tF\ t \Longrightarrow t \in \mathcal{D}\ (P \downarrow n) \longleftrightarrow t \in \mathcal{D}\ P$›
by (*auto simp add*: *D-restriction-process$_{ptick}$*)

lemma *D_{min}-restriction-process$_{ptick}$* :
‹$\mathcal{D}_{min}\ (P \downarrow n) = \{t.\ t \in \mathcal{D}_{min}\ P \wedge length\ t \leq n\} \cup \{t \in \mathcal{T}\ P - \mathcal{D}$
$P.\ length\ t = n \wedge tF\ t\}$›
(**is** ‹*?lhs* = *?rhs*›)
proof (*intro subset-antisym subsetI*)
fix t **assume** ‹$t \in$ *?lhs*›
thus ‹$t \in$ *?rhs*›
proof (*elim D-restriction-process$_{ptick}$E*[*OF D$_{min}$-D*])
from ‹$t \in$ *?lhs*› **show** ‹$t \in \mathcal{D}\ P \Longrightarrow length\ t \leq n \Longrightarrow t \in$ *?rhs*›
by (*auto simp add*: *Divergences$_{min}$-def min-elems-def D-restriction-process$_{ptick}$*)
next
fix $u\ v$ **assume** $*$: ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ P$› ‹*length* $u = n$› ‹$tF\ u$›
‹*ftF* v›
have ‹$v = []$›
proof (*rule ccontr*)
assume ‹$v \neq []$›
with $*$ **have** ‹*butlast* $t \in \mathcal{D}\ (P \downarrow n)$›
by (*metis D-restriction-process$_{ptick}$I butlast-append*
ftF-iff-tF-butlast is-processT7 tF-imp-ftF)

with ‹$t \in$ *?lhs*› **have** ‹$t = []$›
by (*metis Divergences*$_{min}$*-def Prefix-Order.prefixI append-butlast-last-id*
list.discI min-elems-no-list-set-list-set self-append-conv)
with ‹$v \neq []$› **show** *False* **by** (*simp add*: ∗(*1*))
qed
with ∗(*1−4*) ‹$t \in$ *?lhs*› **show** ‹$t \in$ *?rhs*›
by (*simp add*: *D-restriction-process*$_{ptick}$ *Divergences*$_{min}$*-def*
min-elems-def)
qed
next
show ‹$t \in$ *?rhs* $\Longrightarrow$ $t \in$ *?lhs*› **for** t
by (*auto simp add*: *Divergences*$_{min}$*-def min-elems-def intro*: *D-restriction-process*$_{ptick}$*I*
elim!: *D-restriction-process*$_{ptick}$*E*)
(*meson Prefix-Order.prefixI dual-order.strict-iff-not dual-order.trans*
less-length-mono,
metis Prefix-Order.strict-prefixE' is-processT7 tF-append-iff
tF-imp-ftF,
metis Prefix-Order.prefixI leD le-length-mono less-length-mono)
qed

end

1.3 Proof obligation

instance
proof *intro-classes*
fix P Q :: ‹('*a*, '*r*) *process*$_{ptick}$›
have ‹$P \downarrow 0 = \bot$› **by** (*simp add*: *BOT-iff-Nil-D D-restriction-process*$_{ptick}$)
thus ‹$P \downarrow 0 \sqsubseteq_{FD} Q \downarrow 0$› **by** *simp*
next
show ‹$P \downarrow n \downarrow m = P \downarrow min\ n\ m$› **for** P :: ‹('*a*, '*r*) *process*$_{ptick}$›
and n m
proof (*rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}\ (P \downarrow n \downarrow m) \Longrightarrow t \in \mathcal{D}\ (P \downarrow min\ n\ m)$› **for** t
by (*elim restriction-process*$_{ptick}$*-elims*)
(*auto simp add*: *D-restriction-process*$_{ptick}$ *intro*: *ftF-append*)
next
show ‹$t \in \mathcal{D}\ (P \downarrow min\ n\ m) \Longrightarrow t \in \mathcal{D}\ (P \downarrow n \downarrow m)$› **for** t
by (*elim restriction-process*$_{ptick}$*-elims*)
(*auto simp add*: *restriction-process*$_{ptick}$*-projs min-def split*:
if-split-asm)
next
fix t X **assume** ‹$(t, X) \in \mathcal{F}\ (P \downarrow n \downarrow m)$› ‹$t \notin \mathcal{D}\ (P \downarrow n \downarrow m)$›
hence ‹$(t, X) \in \mathcal{F}\ P \wedge length\ t \leq min\ n\ m$›
by (*elim F-restriction-process*$_{ptick}$*E*) (*auto simp add*: *restric-*
tion-process$_{ptick}$*-projs*)
thus ‹$(t, X) \in \mathcal{F}\ (P \downarrow min\ n\ m)$› **unfolding** *F-restriction-process*$_{ptick}$
by *blast*

next
fix t X **assume** ‹$(t,\ X) \in \mathcal{F}\ (P \downarrow min\ n\ m)$› ‹$t \notin \mathcal{D}\ (P \downarrow min\ n\ m)$›
hence ‹$(t,\ X) \in \mathcal{F}\ P \wedge length\ t \leq min\ n\ m$›
by (*elim F-restriction-process*$_{ptick}$*E*) (*auto simp add*: *restriction-process*$_{ptick}$*-projs*)
thus ‹$(t,\ X) \in \mathcal{F}\ (P \downarrow n \downarrow m)$› **unfolding** *F-restriction-process*$_{ptick}$
by *blast*
qed
next
show ‹$P \sqsubseteq_{FD} Q \Longrightarrow P \downarrow n \sqsubseteq_{FD} Q \downarrow n$› **for** $P\ Q$ :: ‹$('a,\ 'r)$ *process*$_{ptick}$› **and** n
by (*simp add*: *refine-defs restriction-process*$_{ptick}$*-projs*
flip: *T-F-spec*) *blast*
next
fix $P\ Q$ :: ‹$('a,\ 'r)$ *process*$_{ptick}$› **assume** ‹$\neg\ P \sqsubseteq_{FD} Q$›
then consider t **where** ‹$t \in \mathcal{D}\ Q$› ‹$t \notin \mathcal{D}\ P$›
| $t\ X$ **where** ‹$(t,\ X) \in \mathcal{F}\ Q$› ‹$(t,\ X) \notin \mathcal{F}\ P$›
unfolding *refine-defs* **by** *auto*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_{FD} Q \downarrow n$›
proof *cases*
fix t **assume** ‹$t \in \mathcal{D}\ Q$› ‹$t \notin \mathcal{D}\ P$›
with *D-restriction-process*$_{ptick}$*-Suc-length-iff-D*
have ‹$t \in \mathcal{D}\ (Q \downarrow Suc\ (length\ t)) \wedge t \notin \mathcal{D}\ (P \downarrow Suc\ (length\ t))$›
by *blast*
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq_{FD} Q \downarrow Suc\ (length\ t)$› **unfolding** *refine-defs* **by** *blast*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_{FD} Q \downarrow n$› **..**
next
fix $t\ X$ **assume** ‹$(t,\ X) \in \mathcal{F}\ Q$› ‹$(t,\ X) \notin \mathcal{F}\ P$›
with *F-restriction-process*$_{ptick}$*-Suc-length-iff-F*
have ‹$(t,\ X) \in \mathcal{F}\ (Q \downarrow Suc\ (length\ t)) \wedge (t,\ X) \notin \mathcal{F}\ (P \downarrow Suc\ (length\ t))$› **by** *blast*
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq_{FD} Q \downarrow Suc\ (length\ t)$› **unfolding** *refine-defs* **by** *blast*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_{FD} Q \downarrow n$› **..**
qed
qed

— Of course, we immediately recover the structure of *restriction-space*.

corollary ‹*OFCLASS*$(('a,\ 'r)\ process_{ptick},\ restriction\text{-}space\text{-}class)$›
by *intro-classes*

end

instance *process*$_{ptick}$ :: (*type*, *type*) *pcpo-restriction-space*
proof *intro-classes*

show ‹$P \downarrow 0 \sqsubseteq Q \downarrow 0$› **for** P Q :: ‹('a, 'r) $process_{ptick}$›
by (*metis below-refl restriction-0-related*)
next
show ‹$P \downarrow n \sqsubseteq Q \downarrow n$› **if** ‹$P \sqsubseteq Q$› **for** P Q :: ‹('a, 'r) $process_{ptick}$›
and n
proof (*unfold le-approx-def Refusals-after-def*, *safe*)
from ‹$P \sqsubseteq Q$›[*THEN le-approxD*(*1*)] ‹$P \sqsubseteq Q$›[*THEN le-approxD2-T*]
show ‹$t \in \mathcal{D}\ (Q \downarrow n) \Longrightarrow t \in \mathcal{D}\ (P \downarrow n)$› **for** t
by (*simp add*: *D-restriction-process*$_{ptick}$ *subset-iff*) (*metis D-T*)
next
from ‹$P \sqsubseteq Q$›[*THEN le-approxD*(*2*)] ‹$P \sqsubseteq Q$›[*THEN le-approx-lemma-T*]
show ‹$t \notin \mathcal{D}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ (Q \downarrow n)$›
and ‹$t \notin \mathcal{D}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ (Q \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ (P \downarrow n)$› **for** t X
by (*auto simp add*: *restriction-process*$_{ptick}$*-projs*)
next
from ‹$P \sqsubseteq Q$›[*THEN le-approxD*(*3*)] ‹$P \sqsubseteq Q$›[*THEN le-approxD2-T*]
show ‹$t \in \mathcal{D}_{min}\ (P \downarrow n) \Longrightarrow t \in \mathcal{T}\ (Q \downarrow n)$› **for** t
by (*auto simp add*: D_{min}*-restriction-process*$_{ptick}$ *restriction-process*$_{ptick}$*-projs*)
qed
next
fix P Q :: ‹('a, 'r) $process_{ptick}$›
assume ‹$\neg\ P \sqsubseteq Q$›
then consider t **where** ‹$t \in \mathcal{D}\ Q$› ‹$t \notin \mathcal{D}\ P$›
| t X **where** ‹$t \notin \mathcal{D}\ P$› ‹$(t, X) \in \mathcal{F}\ P \longleftrightarrow (t, X) \notin \mathcal{F}\ Q$›
| t **where** ‹$t \in \mathcal{D}_{min}\ P$› ‹$t \notin \mathcal{T}\ Q$›
unfolding *le-approx-def Refusals-after-def* **by** *blast*
thus ‹$\exists n.\ \neg\ P \downarrow n \sqsubseteq Q \downarrow n$›
proof *cases*
fix t **assume** ‹$t \in \mathcal{D}\ Q$› ‹$t \notin \mathcal{D}\ P$›
hence ‹$t \in \mathcal{D}\ (Q \downarrow Suc\ (length\ t))$› ‹$t \notin \mathcal{D}\ (P \downarrow Suc\ (length\ t))$›
by (*simp-all add*: *D-restriction-process*$_{ptick}$*-Suc-length-iff-D*)
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq Q \downarrow Suc\ (length\ t)$›
unfolding *le-approx-def* **by** *blast*
thus ‹$\exists n.\ \neg\ P \downarrow n \sqsubseteq Q \downarrow n$› **..**
next
fix t X **assume** ‹$t \notin \mathcal{D}\ P$› ‹$(t, X) \in \mathcal{F}\ P \longleftrightarrow (t, X) \notin \mathcal{F}\ Q$›
hence ‹$t \notin \mathcal{D}\ (P \downarrow Suc\ (length\ t))$›
‹$(t, X) \in \mathcal{F}\ (P \downarrow Suc\ (length\ t)) \longleftrightarrow (t, X) \notin \mathcal{F}\ (Q \downarrow Suc\ (length\ t))$›
by (*simp-all add*: *D-restriction-process*$_{ptick}$*-Suc-length-iff-D*
F-restriction-process$_{ptick}$*-Suc-length-iff-F*)
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq Q \downarrow Suc\ (length\ t)$›
unfolding *le-approx-def Refusals-after-def* **by** *blast*
thus ‹$\exists n.\ \neg\ P \downarrow n \sqsubseteq Q \downarrow n$› **..**
next
fix t **assume** ‹$t \in \mathcal{D}_{min}\ P$› ‹$t \notin \mathcal{T}\ Q$›
hence ‹$t \in \mathcal{D}_{min}\ (P \downarrow Suc\ (length\ t))$› ‹$t \notin \mathcal{T}\ (Q \downarrow Suc\ (length$

$t))\rangle$
by (*simp-all add*: D_{min}*-restriction-process*$_{ptick}$ *T-restriction-process*$_{ptick}$*-Suc-length-iff-T*)
hence ‹$\neg$ $P \downarrow$ *Suc* (*length t*) $\sqsubseteq$ $Q \downarrow$ *Suc* (*length t*)›
using $Divergences_{min}$*-def le-approx-def* **by** *blast*
thus ‹$\exists n. \neg P \downarrow n \sqsubseteq Q \downarrow n$› **..**
qed
next
show ‹*chain* $S \Longrightarrow \exists P.$ *range* S <<| P› **for** S :: ‹*nat* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$›
by (*simp add*: *cpo-class.cpo*)
next
show ‹$\exists P$ :: ('*a*, '*r*) *process*$_{ptick}$. $\forall Q.\ P \sqsubseteq Q$› **by** *blast*
qed

setup ‹*Sign.add-const-constraint* (***const-name***‹*restriction*›, *SOME* ***typ***‹('*a*, '*r*) *process*$_{ptick}$ $\Rightarrow$ *nat* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$›)›
— Only allow ($\downarrow$) for ('*a*, '*r*) *process*$_{ptick}$ (otherwise we would often have to specify).

1.4 Compatibility with Refinements

lemma *leF-restriction-process*$_{ptick}I$: ‹$P \downarrow n \sqsubseteq_F Q \downarrow n$›
if ‹$\bigwedge s\ X.\ (s, X) \in \mathcal{F}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow (s, X) \in \mathcal{F}\ (P \downarrow n)$›
proof (*unfold failure-refine-def*, *safe*)
show ‹$(s, X) \in \mathcal{F}\ (Q \downarrow n) \Longrightarrow (s, X) \in \mathcal{F}\ (P \downarrow n)$› **for** $s\ X$
proof (*elim F-restriction-process*$_{ptick}E$ *exE conjE*)
show ‹$(s, X) \in \mathcal{F}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow (s, X) \in \mathcal{F}\ (P \downarrow n)$›
by (*simp add*: *F-restriction-process*$_{ptick}$) (*meson F-restriction-process*$_{ptick}E$ *that*)
next
fix $s'\ t'$
assume ‹$s = s' @ t'$› ‹$s' \in \mathcal{T}\ Q$› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
from ‹$s' \in \mathcal{T}\ Q$› ‹*length* $s' = n$› **have** ‹$s' \in \mathcal{T}\ P$›
by (*metis F-T T-F dual-order.refl length-le-in-T-restriction-process*$_{ptick}$ *that*)
with ‹$s = s' @ t'$› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
show ‹$(s, X) \in \mathcal{F}\ (P \downarrow n)$› **by** (*simp add*: *F-restriction-process*$_{ptick}$) *blast*
qed
qed

lemma *leT-restriction-process*$_{ptick}I$: ‹$P \downarrow n \sqsubseteq_T Q \downarrow n$›
if ‹$\bigwedge s.\ s \in \mathcal{T}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow s \in \mathcal{T}\ (P \downarrow n)$›
proof (*unfold trace-refine-def*, *safe*)
show ‹$s \in \mathcal{T}\ (Q \downarrow n) \Longrightarrow s \in \mathcal{T}\ (P \downarrow n)$› **for** s
proof (*elim T-restriction-process*$_{ptick}E$ *exE conjE*)
show ‹$s \in \mathcal{T}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow s \in \mathcal{T}\ (P \downarrow n)$›

by (*simp add*: *T-restriction-process*$_{ptick}$) (*meson T-restriction-process*$_{ptick}E$ *that*)
next
fix s' t'
assume ‹$s = s'$ @ t'› ‹$s' \in \mathcal{T}$ Q› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
from ‹$s' \in \mathcal{T}$ Q› ‹*length* $s' = n$› **have** ‹$s' \in \mathcal{T}$ P›
using *length-le-in-T-restriction-process*$_{ptick}$ *that* **by** *blast*
with ‹$s = s'$ @ t'› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
show ‹$s \in \mathcal{T}$ $(P \downarrow n)$› **by** (*simp add*: *T-restriction-process*$_{ptick}$) *blast*
qed
qed

lemma *leDT-restriction-process*$_{ptick}I$: ‹$P \downarrow n \sqsubseteq_{DT}$ $Q \downarrow n$›
if ‹$\bigwedge s.\ s \in \mathcal{T}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow s \in \mathcal{T}\ (P \downarrow n)$›
and ‹$\bigwedge s.$ *length* $s \leq n \Longrightarrow s \in \mathcal{D}\ Q \Longrightarrow s \in \mathcal{D}\ (P \downarrow n)$›
proof (*rule leD-leT-imp-leDT*)
show ‹$P \downarrow n \sqsubseteq_{T}$ $Q \downarrow n$› **by** (*simp add*: *leT-restriction-process*$_{ptick}I$ *that*(*1*))
next
show ‹$P \downarrow n \sqsubseteq_{D}$ $Q \downarrow n$›
proof (*unfold divergence-refine-def*, *rule subsetI*)
show ‹$s \in \mathcal{D}\ (Q \downarrow n) \Longrightarrow s \in \mathcal{D}\ (P \downarrow n)$› **for** s
proof (*elim D-restriction-process*$_{ptick}E$ *exE conjE*)
show ‹$s \in \mathcal{D}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow s \in \mathcal{D}\ (P \downarrow n)$›
by (*simp add*: *D-restriction-process*$_{ptick}$) (*meson D-restriction-process*$_{ptick}E$ *that*(*2*))
next
fix s' t'
assume ‹$s = s'$ @ t'› ‹$s' \in \mathcal{T}$ Q› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
from ‹$s' \in \mathcal{T}$ Q› ‹*length* $s' = n$› **have** ‹$s' \in \mathcal{T}$ P›
using *length-le-in-T-restriction-process*$_{ptick}$ *that*(*1*) **by** *blast*
with ‹$s = s'$ @ t'› ‹*length* $s' = n$› ‹*tF* s'› ‹*ftF* t'›
show ‹$s \in \mathcal{D}\ (P \downarrow n)$› **by** (*simp add*: *D-restriction-process*$_{ptick}$) *blast*
qed
qed
qed

lemma *leFD-restriction-process*$_{ptick}I$: ‹$P \downarrow n \sqsubseteq_{FD}$ $Q \downarrow n$›
if ‹$\bigwedge s\ X.\ (s,\ X) \in \mathcal{F}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow (s,\ X) \in \mathcal{F}\ (P \downarrow n)$›
and ‹$\bigwedge s.\ s \in \mathcal{D}\ Q \Longrightarrow$ *length* $s \leq n \Longrightarrow s \in \mathcal{D}\ (P \downarrow n)$›
proof (*rule leF-leD-imp-leFD*)
show ‹$P \downarrow n \sqsubseteq_{F}$ $Q \downarrow n$› **by** (*simp add*: *leF-restriction-process*$_{ptick}I$ *that*(*1*))
next
show ‹$P \downarrow n \sqsubseteq_{D}$ $Q \downarrow n$› **by** (*meson T-F-spec leDT-imp-leD leDT-restriction-process*$_{ptick}I$

that)
qed

1.5 First Laws

lemma *restriction-process$_{ptick}$-0* [*simp*] : ‹$P \downarrow 0 = \bot$›
by (*simp add*: *BOT-iff-Nil-D D-restriction-process$_{ptick}$*)

lemma *restriction-process$_{ptick}$-BOT* [*simp*] : ‹($\bot$ :: ('*a*, '*r*) *process$_{ptick}$*) $\downarrow$ *n* = $\bot$›
by (*simp add*: *BOT-iff-Nil-D D-restriction-process$_{ptick}$ D-BOT*)

lemma *restriction-process$_{ptick}$-is-BOT-iff* :
‹$P \downarrow n = \bot \longleftrightarrow n = 0 \vee P = \bot$›
by (*auto simp add*: *BOT-iff-Nil-D D-restriction-process$_{ptick}$*)

lemma *restriction-process$_{ptick}$-STOP* [*simp*] : ‹*STOP* $\downarrow$ *n* = (*if n* = *0 then* $\bot$ *else STOP*)›
by (*simp add*: *STOP-iff-T T-restriction-process$_{ptick}$ T-STOP*)

lemma *restriction-process$_{ptick}$-is-STOP-iff* : ‹$P \downarrow n = STOP \longleftrightarrow n \neq 0 \wedge P = STOP$›
by (*simp add*: *STOP-iff-T T-restriction-process$_{ptick}$ set-eq-iff*)
(*metis* (*no-types*, *lifting*) *append-self-conv2 ftF-single gr0I less-numeral-extra*(*3*) *list.discI list.size*(*3*) *tF-Nil*)

lemma *restriction-process$_{ptick}$-SKIP* [*simp*] : ‹*SKIP r* $\downarrow$ *n* = (*if n* = *0 then* $\bot$ *else SKIP r*)›
by *simp* (*auto simp add*: *Process-eq-spec restriction-process$_{ptick}$-projs SKIP-projs*)

lemma *restriction-process$_{ptick}$-is-SKIP-iff* : ‹$P \downarrow n = SKIP\ r \longleftrightarrow n \neq 0 \wedge P = SKIP\ r$›
proof (*intro iffI conjI*)
show ‹$n \neq 0 \wedge P = SKIP\ r \Longrightarrow P \downarrow n = SKIP\ r$› **by** *simp*
next
show ‹$P \downarrow n = SKIP\ r \Longrightarrow n \neq 0$› **by** (*metis restriction-process$_{ptick}$-0 SKIP-neq-BOT*)
next
show ‹$P \downarrow n = SKIP\ r \Longrightarrow P = SKIP\ r$›
by (*simp add*: *Process-eq-spec set-eq-iff SKIP-projs restriction-process$_{ptick}$-projs*, *safe*; *metis*)
qed

lemma *restriction-process$_{ptick}$-SKIPS* [*simp*] : ‹*SKIPS R* $\downarrow$ *n* = (*if n* = *0 then* $\bot$ *else SKIPS R*)›

by *simp* (*auto simp add*: *Process-eq-spec restriction-process$_{ptick}$-projs SKIPS-projs*)

lemma *restriction-process$_{ptick}$-is-SKIPS-iff* : ‹$P \downarrow n = SKIPS\ R \longleftrightarrow n \neq 0 \wedge P = SKIPS\ R$›
proof (*cases* ‹$R = \{\}$›)
show ‹$R = \{\} \Longrightarrow P \downarrow n = SKIPS\ R \longleftrightarrow n \neq 0 \wedge P = SKIPS\ R$›
by (*simp add*: *restriction-process$_{ptick}$-is-STOP-iff*)
next
show ‹$P \downarrow n = SKIPS\ R \longleftrightarrow n \neq 0 \wedge P = SKIPS\ R$› **if** ‹$R \neq \{\}$›
proof (*intro iffI conjI*)
show ‹$n \neq 0 \wedge P = SKIPS\ R \Longrightarrow P \downarrow n = SKIPS\ R$› **by** *simp*
next
show ‹$P \downarrow n = SKIPS\ R \Longrightarrow n \neq 0$›
by (*metis BOT-iff-Nil-D D-SKIPS empty-iff restriction-process$_{ptick}$-0*)
next
show ‹$P \downarrow n = SKIPS\ R \Longrightarrow P = SKIPS\ R$›
by (*simp add*: *Process-eq-spec* ‹$R \neq \{\}$› *SKIPS-projs restriction-process$_{ptick}$-projs*, *safe*, *blast*)
qed
qed

1.6 Monotony

1.6.1 $P \downarrow n$ is an Approximation of the P

lemma *restriction-process$_{ptick}$-approx-self* : ‹$P \downarrow n \sqsubseteq P$›
proof (*unfold le-approx-def Refusals-after-def*, *safe*)
show ‹$t \in \mathcal{D}\ P \Longrightarrow t \in \mathcal{D}\ (P \downarrow n)$› **for** t **by** (*simp add*: *D-restriction-process$_{ptick}$*)
next
show ‹$t \notin \mathcal{D}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ P$›
for $t\ X$
by (*auto simp add*: *D-restriction-process$_{ptick}$ elim*: *F-restriction-process$_{ptick}$E*)
next
show ‹$t \notin \mathcal{D}\ (P \downarrow n) \Longrightarrow (t, X) \in \mathcal{F}\ P \Longrightarrow (t, X) \in \mathcal{F}\ (P \downarrow n)$›
for $t\ X$
by (*auto simp add*: *restriction-process$_{ptick}$-projs*)
next
show ‹$t \in \mathcal{D}_{min}\ (P \downarrow n) \Longrightarrow t \in \mathcal{T}\ P$› **for** t
by (*auto simp add*: *D_{min}-restriction-process$_{ptick}$ D_{min}-D D-T*)
qed

lemma *restriction-process$_{ptick}$-FD-self* : ‹$P \downarrow n \sqsubseteq_{FD} P$›
by (*simp add*: *le-approx-imp-le-FD restriction-process$_{ptick}$-approx-self*)

lemma *restriction-process$_{ptick}$-F-self* : ‹$P \downarrow n \sqsubseteq_{F} P$›
by (*simp add*: *restriction-process$_{ptick}$-FD-self leFD-imp-leF*)

lemma *restriction-process$_{ptick}$-D-self* : ‹$P \downarrow n \sqsubseteq_{D} P$›
by (*simp add*: *restriction-process$_{ptick}$-FD-self leFD-imp-leD*)

lemma *restriction-process$_{ptick}$-T-self* : ‹$P \downarrow n \sqsubseteq_T P$›
by (*simp add*: *restriction-process$_{ptick}$-F-self leF-imp-leT*)

lemma *restriction-process$_{ptick}$-DT-self* : ‹$P \downarrow n \sqsubseteq_{DT} P$›
by (*simp add*: *restriction-process$_{ptick}$-D-self restriction-process$_{ptick}$-T-self leD-leT-imp-leDT*)

1.6.2 Monotony of ($\downarrow$)

lemma *Suc-right-mono-restriction-process$_{ptick}$* : ‹$P \downarrow n \sqsubseteq P \downarrow Suc\ n$›
by (*metis restriction-process$_{ptick}$-approx-self restriction-chainD restriction-chain-restrictions*)

lemma *Suc-right-mono-restriction-process$_{ptick}$-FD* : ‹$P \downarrow n \sqsubseteq_{FD} P \downarrow Suc\ n$›
by (*simp add*: *Suc-right-mono-restriction-process$_{ptick}$ le-approx-imp-le-FD*)

lemma *Suc-right-mono-restriction-process$_{ptick}$-F* : ‹$P \downarrow n \sqsubseteq_F P \downarrow Suc\ n$›
by (*simp add*: *Suc-right-mono-restriction-process$_{ptick}$-FD leFD-imp-leF*)

lemma *Suc-right-mono-restriction-process$_{ptick}$-D* : ‹$P \downarrow n \sqsubseteq_D P \downarrow Suc\ n$›
by (*simp add*: *Suc-right-mono-restriction-process$_{ptick}$-FD leFD-imp-leD*)

lemma *Suc-right-mono-restriction-process$_{ptick}$-T* : ‹$P \downarrow n \sqsubseteq_T P \downarrow Suc\ n$›
by (*simp add*: *Suc-right-mono-restriction-process$_{ptick}$-FD leFD-imp-leF leF-imp-leT*)

lemma *Suc-right-mono-restriction-process$_{ptick}$-DT* : ‹$P \downarrow n \sqsubseteq_{DT} P \downarrow Suc\ n$›
by (*simp add*: *Suc-right-mono-restriction-process$_{ptick}$-D Suc-right-mono-restriction-process$_{ptick}$-T leD-leT-imp-leDT*)

lemma *le-right-mono-restriction-process$_{ptick}$* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq P \downarrow m$›
by (*metis restriction-process$_{ptick}$-approx-self restriction-chain-def-ter restriction-chain-restrictions*)

lemma *le-right-mono-restriction-process$_{ptick}$-FD* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq_{FD} P \downarrow m$›
by (*simp add*: *le-approx-imp-le-FD le-right-mono-restriction-process$_{ptick}$*)

lemma *le-right-mono-restriction-process$_{ptick}$-F* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq_F P \downarrow m$›

by (*simp add*: *leFD-imp-leF le-right-mono-restriction-process*$_{ptick}$*-FD*)

lemma *restriction-process*$_{ptick}$*-le-right-mono-D* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq_D P \downarrow m$›
by (*simp add*: *leFD-imp-leD le-right-mono-restriction-process*$_{ptick}$*-FD*)

lemma *restriction-process*$_{ptick}$*-le-right-mono-T* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq_T P \downarrow m$›
by (*simp add*: *leF-imp-leT le-right-mono-restriction-process*$_{ptick}$*-F*)

lemma *restriction-process*$_{ptick}$*-le-right-mono-DT* : ‹$n \leq m \Longrightarrow P \downarrow n \sqsubseteq_{DT} P \downarrow m$›
by (*simp add*: *restriction-process*$_{ptick}$*-le-right-mono-D restriction-process*$_{ptick}$*-le-right-mono-T leD-leT-imp-leDT*)

1.6.3 Interpretations of Refinements

lemma *ex-not-restriction-leD* : ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_D Q \downarrow n$› **if** ‹$\neg\ P \sqsubseteq_D Q$›
proof −
from ‹$\neg\ P \sqsubseteq_D Q$› **obtain** *t* **where** ‹$t \in \mathcal{D}\ Q$› ‹$t \notin \mathcal{D}\ P$›
unfolding *divergence-refine-def* **by** *blast*
hence ‹$t \in \mathcal{D}\ (Q \downarrow Suc\ (length\ t))$› ‹$t \notin \mathcal{D}\ (P \downarrow Suc\ (length\ t))$›
by (*simp-all add*: *D-restriction-process*$_{ptick}$*-Suc-length-iff-D*)
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq_D Q \downarrow Suc\ (length\ t)$›
unfolding *divergence-refine-def* **by** *blast*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_D Q \downarrow n$› **..**
qed

interpretation *PRS-leF* : *PreorderRestrictionSpace* ‹$(\downarrow)$› ‹$(\sqsubseteq_F)$›
proof *unfold-locales*
show ‹$P \downarrow 0 \sqsubseteq_F Q \downarrow 0$› **for** *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **by** *simp*
next
show ‹$P \sqsubseteq_F Q \Longrightarrow P \downarrow n \sqsubseteq_F Q \downarrow n$› **for** *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **and** *n*
by (*simp add*: *failure-refine-def F-restriction-process*$_{ptick}$ *flip*: *T-F-spec*) *blast*
next
fix *P Q* :: ‹('*a*, '*r*) *process*$_{ptick}$› **assume** ‹$\neg\ P \sqsubseteq_F Q$›
then obtain *t X* **where** ‹$(t, X) \in \mathcal{F}\ Q$› ‹$(t, X) \notin \mathcal{F}\ P$›
unfolding *failure-refine-def* **by** *auto*
with *F-restriction-process*$_{ptick}$*-Suc-length-iff-F*
have ‹$(t, X) \in \mathcal{F}\ (Q \downarrow Suc\ (length\ t)) \wedge (t, X) \notin \mathcal{F}\ (P \downarrow Suc\ (length\ t))$› **by** *blast*
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq_F Q \downarrow Suc\ (length\ t)$› **unfolding** *failure-refine-def* **by** *blast*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_F Q \downarrow n$› **..**
next
show ‹$P \sqsubseteq_F Q \Longrightarrow Q \sqsubseteq_F R \Longrightarrow P \sqsubseteq_F R$› **for** *P Q R* :: ‹('*a*, '*r*)

process$_{ptick}$›
  by (*fact trans-F*)
qed

interpretation *PRS-leT* : *PreorderRestrictionSpace* ‹(↓)› ‹($\sqsubseteq_T$)›
proof *unfold-locales*
 show ‹$P \downarrow 0 \sqsubseteq_T Q \downarrow 0$› **for** $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **by** *simp*
next
 show ‹$P \sqsubseteq_T Q \Longrightarrow P \downarrow n \sqsubseteq_T Q \downarrow n$› **for** $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$›
and *n*
  by (*auto simp add*: *trace-refine-def T-restriction-process*$_{ptick}$)
next
 fix $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **assume** ‹$\neg\ P \sqsubseteq_T Q$›
 then obtain *t* **where** ‹$t \in \mathcal{T}\ Q$› ‹$t \notin \mathcal{T}\ P$›
  unfolding *trace-refine-def* **by** *auto*
 with *T-restriction-process*$_{ptick}$*-Suc-length-iff-T*
 have ‹$t \in \mathcal{T}\ (Q \downarrow Suc\ (length\ t)) \wedge t \notin \mathcal{T}\ (P \downarrow Suc\ (length\ t))$› **by** *blast*
  hence ‹$\neg\ P \downarrow Suc\ (length\ t) \sqsubseteq_T Q \downarrow Suc\ (length\ t)$› **unfolding** *trace-refine-def* **by** *blast*
 thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_T Q \downarrow n$› **..**
next
 show ‹$P \sqsubseteq_T Q \Longrightarrow Q \sqsubseteq_T R \Longrightarrow P \sqsubseteq_T R$› **for** $P\ Q\ R$ :: ‹('*a*, '*r*) *process*$_{ptick}$›
  by (*fact trans-T*)
qed

interpretation *PRS-leDT* : *PreorderRestrictionSpace* ‹(↓)› ‹($\sqsubseteq_{DT}$)›
proof *unfold-locales*
 show ‹$P \downarrow 0 \sqsubseteq_{DT} Q \downarrow 0$› **for** $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **by** *simp*
next
  show ‹$P \sqsubseteq_{DT} Q \Longrightarrow P \downarrow n \sqsubseteq_{DT} Q \downarrow n$› **for** $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **and** *n*
  by (*auto simp add*: *refine-defs restriction-process*$_{ptick}$*-projs*)
next
 fix $P\ Q$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **assume** ‹$\neg\ P \sqsubseteq_{DT} Q$›
 hence ‹$\neg\ P \sqsubseteq_D Q \vee \neg\ P \sqsubseteq_T Q$› **unfolding** *trace-divergence-refine-def* **by** *blast*
 with *ex-not-restriction-leD PRS-leT.ex-not-restriction-related*
 have ‹$(\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_D Q \downarrow n) \vee (\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_T Q \downarrow n)$› **by** *blast*
 thus ‹$\exists\, n.\ \neg\ P \downarrow n \sqsubseteq_{DT} Q \downarrow n$›
  unfolding *trace-divergence-refine-def* **by** *blast*
next
 show ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} R \Longrightarrow P \sqsubseteq_{DT} R$› **for** $P\ Q\ R$ :: ‹('*a*, '*r*) *process*$_{ptick}$›
  by (*fact trans-DT*)

qed

1.7 Continuity

context begin

private lemma *chain-restriction-process$_{ptick}$* : ‹*chain Y* ⟹ *chain* (λ*i*. *Y i* ↓ *n*)›
by (*simp add*: *mono-restriction-below po-class.chain-def*)

private lemma *cont-prem-restriction-process$_{ptick}$* :
‹(⨆ *i*. *Y i*) ↓ *n* = (⨆ *i*. *Y i* ↓ *n*)› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
proof (*rule Process-eq-optimizedI*)
show ‹*t* ∈ 𝒟 *?lhs* ⟹ *t* ∈ 𝒟 *?rhs*› **for** *t*
by (*auto simp add*: *limproc-is-thelub chain-restriction-process$_{ptick}$ D-restriction-process$_{ptick}$ LUB-projs* ‹*chain Y*›)
next
show ‹*t* ∈ 𝒟 *?rhs* ⟹ *t* ∈ 𝒟 *?lhs*› **for** *t*
by (*simp add*: *limproc-is-thelub chain-restriction-process$_{ptick}$ D-restriction-process$_{ptick}$ LUB-projs* ‹*chain Y*›)
(*metis D-T append-eq-append-conv is-processT3-TR-append*)
next
show ‹(*t*, *X*) ∈ ℱ *?lhs* ⟹ (*t*, *X*) ∈ ℱ *?rhs*› **for** *t X*
by (*auto simp add*: *limproc-is-thelub chain-restriction-process$_{ptick}$ F-restriction-process$_{ptick}$ LUB-projs* ‹*chain Y*›)
next
show ‹(*s*, *X*) ∈ ℱ *?rhs* ⟹ (*s*, *X*) ∈ ℱ *?lhs*› **for** *s X*
by (*simp add*: *limproc-is-thelub chain-restriction-process$_{ptick}$ F-restriction-process$_{ptick}$ LUB-projs* ‹*chain Y*›)
(*metis F-T append-eq-append-conv is-processT3-TR-append*)
qed

lemma *restriction-process$_{ptick}$-cont* [*simp*] : ‹*cont* (λ*x*. *f x* ↓ *n*)› **if** ‹*cont f*›
proof (*rule contI2*)
show ‹*monofun* (λ*x*. *f x* ↓ *n*)›
by (*simp add*: *cont2monofunE mono-restriction-below monofunI* ‹*cont f*›)
next
show ‹*chain Y* ⟹ *f* (⨆ *i*. *Y i*) ↓ *n* ⊑ (⨆ *i*. *f* (*Y i*) ↓ *n*)› **for** *Y*
by (*simp add*: *ch2ch-cont cont2contlubE cont-prem-restriction-process$_{ptick}$* ‹*cont f*›)
qed

end

1.8 Completeness

Processes are actually an instance of *complete-restriction-space*.

lemma *chain-restriction-chain* :
‹*restriction-chain* $\sigma \Longrightarrow$ *chain* σ› **for** σ :: ‹*nat* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$›
by (*metis po-class.chainI restriction-process*$_{ptick}$*-approx-self restriction-chainD*)

lemma *restricted-LUB-restriction-chain-is* :
‹$(\lambda n.\ (\bigsqcup n.\ \sigma\ n) \downarrow n) = \sigma$› **if** ‹*restriction-chain* σ›
proof (*rule ext*)
have ‹*chain* σ› **by** (*simp add*: *chain-restriction-chain* ‹*restriction-chain* σ›)
moreover have ‹$\sigma = (\lambda n.\ \sigma\ n \downarrow n)$›
by (*simp add*: *restricted-restriction-chain-is* ‹*restriction-chain* σ›)
ultimately have ‹*chain* $(\lambda n.\ \sigma\ n \downarrow n)$› **by** *simp*

have ‹*length* $t < n \Longrightarrow t \in \mathcal{D}\ (\sigma\ n) \longleftrightarrow (\forall i.\ t \in \mathcal{D}\ (\sigma\ i))$› **for** $t\ n$
proof *safe*
show ‹$t \in \mathcal{D}\ (\sigma\ i)$› **if** ‹*length* $t < n$› ‹$t \in \mathcal{D}\ (\sigma\ n)$› **for** i
proof (*cases* ‹$i \leq n$›)
from ‹$t \in \mathcal{D}\ (\sigma\ n)$› ‹*chain* σ› *le-approx-def po-class.chain-mono*
show ‹$i \leq n \Longrightarrow t \in \mathcal{D}\ (\sigma\ i)$› **by** *blast*
next
from ‹*length* $t < n$› ‹$t \in \mathcal{D}\ (\sigma\ n)$› **show** ‹$\neg\ i \leq n \Longrightarrow t \in \mathcal{D}\ (\sigma\ i)$›
by (*induct n*, *simp-all*)
(*metis* ‹*restriction-chain* σ› *length-less-in-D-restriction-process*$_{ptick}$ *nat-le-linear restriction-chain-def-ter*)
qed
next
show ‹$\forall i.\ t \in \mathcal{D}\ (\sigma\ i) \Longrightarrow t \in \mathcal{D}\ (\sigma\ n)$› **by** *simp*
qed
hence $*$: ‹*length* $t < n \Longrightarrow t \in \mathcal{D}\ (\sigma\ n) \longleftrightarrow t \in \mathcal{D}\ (\bigsqcup i.\ \sigma\ i)$› **for** $t\ n$
by (*simp add*: *D-LUB* ‹*chain* σ› *limproc-is-thelub*)

show ‹$(\bigsqcup n.\ \sigma\ n) \downarrow n = \sigma\ n$› **for** n
proof (*subst* (*3*) ‹$\sigma = (\lambda n.\ \sigma\ n \downarrow n)$›, *rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}\ ((\bigsqcup n.\ \sigma\ n) \downarrow n) \Longrightarrow t \in \mathcal{D}\ (\sigma\ n \downarrow n)$› **for** t
proof (*elim D-restriction-process*$_{ptick}$*E*)
show ‹$t \in \mathcal{D}\ (\bigsqcup n.\ \sigma\ n) \Longrightarrow$ *length* $t \leq n \Longrightarrow t \in \mathcal{D}\ (\sigma\ n \downarrow n)$›
by (*simp add*: ‹*chain* σ› *limproc-is-thelub D-LUB D-restriction-process*$_{ptick}$)
next
show ‹$[\![t = u\ @\ v;\ u \in \mathcal{T}\ (\bigsqcup n.\ \sigma\ n);$ *length* $u = n$; *tF u*; *ftF v*$]\!] \Longrightarrow t \in \mathcal{D}\ (\sigma\ n \downarrow n)$› **for** $u\ v$
by (*auto simp add*: ‹*chain* σ› *limproc-is-thelub LUB-projs restriction-process*$_{ptick}$*-projs*)

```
  qed
 next
  fix t assume ‹t ∈ 𝒟 (σ n ↓ n)›
  hence ‹ftF t› by (simp add: D-imp-ftF)
  with ‹t ∈ 𝒟 (σ n ↓ n)› consider ‹length t < n› ‹t ∈ 𝒟 (σ n)›
    | t' r where ‹t = t' @ [✓(r)]› ‹tF t'› ‹length t' < n› ‹t' ∈ 𝒟 (σ
n)›
    | u v  where ‹t = u @ v› ‹u ∈ 𝒯 (σ n)› ‹length u = n› ‹tF u›
‹ftF v›
    by (auto elim!: D-restriction-process_ptick E)
      (metis D-T Suc-le-lessD antisym-conv2 append.right-neutral
        ftF-Nil ftF-nonempty-append-imp
        is-processT9 length-append-singleton not-tF-and-ftF)
  thus ‹t ∈ 𝒟 ((⨆ n. σ n) ↓ n)›
  proof cases
   show ‹length t < n ⟹ t ∈ 𝒟 (σ n) ⟹ t ∈ 𝒟 ((⨆ n. σ n) ↓ n)›
     by (simp add: D-restriction-process_ptick *)
  next
   fix t' r assume ‹t = t' @ [✓(r)]› ‹tF t'› ‹length t' < n› ‹t' ∈ 𝒟
(σ n)›
    from ‹length t' < n› ‹t' ∈ 𝒟 (σ n)› have ‹t' ∈ 𝒟 ((⨆ n. σ n) ↓
n)›
     by (simp add: D-restriction-process_ptick *)
    thus ‹t ∈ 𝒟 ((⨆ n. σ n) ↓ n)›
     by (simp add: ‹t = t' @ [✓(r)]› ‹tF t'› is-processT7)
  next
   fix u v assume ‹t = u @ v› ‹u ∈ 𝒯 (σ n)› ‹length u = n› ‹tF
u› ‹ftF v›
    from ‹length u = n› ‹u ∈ 𝒯 (σ n)› have ‹u ∈ 𝒯 (σ (Suc n))›
     by (metis length-le-in-T-restriction-process_ptick nat-le-linear
         restriction-chainD ‹restriction-chain σ›)
   from ‹chain σ› ‹u ∈ 𝒯 (σ (Suc n))› D-T le-approxD2-T po-class.chain-mono
    have ‹i ≤ Suc n ⟹ u ∈ 𝒯 (σ i)› for i by blast
    moreover have ‹Suc n < i ⟹ u ∈ 𝒯 (σ i)› for i
    by (subst T-restriction-process_ptick-Suc-length-iff-T[symmetric])
       (metis ‹length u = n› ‹u ∈ 𝒯 (σ (Suc n))›
         restriction-chain-def-bis ‹restriction-chain σ›)
    ultimately have ‹u ∈ 𝒯 (⨆ i. σ i)›
     by (metis T-LUB-2 ‹chain σ› limproc-is-thelub linorder-not-le)
    with ‹ftF v› ‹length u = n› ‹tF u› show ‹t ∈ 𝒟 ((⨆ n. σ n) ↓
n)›
     by (auto simp add: ‹t = u @ v› D-restriction-process_ptick)
  qed
 next
   show ‹(t, X) ∈ ℱ ((⨆ n. σ n) ↓ n) ⟹ t ∉ 𝒟 ((⨆ n. σ n) ↓ n)
⟹ (t, X) ∈ ℱ (σ n ↓ n)› for t X
     by (meson ‹chain σ› is-processT8 is-ub-thelub le-approxD(2)
restriction-process_ptick-approx-self)
 next
```

fix t X **assume** ‹$(t,\ X) \in \mathcal{F}\ (\sigma\ n \downarrow n)$› ‹$t \notin \mathcal{D}\ (\sigma\ n \downarrow n)$›
hence ‹*length* $t \leq n$› ‹$(t,\ X) \in \mathcal{F}\ (\sigma\ n)$› ‹$t \notin \mathcal{D}\ (\sigma\ n)$›
by (*auto elim*!: *F-restriction-process*$_{ptick}$*E simp add*: *D-restriction-process*$_{ptick}$)
thus ‹$(t,\ X) \in \mathcal{F}\ ((\bigsqcup i.\ \sigma\ i) \downarrow n)$›
by (*simp add*: *F-restriction-process*$_{ptick}$)
(*meson* ‹*chain* σ› *is-ub-thelub le-approxD*(*2*))
qed
qed

instance *process*$_{ptick}$ :: (*type*, *type*) *complete-restriction-space*
proof (*intro-classes*, *rule restriction-convergentI*)
show ‹σ $-\downarrow\rightarrow$ $(\bigsqcup i.\ \sigma\ i)$› **if** ‹*restriction-chain* σ› **for** σ :: ‹*nat* ⇒ ($'a$, $'b$) *process*$_{ptick}$›
proof (*subst restricted-LUB-restriction-chain-is*[*symmetric*])
from ‹*restriction-chain* σ› **show** ‹*restriction-chain* σ› **.**
next
from *restriction-tendsto-restrictions*
show ‹$(\lambda n.\ (\bigsqcup i.\ \sigma\ i) \downarrow n)$ $-\downarrow\rightarrow$ $(\bigsqcup i.\ \sigma\ i)$› **.**
qed
qed

This is a very powerful result. Now we can write fixed-point equations for processes like $\upsilon\ X.\ f\ X$, providing the fact that f is *constructive*.

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*restriction*›, *NONE*)›
— Back to normal.

2 Constructiveness of Prefixes

2.1 Equality

lemma *restriction-process*$_{ptick}$*-Mprefix* :
‹$\Box a \in A \rightarrow P\ a \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow \Box a{\in}A \rightarrow (P\ a \downarrow m))$› (**is** ‹*?lhs* = *?rhs*›)
proof (*cases n*)
show ‹$n = 0 \Longrightarrow$ *?lhs* = *?rhs*› **by** *simp*
next
fix m **assume** ‹$n = Suc\ m$›
show ‹*?lhs* = *?rhs*›
proof (*rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow$ $t \in \mathcal{D}$ *?rhs*› **for** t
by (*auto simp add*: ‹$n = Suc\ m$› *Mprefix-projs D-restriction-process*$_{ptick}$) *blast*
next
fix t **assume** ‹$t \in \mathcal{D}$ *?rhs*›
with *D-imp-ftF* **obtain** $a\ t'$
where ‹$a \in A$› ‹$t = ev\ a\ \#\ t'$› ‹*ftF* t'› ‹$t' \in \mathcal{D}\ (P\ a \downarrow m)$›

by (*auto simp add*: ‹$n = Suc\ m$› *D-Mprefix*)
thus ‹$t \in \mathcal{D}\ ?lhs$›
by (*simp add*: ‹$n = Suc\ m$› *D-restriction-process*$_{ptick}$ *Mprefix-projs*)
(*metis append-Cons event*$_{ptick}$*.disc*(*1*) *length-Cons tF-Cons-iff*)
next
show ‹$(t, X) \in \mathcal{F}\ ?lhs \Longrightarrow (t, X) \in \mathcal{F}\ ?rhs$› **for** $t\ X$
by (*auto simp add*: ‹$n = Suc\ m$› *restriction-process*$_{ptick}$*-projs Mprefix-projs*)
next
show ‹$(t, X) \in \mathcal{F}\ ?rhs \Longrightarrow t \notin \mathcal{D}\ ?rhs \Longrightarrow (t, X) \in \mathcal{F}\ ?lhs$› **for** $t\ X$
by (*auto simp add*: ‹$n = Suc\ m$› *restriction-process*$_{ptick}$*-projs Mprefix-projs*)
qed
qed

lemma *restriction-process*$_{ptick}$*-Mndetprefix* :
‹$\sqcap a \in A \rightarrow P\ a \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow \sqcap a \in A \rightarrow (P\ a \downarrow m))$› (**is** ‹*?lhs* = *?rhs*›)
proof (*cases n*)
show ‹$n = 0 \Longrightarrow$ *?lhs* = *?rhs*› **by** *simp*
next
fix m **assume** ‹$n = Suc\ m$›
show ‹*?lhs* = *?rhs*›
proof (*rule Process-eq-optimizedI*)
show ‹$t \in \mathcal{D}\ ?lhs \Longrightarrow t \in \mathcal{D}\ ?rhs$› **for** t
by (*auto simp add*: ‹$n = Suc\ m$› *Mndetprefix-projs D-restriction-process*$_{ptick}$) *blast*
next
fix t **assume** ‹$t \in \mathcal{D}\ ?rhs$›
with *D-imp-ftF* **obtain** $a\ t'$
where ‹$a \in A$› ‹$t = ev\ a\ \#\ t'$› ‹$ftF\ t'$› ‹$t' \in \mathcal{D}\ (P\ a \downarrow m)$›
by (*auto simp add*: ‹$n = Suc\ m$› *D-Mndetprefix′*)
thus ‹$t \in \mathcal{D}\ ?lhs$›
by (*simp add*: ‹$n = Suc\ m$› *D-restriction-process*$_{ptick}$ *Mndetprefix-projs*)
(*metis append-Cons event*$_{ptick}$*.disc*(*1*) *length-Cons tF-Cons-iff*)
next
show ‹$(t, X) \in \mathcal{F}\ ?lhs \Longrightarrow (t, X) \in \mathcal{F}\ ?rhs$› **for** $t\ X$
by (*auto simp add*: ‹$n = Suc\ m$› *restriction-process*$_{ptick}$*-projs Mndetprefix-projs split*: *if-split-asm*)
next
show ‹$(t, X) \in \mathcal{F}\ ?rhs \Longrightarrow t \notin \mathcal{D}\ ?rhs \Longrightarrow (t, X) \in \mathcal{F}\ ?lhs$› **for** $t\ X$
by (*auto simp add*: ‹$n = Suc\ m$› *restriction-process*$_{ptick}$*-projs Mndetprefix-projs split*: *if-split-asm*)
qed

qed

corollary *restriction-process$_{ptick}$-write0* :
‹$a \to P \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow a \to (P \downarrow m))$›
unfolding *write0-def* **by** (*simp add*: *restriction-process$_{ptick}$-Mprefix*)

corollary *restriction-process$_{ptick}$-write* :
‹$c!a \to P \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow c!a \to (P \downarrow m))$›
unfolding *write-def* **by** (*simp add*: *restriction-process$_{ptick}$-Mprefix*)

corollary *restriction-process$_{ptick}$-read* :
‹$c?a{\in}A \to P\ a \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow c?a{\in}A \to (P\ a \downarrow m))$›
unfolding *read-def comp-def* **by** (*simp add*: *restriction-process$_{ptick}$-Mprefix*)

corollary *restriction-process$_{ptick}$-ndet-write* :
‹$c!!a{\in}A \to P\ a \downarrow n = (case\ n\ of\ 0 \Rightarrow \bot \mid Suc\ m \Rightarrow c!!a{\in}A \to (P\ a \downarrow m))$›
unfolding *ndet-write-def comp-def* **by** (*simp add*: *restriction-process$_{ptick}$-Mndetprefix*)

2.2 Constructiveness

lemma *Mprefix-constructive* : ‹*constructive* ($\lambda P.\ \Box a \in A \to P\ a$)›
by (*auto intro*: *constructiveI*
simp add: *restriction-process$_{ptick}$-Mprefix restriction-fun-def*)

lemma *Mndetprefix-constructive* : ‹*constructive* ($\lambda P.\ \sqcap a \in A \to P\ a$)›
by (*auto intro*: *constructiveI*
simp add: *restriction-process$_{ptick}$-Mndetprefix restriction-fun-def*)

lemma *write0-constructive* : ‹*constructive* ($\lambda P.\ a \to P$)›
by (*auto intro*: *constructiveI simp add*: *restriction-process$_{ptick}$-write0*)

lemma *write-constructive* : ‹*constructive* ($\lambda P.\ c!a \to P$)›
by (*auto intro*: *constructiveI simp add*: *restriction-process$_{ptick}$-write*)

lemma *read-constructive* : ‹*constructive* ($\lambda P.\ c?a \in A \to P\ a$)›
by (*auto intro*: *constructiveI*
simp add: *restriction-process$_{ptick}$-read restriction-fun-def*)

lemma *ndet-write-constructive* : ‹*constructive* ($\lambda P.\ c!!a \in A \to P\ a$)›
by (*auto intro*: *constructiveI*
simp add: *restriction-process$_{ptick}$-ndet-write restriction-fun-def*)

3 Non Destructiveness of Choices

3.1 Equality

lemma *restriction-process$_{ptick}$-Guard* :
‹*b* **&** *P* ↓ *n* = (*if n* = *0 then* ⊥ *else b* **&** (*P* ↓ *n*))›
by (*simp add*: *Guard-def*)

lemma *restriction-process$_{ptick}$-Ndet* : ‹*P* ⊓ *Q* ↓ *n* = (*P* ↓ *n*) ⊓ (*Q* ↓ *n*)›
by (*auto simp add*: *Process-eq-spec Ndet-projs restriction-process$_{ptick}$-projs*)

lemma *restriction-process$_{ptick}$-GlobalNdet* :
‹(⊓*a* ∈ *A*. *P a*) ↓ *n* = (*if n* = *0 then* ⊥ *else* ⊓*a* ∈ *A*. (*P a* ↓ *n*))›
by *simp* (*auto simp add*: *Process-eq-spec GlobalNdet-projs restriction-process$_{ptick}$-projs*)

lemma *restriction-process$_{ptick}$-GlobalDet* :
‹(□*a* ∈ *A*. *P a*) ↓ *n* = (*if n* = *0 then* ⊥ *else* □*a* ∈ *A*. (*P a* ↓ *n*))›
(**is** ‹*?lhs* = (*if n* = *0 then* ⊥ *else ?rhs*)›)
proof (*split if-split*, *intro conjI impI*)
show ‹*n* = *0* ⟹ *?lhs* = ⊥› **by** *simp*
next
show ‹*?lhs* = *?rhs*› **if** ‹*n* ≠ *0*›
proof (*rule Process-eq-optimized-bisI*)
show ‹*t* ∈ 𝒟 *?lhs* ⟹ *t* ∈ 𝒟 *?rhs*› **for** *t*
by (*auto simp add*: *GlobalDet-projs D-restriction-process$_{ptick}$* ‹*n* ≠ *0*› *split*: *if-split-asm*)
next
show ‹*t* ∈ 𝒟 *?rhs* ⟹ *t* ∈ 𝒟 *?lhs*› **for** *t*
by (*auto simp add*: *GlobalDet-projs D-restriction-process$_{ptick}$*)
next
show ‹([], *X*) ∈ ℱ *?lhs* ⟹ ([], *X*) ∈ ℱ *?rhs*› **for** *X*
by (*auto simp add*: *restriction-process$_{ptick}$-projs F-GlobalDet*)
next
show ‹([], *X*) ∈ ℱ *?rhs* ⟹ ([], *X*) ∈ ℱ *?lhs*› **for** *X*
by (*auto simp add*: *restriction-process$_{ptick}$-projs F-GlobalDet*)
(*metis append-eq-Cons-conv event$_{ptick}$.disc*(*2*) *tF-Cons-iff*)
next
show ‹(*e* # *t*, *X*) ∈ ℱ *?lhs* ⟹ (*e* # *t*, *X*) ∈ ℱ *?rhs*› **for** *e t X*
by (*auto simp add*: ‹*n* ≠ *0*› *F-restriction-process$_{ptick}$ GlobalDet-projs split*: *if-split-asm*)
next
show ‹(*e* # *t*, *X*) ∈ ℱ *?rhs* ⟹ (*e* # *t*, *X*) ∈ ℱ *?lhs*› **for** *e t X*
by (*auto simp add*: *F-restriction-process$_{ptick}$ GlobalDet-projs*)
qed
qed

lemma *restriction-process$_{ptick}$-Det*: ‹$P \Box Q \downarrow n = (P \downarrow n) \Box (Q \downarrow n)$› (**is** ‹*?lhs* = *?rhs*›)
proof −
have ‹$P \Box Q \downarrow n = \Box i \in \{0, 1 :: nat\}. (if\ i = 0\ then\ P\ else\ Q) \downarrow n$›
by (*simp add*: *GlobalDet-distrib-unit-bis*)
also have ‹$\ldots = (if\ n = 0\ then \perp else \Box i \in \{0, 1 :: nat\}. (if\ i = 0\ then\ P \downarrow n\ else\ Q \downarrow n))$›
by (*simp add*: *restriction-process$_{ptick}$-GlobalDet if-distrib if-distribR*)
also have ‹$\ldots = (P \downarrow n) \Box (Q \downarrow n)$› **by** (*simp add*: *GlobalDet-distrib-unit-bis*)
finally show ‹$P \Box Q \downarrow n = (P \downarrow n) \Box (Q \downarrow n)$› **.**
qed

corollary *restriction-process$_{ptick}$-Sliding*: ‹$P \rhd Q \downarrow n = (P \downarrow n) \rhd (Q \downarrow n)$›
by (*simp add*: *restriction-process$_{ptick}$-Det restriction-process$_{ptick}$-Ndet Sliding-def*)

3.2 Non Destructiveness

lemma *Guard-non-destuctive* : ‹*non-destructive* ($\lambda P.\ b$ **&**$_{\mathcal{P}}$ P)›
by (*simp add*: *Guard-def non-destructiveI*)

lemma *GlobalNdet-non-destructive* : ‹*non-destructive* ($\lambda P. \sqcap a \in A.\ P\ a$)›
by (*auto intro*: *non-destructiveI*
simp add: *restriction-process$_{ptick}$-GlobalNdet restriction-fun-def*)

lemma *Ndet-non-destructive* : ‹*non-destructive* ($\lambda(P, Q).\ P \sqcap Q$)›
by (*auto intro*: *non-destructiveI*
simp add: *restriction-process$_{ptick}$-Ndet restriction-prod-def*)

lemma *GlobalDet-non-destructive* : ‹*non-destructive* ($\lambda P. \Box a \in A.\ P\ a$)›
by (*auto intro*: *non-destructiveI*
simp add: *restriction-process$_{ptick}$-GlobalDet restriction-fun-def*)

lemma *Det-non-destructive* : ‹*non-destructive* ($\lambda(P, Q).\ P \Box Q$)›
by (*auto intro*: *non-destructiveI*
simp add: *restriction-process$_{ptick}$-Det restriction-prod-def*)

corollary *Sliding-non-destructive* : ‹*non-destructive* ($\lambda(P, Q).\ P \rhd Q$)›
by (*auto intro*: *non-destructiveI*
simp add: *restriction-process$_{ptick}$-Sliding restriction-prod-def*)

4 Non Destructiveness of Renaming

4.1 Equality

lemma *restriction-process$_{ptick}$-Renaming*:
‹*Renaming P f g* ↓ *n* = *Renaming* (*P* ↓ *n*) *f g*› (**is** ‹*?lhs* = *?rhs*›)
proof (*rule Process-eq-optimizedI*)
show ‹*t* ∈ $\mathcal{D}$ *?lhs* ⟹ *t* ∈ $\mathcal{D}$ *?rhs*› **for** *t*
by (*auto simp add*: *Renaming-projs D-restriction-process$_{ptick}$*)
(*metis append.right-neutral ftF-Nil tF-map-event$_{ptick}$-iff*,
use ftF-append **in** *blast*)
next
show ‹*t* ∈ $\mathcal{D}$ *?rhs* ⟹ *t* ∈ $\mathcal{D}$ *?lhs*› **for** *t*
by (*auto simp add*: *Renaming-projs D-restriction-process$_{ptick}$*
ftF-append tF-map-event$_{ptick}$-iff)
next
fix *t X* **assume** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› ‹*t* ∉ $\mathcal{D}$ *?lhs*›
then obtain *u* **where** ‹(*u*, *map-event$_{ptick}$ f g* −‘ *X*) ∈ $\mathcal{F}$ *P*› ‹*t* = *map* (*map-event$_{ptick}$ f g*) *u*›
by (*simp add*: *Renaming-projs restriction-process$_{ptick}$-projs*) *blast*
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?rhs*› **by** (*auto simp add*: *F-Renaming F-restriction-process$_{ptick}$*)
next
fix *t X* **assume** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?rhs*› ‹*t* ∉ $\mathcal{D}$ *?rhs*›
then obtain *u* **where** ‹(*u*, *map-event$_{ptick}$ f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ↓ *n*)›
‹*t* = *map* (*map-event$_{ptick}$ f g*) *u*›
unfolding *Renaming-projs* **by** *blast*
from ‹(*u*, *map-event$_{ptick}$ f g* −‘ *X*) ∈ $\mathcal{F}$ (*P* ↓ *n*)›
consider ‹*u* ∈ $\mathcal{D}$ (*P* ↓ *n*)› | ‹(*u*, *map-event$_{ptick}$ f g* −‘ *X*) ∈ $\mathcal{F}$ *P*›
unfolding *restriction-process$_{ptick}$-projs* **by** *blast*
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
proof *cases*
assume ‹*u* ∈ $\mathcal{D}$ (*P* ↓ *n*)›
hence ‹*t* ∈ $\mathcal{D}$ *?rhs*›
proof (*elim D-restriction-process$_{ptick}$E*)
from ‹*t* = *map* (*map-event$_{ptick}$ f g*) *u*› **show** ‹*u* ∈ $\mathcal{D}$ *P* ⟹ *t* ∈ $\mathcal{D}$ *?rhs*›
by (*cases ‹tF u›*, *simp-all add*: *D-Renaming D-restriction-process$_{ptick}$*)
(*use ftF-Nil* **in** *blast*,
metis D-imp-ftF butlast-snoc div-butlast-when-non-tF-iff
ftF-iff-tF-butlast ftF-single map-append
ftF-map-event$_{ptick}$-iff not-tF-and-ftF)
next
show ‹⟦*u* = *v* @ *w*; *v* ∈ $\mathcal{T}$ *P*; *length v* = *n*; *tF v*; *ftF w*⟧ ⟹ *t* ∈ $\mathcal{D}$ *?rhs*› **for** *v w*
by (*simp add*: *D-Renaming D-restriction-process$_{ptick}$ ‹t = map* (*map-event$_{ptick}$ f g*) *u›*)
(*use ftF-Nil ftF-map-event$_{ptick}$-iff* **in** *blast*)
qed
with ‹*t* ∉ $\mathcal{D}$ *?rhs*› **have** *False* **..**
thus ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*› **..**

next
show ‹$(u,\ map\text{-}event_{ptick}\ f\ g\ -`\ X) \in \mathcal{F}\ P \Longrightarrow (t,\ X) \in \mathcal{F}\ (Renaming\ P\ f\ g \downarrow n)$›
by (*auto simp add*: *F-restriction-process*$_{ptick}$ *F-Renaming* ‹$t = map\ (map\text{-}event_{ptick}\ f\ g)\ u$›)
qed
qed

4.2 Non Destructiveness

lemma *Renaming-non-destructive* [*simp*] :
‹*non-destructive* ($\lambda P.\ Renaming\ P\ f\ g$)›
by (*auto intro*: *non-destructiveI simp add*: *restriction-process*$_{ptick}$*-Renaming*)

5 Non Destructiveness of Sequential Composition

5.1 Refinement

lemma *restriction-process*$_{ptick}$*-Seq-FD* :
‹P **;** $Q \downarrow n \sqsubseteq_{FD} (P \downarrow n)$ **;** $(Q \downarrow n)$› (**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
proof −
have $*$: ‹$t \in \mathcal{D}\ (P \downarrow n) \Longrightarrow t \in \mathcal{D}$ *?lhs*› **for** t
by (*elim D-restriction-process*$_{ptick}$*E*)
(*auto simp add*: *Seq-projs D-restriction-process*$_{ptick}$)
{ fix $t\ u\ v\ r\ w\ x$
assume ‹u @ [✓(r)] $\in \mathcal{T}\ P$› ‹*length* $u < n$› ‹$v = w$ @ x› ‹$w \in \mathcal{T}$ Q›
‹*length* $w = n$› ‹*tF* w› ‹*ftF* x› ‹$t = u$ @ v›
hence ‹$t = (u$ @ *take* $(n - length\ u)\ w)$ @ *drop* $(n - length\ u)\ w$ @ $x \wedge$
u @ *take* $(n - length\ u)\ w \in \mathcal{T}\ (P$ **;** $Q) \wedge$
length $(u$ @ *take* $(n - length\ u)\ w) = n \wedge$
tF $(u$ @ *take* $(n - length\ u)\ w) \wedge$ *ftF* (*drop* $(n - length\ u)\ w$ @ $x)$›
by (*simp add*: ‹$t = u$ @ v› *T-Seq*)
(*metis append-T-imp-tF append-take-drop-id ftF-append*
is-processT3-TR-append list.distinct(1) *tF-append-iff*)
with *D-restriction-process*$_{ptick}$ **have** ‹$t \in \mathcal{D}$ *?lhs*› **by** *blast*
} **note** $** = this$

show ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›
proof (*unfold refine-defs*, *safe*)
show *div* : ‹$t \in \mathcal{D}$ *?lhs*› **if** ‹$t \in \mathcal{D}$ *?rhs*› **for** t
proof −
from ‹$t \in \mathcal{D}$ *?rhs*› **consider** ‹$t \in \mathcal{D}\ (P \downarrow n)$›

```
      | u v r where ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 (P ↓ n)› ‹v ∈ 𝒟
(Q ↓ n)›
       unfolding D-Seq by blast
     thus ‹t ∈ 𝒟 ?lhs›
     proof cases
       show ‹t ∈ 𝒟 (P ↓ n) ⟹ t ∈ 𝒟 ?lhs› by (fact ∗)
     next
       fix u v r assume ‹t = u @ v› ‹u @ [✓(r)] ∈ 𝒯 (P ↓ n)› ‹v ∈
𝒟 (Q ↓ n)›
       from ‹u @ [✓(r)] ∈ 𝒯 (P ↓ n)› consider ‹u @ [✓(r)] ∈ 𝒟 (P
↓ n)› | ‹u @ [✓(r)] ∈ 𝒯 P› ‹length u < n›
       by (elim T-restriction-process_ptick E) (auto simp add: D-restriction-process_ptick)
       thus ‹t ∈ 𝒟 ?lhs›
       proof cases
         show ‹u @ [✓(r)] ∈ 𝒟 (P ↓ n) ⟹ t ∈ 𝒟 ?lhs›
           by (metis ∗ D-imp-ftF ‹t = u @ v› ‹v ∈ 𝒟 (Q ↓ n)›
               ftF-append-iff is-processT7 is-processT9 not-Cons-self)
       next
        from ‹v ∈ 𝒟 (Q ↓ n)› show ‹u @ [✓(r)] ∈ 𝒯 P ⟹ length u
< n ⟹ t ∈ 𝒟 ?lhs›
         proof (elim D-restriction-process_ptick E exE conjE)
           show ‹u @ [✓(r)] ∈ 𝒯 P ⟹ v ∈ 𝒟 Q ⟹ t ∈ 𝒟 ?lhs›
           by (simp add: ‹t = u @ v› D-restriction-process_ptick D-Seq)
blast
         next
           show ‹⟦u @ [✓(r)] ∈ 𝒯 P; length u < n; v = w @ x; w ∈
𝒯 Q;
                 length w = n; tF w; ftF x⟧ ⟹ t ∈ 𝒟 ?lhs› for w x
             using ∗∗ ‹t = u @ v› by blast
         qed
       qed
     qed
   qed

   have mono : ‹(P ↓ n) ; (Q ↓ n) ⊑ P ; Q›
     by (simp add: fun-below-iff mono-Seq restriction-fun-def
         restriction-process_ptick-approx-self)

   show ‹(t, X) ∈ ℱ ?lhs› if ‹(t, X) ∈ ℱ ?rhs› for t X
       by (meson F-restriction-process_ptick I div is-processT8 mono
le-approxD(2) that)
  qed
qed


corollary restriction-process_ptick-MultiSeq-FD :
  ‹(SEQ l ∈@ L. P l) ↓ n ⊑_FD SEQ l ∈@ L. (P l ↓ n)›
proof (induct L rule: rev-induct)
  show ‹(SEQ l ∈@ []. P l) ↓ n ⊑_FD SEQ l ∈@ []. (P l ↓ n)› by simp
```

next
fix *a L*
assume *hyp*: ‹(*SEQ l* ∈@ *L*. *P l*) ↓ *n* ⊑$_{FD}$ *SEQ l* ∈@ *L*. (*P l* ↓ *n*)›
have ‹(*SEQ l* ∈@ (*L* @ [*a*]). *P l*) ↓ *n* = (*SEQ l* ∈@ *L*. *P l* ; *P a*) ↓ *n*› **by** *simp*
also have ‹... ⊑$_{FD}$ *SEQ l* ∈@ *L*. (*P l* ↓ *n*) ; (*P a* ↓ *n*)›
by (*fact trans-FD*[*OF restriction-process*$_{ptick}$*-Seq-FD mono-Seq-FD*[*OF hyp FD-refl*]])
also have ‹... = *SEQ l*∈@(*L* @ [*a*]). (*P l* ↓ *n*)› **by** *simp*
finally show ‹(*SEQ l* ∈@ (*L* @ [*a*]). *P l*) ↓ *n* ⊑$_{FD}$ ...› .
qed

5.2 Non Destructiveness

lemma *Seq-non-destructive* :
‹*non-destructive* (λ(*P* :: ('*a*, '*r*) *process*$_{ptick}$, *Q*). *P* ; *Q*)›
proof (*rule order-non-destructiveI*, *clarify*)
fix *P P' Q Q'* :: ‹('*a*, '*r*) *process*$_{ptick}$› **and** *n*
assume ‹(*P*, *Q*) ↓ *n* = (*P'*, *Q'*) ↓ *n*› ‹*0* < *n*›
hence ‹*P* ↓ *n* = *P'* ↓ *n*› ‹*Q* ↓ *n* = *Q'* ↓ *n*›
by (*simp-all add*: *restriction-prod-def*)
show ‹*P* ; *Q* ↓ *n* ⊑$_{FD}$ *P'* ; *Q'* ↓ *n*›
proof (*rule leFD-restriction-process*$_{ptick}$*I*)
show ‹*t* ∈ 𝒟 (*P'* ; *Q'*) ⟹ *t* ∈ 𝒟 (*P* ; *Q* ↓ *n*)› **for** *t*
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*P* ↓ *n* = *P'* ↓ *n*› ‹*Q* ↓ *n* = *Q'* ↓ *n*›
in-mono le-FD-D(*1*) *mono-Seq-FD restriction-process*$_{ptick}$*-FD-self*
restriction-process$_{ptick}$*-Seq-FD*)
next
show ‹(*s*, *X*) ∈ ℱ (*P'* ; *Q'*) ⟹ (*s*, *X*) ∈ ℱ (*P* ; *Q* ↓ *n*)› **for** *s X*
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*P* ↓ *n* = *P'* ↓ *n*› ‹*Q* ↓ *n* = *Q'* ↓ *n*›
failure-refine-def leFD-imp-leF mono-Seq-FD
restriction-process$_{ptick}$*-FD-self*
restriction-process$_{ptick}$*-Seq-FD subset-iff*)
qed
qed

6 Non Destructiveness of Synchronization Product

6.1 Preliminaries

lemma *D-Sync-optimized* :
‹𝒟 (*P* ⟦*A*⟧ *Q*) =
{*v* @ *w* |*t u v w*. *tF v* ∧ *ftF w* ∧

```
                v setinterleaves ((t, u), range tick ∪ ev ‘ A) ∧
                (t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P)}›
  (is ‹- = ?rhs›)
proof (intro subset-antisym subsetI)
  show ‹d ∈ ?rhs ⟹ d ∈ 𝒟 (P ⟦A⟧ Q)› for d
    by (auto simp add: D-Sync)
next
  fix d assume ‹d ∈ 𝒟 (P ⟦A⟧ Q)›
  then obtain t u v w
    where * : ‹d = v @ w› ‹ftF w› ‹tF v ∨ w = []›
      ‹v setinterleaves ((t, u), range tick ∪ ev ‘ A)›
      ‹t ∈ 𝒟 P ∧ u ∈ 𝒯 Q ∨ t ∈ 𝒟 Q ∧ u ∈ 𝒯 P›
    unfolding D-Sync by blast
  show ‹d ∈ ?rhs›
  proof (cases ‹tF v›)
    from * show ‹tF v ⟹ d ∈ ?rhs› by blast
  next
    assume ‹¬ tF v›
    with *(1, 3) have ‹w = []› ‹d = v› by simp-all
    from D-imp-ftF ‹d = v› ‹d ∈ 𝒟 (P ⟦A⟧ Q)›
    have ‹ftF v› by blast
    with ‹¬ tF v› obtain r v' where ‹v = v' @ [✓(r)]›
      by (meson not-tF-and-ftF)
    with *(4) obtain t' u'
      where ** : ‹t = t' @ [✓(r)]› ‹u = u' @ [✓(r)]›
        ‹v' setinterleaves ((t', u'), range tick ∪ ev ‘ A)›
      by (simp add: ‹v = v' @ [✓(r)]›)
        (meson *(5) D-imp-ftF SyncWithTick-imp-NTF T-imp-ftF)
    have ‹t' ∈ 𝒟 P ∧ u' ∈ 𝒯 Q ∨ t' ∈ 𝒟 Q ∧ u' ∈ 𝒯 P›
      by (metis *(5) **(1,2) is-processT3-TR-append is-processT9)
    with **(3) ‹d = v› ‹ftF v› ‹v = v' @ [✓(r)]›
      ftF-nonempty-append-imp show ‹d ∈ ?rhs› by blast
  qed
qed

lemma tF-interleave-iff :
  ‹t setinterleaves ((u, v), S) ⟹ tF t ⟷ tF u ∧ tF v›
  by (induct ‹(u, S, v)› arbitrary: t u v rule: setinterleaving.induct)
    (auto split: if-split-asm option.split-asm)

lemma interleave-subsetL :
  ‹tF t ⟹ {a. ev a ∈ set u} ⊆ A ⟹
   t setinterleaves ((u, v), range tick ∪ ev ‘ A) ⟹ t = v›
  for t u v :: ‹('a, 'r) trace_ptick›
proof (induct ‹(u, range tick ∪ ev ‘ A :: ('a, 'r) refusal_ptick, v)›
    arbitrary: t u v rule: setinterleaving.induct)
  case 1 thus ?case by simp
next
 case (2 y v) thus ?case by (auto simp add: image-iff split: if-split-asm)
```

next
case (*3 x u*) **thus** *?case*
by (*simp add*: *image-iff subset-iff split*: *if-split-asm*)
(*metis* (*mono-tags*, *lifting*) $event_{ptick}$.*exhaust*)
next
case (*4 x u y v*)
from *4.prems* **show** *?case*
apply (*simp add*: *subset-iff split*: *if-split-asm*)
apply (*metis* (*no-types*, *lifting*) *4.hyps*(*1*) *Un-iff*
mem-Collect-eq subsetI tF-Cons-iff)
apply (*metis* (*no-types*, *lifting*) *4.hyps*(*2*,*4*) *4.prems*(*2*,*3*) *SyncHd-Tl*
SyncSameHdTl list.sel(*1*) *setinterleaving-dual tF-Cons-iff*)
by (*metis* $event_{ptick}$.*exhaust imageI rangeI*)+
qed

lemma *interleave-subsetR* :
‹*tF t* $\Longrightarrow$ {*a*. *ev a* $\in$ *set v*} $\subseteq$ *A* $\Longrightarrow$
t setinterleaves ((*u*, *v*), *range tick* $\cup$ *ev* ‘ *A*) $\Longrightarrow$ *t* = *u*›
by (*simp add*: *interleave-subsetL setinterleaving-dual*)

lemma *interleave-imp-lengthLR-le* :
‹*t setinterleaves* ((*u*, *v*), *S*) $\Longrightarrow$
length u $\leq$ *length t* $\wedge$ *length v* $\leq$ *length t*›
by (*induct* ‹(*u*, *S*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving.induct*;
simp split: *if-split-asm*; *use nat-le-linear not-less-eq-eq* **in** *fastforce*)

lemma *interleave-le-prefixLR* :
‹*t setinterleaves* ((*u*, *v*), *S*) $\Longrightarrow$ *u*′ $\leq$ *u* $\Longrightarrow$ *v*′ $\leq$ *v* $\Longrightarrow$
($\exists$ *t*′ $\leq$ *t*. $\exists$ *v*″ $\leq$ *v*′. *t*′ *setinterleaves* ((*u*′, *v*″), *S*)) $\vee$
($\exists$ *t*′ $\leq$ *t*. $\exists$ *u*″ $\leq$ *u*′. *t*′ *setinterleaves* ((*u*″, *v*′), *S*))›
proof (*induct* ‹(*u*, *S*, *v*)›
arbitrary: *t u u*′ *v v*′ *rule*: *setinterleaving.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 y v*)
thus *?case* **by** (*simp split*: *if-split-asm*)
(*metis si-empty1 insert-iff Nil-le*)
next
case (*3 x u*)
thus *?case* **by** (*simp split*: *if-split-asm*)
(*metis si-empty1 insert-iff Nil-le*)
next
case (*4 x u y v*)
show *?case*
proof (*cases* ‹*u*′ = [] $\vee$ *v*′ = []›)
show ‹*u*′ = [] $\vee$ *v*′ = [] $\Longrightarrow$ *?case*› **by** *force*
next

```
    assume ‹¬ (u' = [] ∨ v' = [])›
    with 4.prems(2, 3)
    obtain u'' v'' where ‹u' = x # u''› ‹u'' ≤ u› ‹v' = y # v''› ‹v''
≤ v›
      by (meson Prefix-Order.prefix-Cons)
    with 4.prems(1) consider (both-in)    t' where ‹x ∈ S› ‹y ∈ S›
‹x = y› ‹t = x # t'›
      ‹t' setinterleaves ((u, v), S)›
    |      (inR-mvL)    t' where ‹x ∉ S› ‹y ∈ S› ‹t = x # t'›
      ‹t' setinterleaves ((u, y # v), S)›
    |      (inL-mvR)    t' where ‹x ∈ S› ‹y ∉ S› ‹t = y # t'›
      ‹t' setinterleaves ((x # u, v), S)›
    |      (notin-mvL) t' where ‹x ∉ S› ‹y ∉ S› ‹t = x # t'›
      ‹t' setinterleaves ((u, y # v), S)›
    |      (notin-mvR) t' where ‹x ∉ S› ‹y ∉ S› ‹t = y # t'›
      ‹t' setinterleaves ((x # u, v), S)›
      by (auto split: if-split-asm)
    thus ?case
    proof cases
      case both-in
      from 4.hyps(1)[OF both-in(1−3, 5) ‹u'' ≤ u› ‹v'' ≤ v›]
      show ?thesis
      proof (elim disjE exE conjE)
        fix t'' v'''
        assume ‹t'' ≤ t'› ‹v''' ≤ v''› ‹t'' setinterleaves ((u'', v'''), S)›
        hence ‹y # t'' ≤ t ∧ y # v''' ≤ v' ∧
              (y # t'') setinterleaves ((u', y # v'''), S)›
          by (simp add: ‹u' = x # u''› ‹v' = y # v''› both-in(2−4))
        thus ?thesis by blast
      next
        fix t'' u'''
        assume ‹t'' ≤ t'› ‹u''' ≤ u''› ‹t'' setinterleaves ((u''', v''), S)›
        hence ‹x # t'' ≤ t ∧ x # u''' ≤ u' ∧
              (x # t'') setinterleaves ((x # u''', v'), S)›
          by (simp add: ‹u' = x # u''› ‹v' = y # v''› both-in(2−4))
        thus ?thesis by blast
      qed
    next
      case inR-mvL
     from 4.hyps(5)[simplified, OF inR-mvL(1, 2 ,4) ‹u'' ≤ u› ‹v' ≤
y # v›]
      show ?thesis
      proof (elim disjE exE conjE)
        fix t'' v'''
        assume ‹t'' ≤ t'› ‹v''' ≤ v'› ‹t'' setinterleaves ((u'', v'''), S)›
        hence ‹x # t'' ≤ t ∧ v''' ≤ v' ∧
              (x # t'') setinterleaves ((u', v'''), S)›
          by (cases v''') (simp-all add: ‹u' = x # u''› ‹v' = y # v''›
inR-mvL(1, 3))
```

```
      thus ?thesis by blast
    next
      fix t'' u'''
      assume ‹t'' ≤ t'› ‹u''' ≤ u''› ‹t'' setinterleaves ((u''', v'), S)›
      hence ‹x # t'' ≤ t ∧ x # u''' ≤ u' ∧
            (x # t'') setinterleaves ((x # u''', v'), S)›
        by (simp add: ‹u' = x # u''› ‹v' = y # v''› inR-mvL(1, 3))
      thus ?thesis by blast
    qed
  next
    case inL-mvR
    from 4.hyps(2)[OF inL-mvR(1, 2, 4) ‹u' ≤ x # u› ‹v'' ≤ v›]
    show ?thesis
    proof (elim disjE exE conjE)
      fix t'' v'''
      assume ‹t'' ≤ t'› ‹v''' ≤ v''› ‹t'' setinterleaves ((u', v'''), S)›
      hence ‹y # t'' ≤ t ∧ y # v''' ≤ v' ∧
            (y # t'') setinterleaves ((u', y # v'''), S)›
        by (simp add: ‹u' = x # u''› ‹v' = y # v''› inL-mvR(2, 3))
      thus ?thesis by blast
    next
      fix t'' u'''
      assume ‹t'' ≤ t'› ‹u''' ≤ u'› ‹t'' setinterleaves ((u''', v''), S)›
      hence ‹y # t'' ≤ t ∧ u''' ≤ u' ∧
            (y # t'') setinterleaves ((u''', v'), S)›
        by (cases u''') (simp-all add: ‹u' = x # u''› ‹v' = y # v''›
inL-mvR(2, 3))
      thus ?thesis by blast
    qed
  next
    case notin-mvL
    from 4.hyps(3)[OF notin-mvL(1, 2, 4) ‹u'' ≤ u› ‹v' ≤ y # v›]
    show ?thesis
    proof (elim disjE exE conjE)
      fix t'' v'''
      assume ‹t'' ≤ t'› ‹v''' ≤ v'› ‹t'' setinterleaves ((u'', v'''), S)›
      hence ‹x # t'' ≤ t ∧ v''' ≤ v' ∧
            (x # t'') setinterleaves ((u', v'''), S)›
        by (cases v''') (simp-all add: ‹u' = x # u''› ‹v' = y # v''›
notin-mvL(1, 3))
      thus ?thesis by blast
    next
      fix t'' u'''
      assume ‹t'' ≤ t'› ‹u''' ≤ u''› ‹t'' setinterleaves ((u''', v'), S)›
      hence ‹x # t'' ≤ t ∧ x # u''' ≤ u' ∧
            (x # t'') setinterleaves ((x # u''', v'), S)›
        by (simp add: ‹u' = x # u''› ‹v' = y # v''› notin-mvL(1, 3))
      thus ?thesis by blast
    qed
```

```
    next
      case notin-mvR
      from 4.hyps(4)[OF notin-mvR(1, 2, 4) ‹u' ≤ x # u› ‹v'' ≤ v›]
      show ?thesis
      proof (elim disjE exE conjE)
        fix t'' v'''
        assume ‹t'' ≤ t'› ‹v''' ≤ v''› ‹t'' setinterleaves ((u', v'''), S)›
        hence ‹y # t'' ≤ t ∧ y # v''' ≤ v' ∧
              (y # t'') setinterleaves ((u', y # v'''), S)›
         by (simp add: ‹u' = x # u''› ‹v' = y # v''› notin-mvR(2, 3))
        thus ?thesis by blast
      next
        fix t'' u'''
        assume ‹t'' ≤ t'› ‹u''' ≤ u'› ‹t'' setinterleaves ((u''', v''), S)›
        hence ‹y # t'' ≤ t ∧ u''' ≤ u' ∧
              (y # t'') setinterleaves ((u''', v'), S)›
          by (cases u''') (simp-all add: ‹u' = x # u''› ‹v' = y # v''›
notin-mvR(2, 3))
        thus ?thesis by blast
      qed
    qed
  qed
qed


lemma restriction-process_ptick-Sync-FD-div-oneside :
  assumes ‹tF u› ‹ftF v› ‹t-P ∈ 𝒟 (P ↓ n)› ‹t-Q ∈ 𝒯 (Q ↓ n)›
    ‹u setinterleaves ((t-P, t-Q), range tick ∪ ev ‘ A)›
  shows ‹u @ v ∈ 𝒟 (P ⟦A⟧ Q ↓ n)›
proof (insert assms(3, 4), elim D-restriction-process_ptickE T-restriction-process_ptickE)
  from assms(1, 2, 5) show ‹t-P ∈ 𝒟 P ⟹ t-Q ∈ 𝒯 Q ⟹ u @ v
∈ 𝒟 (P ⟦A⟧ Q ↓ n)›
    by (auto simp add: D-restriction-process_ptick D-Sync)
next
  fix t-Q' t-Q''
  assume ∗ : ‹t-P ∈ 𝒟 P› ‹length t-P ≤ n› ‹t-Q = t-Q' @ t-Q''›
    ‹t-Q' ∈ 𝒯 Q› ‹length t-Q' = n› ‹tF t-Q'› ‹ftF t-Q''›
  from ‹t-Q = t-Q' @ t-Q''› have ‹t-Q' ≤ t-Q› by simp
  from interleave-le-right[OF assms(5) this]
  obtain t-P' t-P'' u' u''
    where ∗∗ : ‹u = u' @ u''› ‹t-P = t-P' @ t-P''›
      ‹u' setinterleaves ((t-P', t-Q'), range tick ∪ ev ‘ A)›
    by (meson Prefix-Order.prefixE)
  from assms(1) ‹u = u' @ u''› have ‹tF u'› by auto
  moreover from ∗(1,4) ∗∗(2,3) have ‹u' ∈ 𝒯 (P ⟦A⟧ Q)›
    by (simp add: T-Sync) (metis D-T is-processT3-TR-append)
  moreover have ‹length t-Q' ≤ length u'›
    using ∗∗(3) interleave-imp-lengthLR-le by blast
  ultimately have ‹u' ∈ 𝒟 (P ⟦A⟧ Q ↓ n)›
```

by (*metis* ∗(*5*) *D-restriction-process*$_{ptick}$*I nless-le*)
with ∗∗(*1*) *assms*(*1*, *2*) **show** ‹*u* @ *v* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
by (*metis is-processT7 tF-append-iff tF-imp-ftF*)
next
fix *t-P′ t-P″*
assume ∗ : ‹*t-P* = *t-P′* @ *t-P″*› ‹*t-P′* ∈ $\mathcal{T}$ *P*› ‹*length t-P′* = *n*›
‹*tF t-P′*› ‹*ftF t-P″*› ‹*t-Q* ∈ $\mathcal{T}$ *Q*› ‹*length t-Q* ≤ *n*›
from ‹*t-P* = *t-P′* @ *t-P″*› **have** ‹*t-P′* ≤ *t-P*› **by** *simp*
from *interleave-le-left*[*OF assms*(*5*) *this*]
obtain *t-Q′ t-Q″ u′ u″*
where ∗∗ : ‹*u* = *u′* @ *u″*› ‹*t-Q* = *t-Q′* @ *t-Q″*›
‹*u′ setinterleaves* ((*t-P′*, *t-Q′*), *range tick* ∪ *ev* ‘ *A*)›
by (*meson Prefix-Order.prefixE*)
from *assms*(*1*) ‹*u* = *u′* @ *u″*› **have** ‹*tF u′*› **by** *auto*
moreover from ∗(*2*,*6*) ∗∗(*2*,*3*) **have** ‹*u′* ∈ $\mathcal{T}$ (*P* ⟦*A*⟧ *Q*)›
by (*simp add*: *T-Sync*) (*metis is-processT3-TR-append*)
moreover have ‹*length t-P′* ≤ *length u′*›
using ∗∗(*3*) *interleave-imp-lengthLR-le* **by** *blast*
ultimately have ‹*u′* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
by (*metis* ∗(*3*) *D-restriction-process*$_{ptick}$*I nless-le*)
with ∗∗(*1*) *assms*(*1*, *2*) **show** ‹*u* @ *v* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
by (*metis is-processT7 tF-append-iff tF-imp-ftF*)
next
fix *t-P′ t-P″ t-Q′ t-Q″*
assume $: ‹*t-P* = *t-P′* @ *t-P″*› ‹*t-P′* ∈ $\mathcal{T}$ *P*› ‹*length t-P′* = *n*›
‹*tF t-P′*› ‹*ftF t-P″*› ‹*t-Q* = *t-Q′* @ *t-Q″*› ‹*t-Q′* ∈ $\mathcal{T}$ *Q*›
‹*length t-Q′* = *n*› ‹*tF t-Q′*› ‹*ftF t-Q″*›
from $(*1*, *6*) **have** ‹*t-P′* ≤ *t-P*› ‹*t-Q′* ≤ *t-Q*› **by** *simp-all*
from *interleave-le-prefixLR*[*OF assms*(*5*) *this*]
show ‹*u* @ *v* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
proof (*elim disjE conjE exE*)
fix *u′ t-Q‴* **assume** $$: ‹*u′* ≤ *u*› ‹*t-Q‴* ≤ *t-Q′*›
‹*u′ setinterleaves* ((*t-P′*, *t-Q‴*), *range tick* ∪ *ev* ‘ *A*)›
from $(*7*) $$(*2*) *is-processT3-TR* **have** ‹*t-Q‴* ∈ $\mathcal{T}$ *Q*› **by** *blast*
with $$(*3*) ‹*t-P′* ∈ $\mathcal{T}$ *P*› **have** ‹*u′* ∈ $\mathcal{T}$ (*P* ⟦*A*⟧ *Q*)›
by (*auto simp add*: *T-Sync*)
moreover have ‹*n* ≤ *length u′*›
using $(*3*) $$(*3*) *interleave-imp-lengthLR-le* **by** *blast*
ultimately have ‹*u′* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
by (*metis* $$(*1*) *D-restriction-process*$_{ptick}$*I Prefix-Order.prefixE*
assms(*1*) *nless-le tF-append-iff*)
thus ‹*u* @ *v* ∈ $\mathcal{D}$ (*P* ⟦*A*⟧ *Q* ↓ *n*)›
by (*metis* $$(*1*) *Prefix-Order.prefixE assms*(*1*,*2*) *is-processT7*
tF-append-iff tF-imp-ftF)
next
fix *u′ t-P‴* **assume** $$: ‹*u′* ≤ *u*› ‹*t-P‴* ≤ *t-P′*›
‹*u′ setinterleaves* ((*t-P‴*, *t-Q′*), *range tick* ∪ *ev* ‘ *A*)›
from $(*2*) $$(*2*) *is-processT3-TR* **have** ‹*t-P‴* ∈ $\mathcal{T}$ *P*› **by** *blast*
with $$(*3*) ‹*t-Q′* ∈ $\mathcal{T}$ *Q*› **have** ‹*u′* ∈ $\mathcal{T}$ (*P* ⟦*A*⟧ *Q*)›

by (*auto simp add*: *T-Sync*)
moreover have ‹$n \leq$ *length* u'›
using $(8) $$(3) *interleave-imp-lengthLR-le* **by** *blast*
ultimately have ‹$u' \in \mathcal{D}\ (P \llbracket A \rrbracket\ Q \downarrow n)$›
by (*metis* $$(1) *D-restriction-process*$_{ptick}$*I Prefix-Order.prefixE assms*(1) *nless-le tF-append-iff*)
thus ‹$u \mathbin{@} v \in \mathcal{D}\ (P \llbracket A \rrbracket\ Q \downarrow n)$›
by (*metis* $$(1) *Prefix-Order.prefixE assms*(1,2) *is-processT7 tF-append-iff tF-imp-ftF*)
qed
qed

6.2 Refinement

lemma *restriction-process*$_{ptick}$*-Sync-FD* :
‹$P \llbracket A \rrbracket\ Q \downarrow n \sqsubseteq_{FD} (P \downarrow n) \llbracket A \rrbracket\ (Q \downarrow n)$› (**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
proof (*unfold refine-defs*, *safe*)
show ‹$t \in \mathcal{D}$ *?rhs* $\Longrightarrow t \in \mathcal{D}$ *?lhs*› **for** t
by (*unfold D-Sync-optimized*, *safe*)
(*solves* ‹*simp add*: *restriction-process*$_{ptick}$*-Sync-FD-div-oneside*›,
metis Sync-commute restriction-process$_{ptick}$*-Sync-FD-div-oneside*)
thus ‹$(t, X) \in \mathcal{F}\ ((P \downarrow n) \llbracket A \rrbracket\ (Q \downarrow n)) \Longrightarrow (t, X) \in \mathcal{F}\ (P \llbracket A \rrbracket\ Q \downarrow n)$› **for** $t\ X$
by (*meson is-processT8 le-approxD*(2) *mono-Sync restriction-process*$_{ptick}$*-approx-self*)
qed

The equality does not hold in general, but we can establish it by adding an assumption over the strict alphabets of the processes.

lemma *strict-events-of-subset-restriction-process*$_{ptick}$*-Sync* :
‹$P \llbracket A \rrbracket\ Q \downarrow n = (P \downarrow n) \llbracket A \rrbracket\ (Q \downarrow n)$› (**is** ‹*?lhs* = *?rhs*›)
if ‹$\boldsymbol{\alpha}(P) \subseteq A \vee \boldsymbol{\alpha}(Q) \subseteq A$›
proof (*rule FD-antisym*)
show ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*› **by** (*fact restriction-process*$_{ptick}$*-Sync-FD*)
next
have *div* : ‹$t \in \mathcal{D}\ (P \llbracket A \rrbracket\ Q) \Longrightarrow t \in \mathcal{D}$ *?rhs*› **for** t
by (*auto simp add*: *D-Sync restriction-process*$_{ptick}$*-projs*)

{ fix $t\ u\ v$ **assume** ‹$t = u \mathbin{@} v$› ‹$u \in \mathcal{T}\ (P \llbracket A \rrbracket\ Q)$› ‹*length* $u = n$› ‹*tF* u› ‹*ftF* v›
from *this*(2) **consider** ‹$u \in \mathcal{D}\ (P \llbracket A \rrbracket\ Q)$›
| *t-P t-Q* **where** ‹*t-P* $\in \mathcal{T}\ P$› ‹*t-Q* $\in \mathcal{T}\ Q$›
‹u *setinterleaves* ((*t-P*, *t-Q*), *range tick* $\cup$ *ev* ‘ A)›
unfolding *Sync-projs* **by** *blast*
hence ‹$t \in \mathcal{D}$ *?rhs*›
proof *cases*
show ‹$u \in \mathcal{D}\ (P \llbracket A \rrbracket\ Q) \Longrightarrow t \in \mathcal{D}$ *?rhs*›
by (*simp add*: ‹*ftF* v› ‹$t = u \mathbin{@} v$› ‹*tF* u› *div is-processT7*)
next
fix *t-P t-Q* **assume** ‹*t-P* $\in \mathcal{T}\ P$› ‹*t-Q* $\in \mathcal{T}\ Q$›

```
      and setinter : ‹u setinterleaves ((t-P, t-Q), range tick ∪ ev ‘ A)›
    consider ‹t-P ∈ 𝒟 P ∨ t-Q ∈ 𝒟 Q› | ‹t-P ∉ 𝒟 P› ‹t-Q ∉ 𝒟 Q›
by blast
    thus ‹t ∈ 𝒟 ?rhs›
    proof cases
      assume ‹t-P ∈ 𝒟 P ∨ t-Q ∈ 𝒟 Q›
       with ‹t-P ∈ 𝒯 P› ‹t-Q ∈ 𝒯 Q› setinter ‹ftF v› ‹t = u @ v›
‹tF u›
      have ‹t ∈ 𝒟 (P ⟦A⟧ Q)›
        using setinterleaving-dual by (simp add: D-Sync) blast
      thus ‹t ∈ 𝒟 ?rhs› by (fact div)
    next
      assume ‹t-P ∉ 𝒟 P› ‹t-Q ∉ 𝒟 Q›
      with ‹t-P ∈ 𝒯 P› ‹t-Q ∈ 𝒯 Q› ‹α(P) ⊆ A ∨ α(Q) ⊆ A›
      have ‹{a. ev a ∈ set t-P} ⊆ A ∨ {a. ev a ∈ set t-Q} ⊆ A›
        by (auto intro: strict-events-of-memI)
      with interleave-subsetL[OF ‹tF u› - setinter]
        interleave-subsetR[OF ‹tF u› - setinter]
      have ‹u = t-P ∨ u = t-Q› by blast
       with ‹length u = n› have ‹length t-P = n ∨ length t-Q = n›
by auto
      moreover from ‹tF u› tF-interleave-iff[OF setinter]
      have ‹tF t-P› ‹tF t-Q› by simp-all
      ultimately have ‹t-P ∈ 𝒟 (P ↓ n) ∨ t-Q ∈ 𝒟 (Q ↓ n)›
     using ‹t-P ∈ 𝒯 P› ‹t-Q ∈ 𝒯 Q› by (metis D-restriction-process_ptick I)
      moreover from ‹t-P ∈ 𝒯 P› ‹t-Q ∈ 𝒯 Q›
      have ‹t-P ∈ 𝒯 (P ↓ n)› ‹t-Q ∈ 𝒯 (Q ↓ n)›
        by (simp-all add: T-restriction-process_ptick I)
      ultimately show ‹t ∈ 𝒟 ?rhs›
        using ‹ftF v› ‹t = u @ v› ‹tF u› setinter
        by (simp add: D-Sync-optimized)
          (metis setinterleaving-dual)
    qed
  qed
 } note ∗ = this

 show ‹?rhs ⊑_FD ?lhs›
 proof (unfold refine-defs, safe)
   show ‹t ∈ 𝒟 ?lhs ⟹ t ∈ 𝒟 ?rhs› for t
   proof (elim D-restriction-process_ptick E)
     show ‹t ∈ 𝒟 (P ⟦A⟧ Q) ⟹ t ∈ 𝒟 ?rhs› by (fact div)
   next
     show ‹⟦t = u @ v; u ∈ 𝒯 (P ⟦A⟧ Q); length u = n; tF u; ftF v⟧
           ⟹ t ∈ 𝒟 ?rhs› for u v by (fact ∗)
   qed
 next
   show ‹(t, X) ∈ ℱ ?lhs ⟹ (t, X) ∈ ℱ ?rhs› for t X
   proof (elim F-restriction-process_ptick E)
```

assume ‹$(t,\ X) \in \mathcal{F}\ (P\ [\![A]\!]\ Q)$›
then consider ‹$t \in \mathcal{D}\ (P\ [\![A]\!]\ Q)$›
| (*fail*) *t-P t-Q X-P X-Q* **where** ‹$(t\text{-}P,\ X\text{-}P) \in \mathcal{F}\ P$› ‹$(t\text{-}Q,\ X\text{-}Q) \in \mathcal{F}\ Q$›
‹*t setinterleaves* $((t\text{-}P,\ t\text{-}Q),\ range\ tick \cup ev\ `\ A)$›
‹$X = (X\text{-}P \cup X\text{-}Q) \cap (range\ tick \cup ev\ `\ A) \cup X\text{-}P \cap X\text{-}Q$›
unfolding *Sync-projs* **by** *blast*
thus ‹$(t,\ X) \in \mathcal{F}$ *?rhs*›
proof *cases*
from *div D-F* **show** ‹$t \in \mathcal{D}\ (P\ [\![A]\!]\ Q) \Longrightarrow (t,\ X) \in \mathcal{F}$ *?rhs*› **by** *blast*
next
case *fail*
thus ‹$(t,\ X) \in \mathcal{F}$ *?rhs*›
by (*auto simp add*: *F-Sync F-restriction-process*$_{ptick}$)
qed
next
show ‹$[\![t = u\ @\ v;\ u \in \mathcal{T}\ (P\ [\![A]\!]\ Q);\ length\ u = n;\ tF\ u;\ ftF\ v]\!] \Longrightarrow (t,\ X) \in \mathcal{F}$ *?rhs*› **for** *u v* **by** (*simp add*: $*$ *is-processT8*)
qed
qed
qed

corollary *restriction-process*$_{ptick}$*-MultiSync-FD* :
‹$[\![A]\!]\ m \in\#\ M.\ P\ l \downarrow n \sqsubseteq_{FD} [\![A]\!]\ m \in\#\ M.\ (P\ l \downarrow n)$›
proof (*induct M rule*: *induct-subset-mset-empty-single*)
case *1* **show** *?case* **by** *simp*
next
case (*2 m*) **show** *?case* **by** *simp*
next
case (*3 N m*)
show *?case*
by (*simp add*: ‹$N \neq \{\#\}$›)
(*fact trans-FD*[*OF restriction-process*$_{ptick}$*-Sync-FD mono-Sync-FD*[*OF FD-refl 3.hyps*(*4*)]])
qed

In the following corollary, we could be more precise by having the condition on at least *size M − 1* processes.

corollary *strict-events-of-subset-restriction-process*$_{ptick}$*-MultiSync* :
‹$[\![A]\!]\ m \in\#\ M.\ P\ m \downarrow n = (if\ n = 0\ then\ \bot\ else\ [\![A]\!]\ m \in\#\ M.\ (P\ m \downarrow n))$›
— *if n = 0 then* ⊥ *else* - is necessary because we can have $M = \{\#\}$.
if ‹$\bigwedge m.\ m \in\#\ M \Longrightarrow \boldsymbol{\alpha}(P\ m) \subseteq A$›
proof (*split if-split*, *intro conjI impI*)
show ‹$n = 0 \Longrightarrow [\![A]\!]\ m \in\#\ M.\ P\ m \downarrow n = \bot$› **by** *simp*
next
show ‹$[\![A]\!]\ m \in\#\ M.\ P\ m \downarrow n = [\![A]\!]\ m \in\#\ M.\ (P\ m \downarrow n)$› **if** ‹$n$

$\neq 0\rangle$
proof (*induct M rule*: *induct-subset-mset-empty-single*)
case *1* **from** ‹$n \neq 0$› **show** *?case* **by** *simp*
next
case (*2 m*) **show** *?case* **by** *simp*
next
case (*3 N m*)
have ‹($[\![A]\!]$ $n \in \# add\text{-}mset\ m\ N.\ P\ n) \downarrow n = (P\ m\ [\![A]\!]\ [\![A]\!]\ n \in \# N.$
$P\ n) \downarrow n\rangle$
by (*simp add*: ‹$N \neq \{\#\}$›)
also have ‹$\ldots = (P\ m \downarrow n)\ [\![A]\!]\ ([\![A]\!]\ n \in \# N.\ P\ n \downarrow n)\rangle$
by (*rule strict-events-of-subset-restriction-process$_{ptick}$-Sync*)
(*simp add*: *3.hyps*(*1*) ‹$\bigwedge m.\ m \in \#\ M \Longrightarrow \boldsymbol{\alpha}(P\ m) \subseteq A\rangle$)
also have ‹($[\![A]\!]$ $m \in \# N.\ P\ m) \downarrow n = [\![A]\!]$ $m \in \# N.\ (P\ m \downarrow n)\rangle$ **by**
(*fact 3.hyps*(*4*))
finally show *?case* **by** (*simp add*: ‹$N \neq \{\#\}$›)
qed
qed

corollary *restriction-process$_{ptick}$-Par* :
‹$P \parallel Q \downarrow n = (P \downarrow n) \parallel (Q \downarrow n)\rangle$
by (*simp add*: *strict-events-of-subset-restriction-process$_{ptick}$-Sync*)

corollary *restriction-process$_{ptick}$-MultiPar* :
‹$\|\| m \in \# M.\ P\ l \downarrow n = (if\ n = 0\ then\ \bot\ else\ \|\| m \in \# M.\ (P\ l \downarrow$
$n))\rangle$
by (*simp add*: *strict-events-of-subset-restriction-process$_{ptick}$-MultiSync*)

6.3 Non Destructiveness

lemma *Sync-non-destructive* :
‹*non-destructive* ($\lambda(P :: ('a, 'r)\ process_{ptick},\ Q).\ P\ [\![A]\!]\ Q)\rangle$
proof (*rule order-non-destructiveI*, *clarify*)
fix $P\ P'\ Q\ Q' ::$ ‹$('a, 'r)\ process_{ptick}\rangle$ **and** n
assume ‹$(P,\ Q) \downarrow n = (P',\ Q') \downarrow n\rangle$ ‹$0 < n\rangle$
hence ‹$P \downarrow n = P' \downarrow n\rangle$ ‹$Q \downarrow n = Q' \downarrow n\rangle$
by (*simp-all add*: *restriction-prod-def*)
show ‹$P\ [\![A]\!]\ Q \downarrow n \sqsubseteq_{FD} P'\ [\![A]\!]\ Q' \downarrow n\rangle$
proof (*rule leFD-restriction-process$_{ptick}$I*)
show ‹$t \in \mathcal{D}\ (P'\ [\![A]\!]\ Q') \Longrightarrow t \in \mathcal{D}\ (P\ [\![A]\!]\ Q \downarrow n)\rangle$ **for** t
by (*metis* (*mono-tags*, *opaque-lifting*) ‹$P \downarrow n = P' \downarrow n\rangle$ ‹$Q \downarrow n$
$= Q' \downarrow n\rangle$
in-mono le-FD-D(*1*) *mono-Sync-FD restriction-process$_{ptick}$-FD-self*
restriction-process$_{ptick}$-Sync-FD)
next
show ‹$(s,\ X) \in \mathcal{F}\ (P'\ [\![A]\!]\ Q') \Longrightarrow (s,\ X) \in \mathcal{F}\ (P\ [\![A]\!]\ Q \downarrow n)\rangle$ **for**
$s\ X$

by (*metis* (*mono-tags*, *opaque-lifting*) ‹$P \downarrow n = P' \downarrow n$› ‹$Q \downarrow n = Q' \downarrow n$›
failure-refine-def leFD-imp-leF mono-Sync-FD
restriction-process$_{ptick}$-FD-self
restriction-process$_{ptick}$-Sync-FD subset-iff)
qed
qed

7 Non Destructiveness of Throw

7.1 Equality

lemma *Depth-Throw-1-is-constant*: ‹$P \ominus a \in A.\ Q1\ a \downarrow 1 = P \ominus a \in A.\ Q2\ a \downarrow 1$›
proof (*rule FD-antisym*)
show ‹$P \ominus a \in A.\ Q2\ a \downarrow 1 \sqsubseteq_{FD} P \ominus a \in A.\ Q1\ a \downarrow 1$› **for** *Q1 Q2*
proof (*unfold refine-defs*, *safe*)
show *div* : ‹$t \in \mathcal{D}\ (Throw\ P\ A\ Q1 \downarrow 1) \Longrightarrow t \in \mathcal{D}\ (Throw\ P\ A\ Q2 \downarrow 1)$› **for** *t*
proof (*elim D-restriction-process$_{ptick}$E*)
assume ‹$t \in \mathcal{D}\ (P \ominus a \in A.\ Q1\ a)$› **and** ‹*length t* ≤ 1›
from ‹*length t* ≤ 1› **consider** ‹$t = []$› | *e* **where** ‹$t = [e]$› **by** (*cases t*) *simp-all*
thus ‹$t \in \mathcal{D}\ (P \ominus a \in A.\ Q2\ a \downarrow 1)$›
proof *cases*
from ‹$t \in \mathcal{D}\ (P \ominus a \in A.\ Q1\ a)$› **show** ‹$t = [] \Longrightarrow t \in \mathcal{D}\ (P \ominus a \in A.\ Q2\ a \downarrow 1)$›
by (*simp add*: *D-restriction-process$_{ptick}$ D-Throw*)
next
fix *e* **assume** ‹$t = [e]$›
with ‹$t \in \mathcal{D}\ (P \ominus a \in A.\ Q1\ a)$›
consider ‹$[] \in \mathcal{D}\ P$› | *a* **where** ‹$t = [ev\ a]$› ‹$[ev\ a] \in \mathcal{D}\ P$› ‹$a \notin A$›
| *a* **where** ‹$t = [ev\ a]$› ‹$[ev\ a] \in \mathcal{T}\ P$› ‹$a \in A$›
by (*auto simp add*: *D-Throw disjoint-iff image-iff*)
(*metis D-T append-Nil event$_{ptick}$.exhaust process-charn*,
metis append-Nil empty-iff empty-set hd-append2 hd-in-set in-set-conv-decomp set-ConsD)
thus ‹$t \in \mathcal{D}\ (P \ominus a \in A.\ Q2\ a \downarrow 1)$›
proof *cases*
show ‹$t = [ev\ a] \Longrightarrow [ev\ a] \in \mathcal{T}\ P \Longrightarrow a \in A \Longrightarrow t \in \mathcal{D}\ (P \ominus a \in A.\ Q2\ a \downarrow 1)$› **for** *a*
by (*simp add*: *D-restriction-process$_{ptick}$ T-Throw*)
(*metis append-Nil append-self-conv ftF-charn inf-bot-left
is-ev-def is-processT1-TR length-0-conv length-Cons*

list.set(*1*)
tF-Cons-iff tF-Nil)
next
show ‹[] ∈ 𝒟 *P* ⟹ *t* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q2 a* ↓ *1*)›
by (*simp flip*: *BOT-iff-Nil-D add*: *D-BOT*)
(*use* ‹*t* = [*e*]› *ftF-single* **in** *blast*)
next
show ‹*t* = [*ev a*] ⟹ [*ev a*] ∈ 𝒟 *P* ⟹ *a* ∉ *A* ⟹ *t* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q2 a* ↓ *1*)› **for** *a*
by (*simp add*: *D-restriction-process*$_{ptick}$ *D-Throw disjoint-iff image-iff*)
(*metis append.right-neutral empty-set event*$_{ptick}$*.disc*(*1*) *event*$_{ptick}$*.sel*(*1*)
ftF-Nil list.simps(*15*) *singletonD tF-Cons-iff tF-Nil*)
qed
qed
next
fix *u v* **assume** ∗ : ‹*t* = *u* @ *v*› ‹*u* ∈ 𝒯 (*Throw P A Q1*)› ‹*length u* = *1*› ‹*tF u*› ‹*ftF v*›
from ‹*length u* = *1*› ‹*tF u*› **obtain** *a* **where** ‹*u* = [*ev a*]›
by (*cases u*) (*auto simp add*: *is-ev-def*)
with ∗(*2*) **show** ‹*t* ∈ 𝒟 (*Throw P A Q2* ↓ *1*)›
by (*simp add*: ‹*t* = *u* @ *v*› *D-restriction-process*$_{ptick}$ *Throw-projs Cons-eq-append-conv*)
(*metis* (*no-types*) ∗(*3*−*5*) *One-nat-def append-Nil empty-set inf-bot-left*
insert-disjoint(*2*) *is-processT1-TR list.simps*(*15*))
qed

show ‹(*t*, *X*) ∈ ℱ (*Throw P A Q1* ↓ *1*) ⟹ (*t*, *X*) ∈ ℱ (*Throw P A Q2* ↓ *1*)› **for** *t X*
proof (*elim F-restriction-process*$_{ptick}$*E*)
assume ‹(*t*, *X*) ∈ ℱ (*P* Θ *a* ∈ *A*. *Q1 a*)› ‹*length t* ≤ *1*›
then consider ‹*t* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q1 a*)› | ‹(*t*, *X*) ∈ ℱ *P*› ‹*set t* ∩ *ev* ‘ *A* = {}›
| *a* **where** ‹*t* = [*ev a*]› ‹[*ev a*] ∈ 𝒯 *P*› ‹*a* ∈ *A*›
by (*auto simp add*: *F-Throw D-Throw*)
thus ‹(*t*, *X*) ∈ ℱ (*P* Θ *a* ∈ *A*. *Q2 a* ↓ *1*)›
proof *cases*
from *D-F div* ‹*length t* ≤ *1*›
show ‹*t* ∈ 𝒟 (*P* Θ *a* ∈ *A*. *Q1 a*) ⟹ (*t*, *X*) ∈ ℱ (*P* Θ *a* ∈ *A*. *Q2 a* ↓ *1*)›
using *D-restriction-process*$_{ptick}$*I* **by** *blast*
next
show ‹(*t*, *X*) ∈ ℱ *P* ⟹ *set t* ∩ *ev* ‘ *A* = {} ⟹ (*t*, *X*) ∈ ℱ (*Throw P A Q2* ↓ *1*)›
by (*simp add*: *F-restriction-process*$_{ptick}$ *F-Throw*)
next
show ‹⟦*t* = [*ev a*]; [*ev a*] ∈ 𝒯 *P*; *a* ∈ *A*⟧ ⟹ (*t*, *X*) ∈ ℱ (*Throw*

P A Q2 $\downarrow$ *1)*› **for** *a*
by (*simp add*: *F-restriction-process*$_{ptick}$ *T-Throw*)
(*metis append.right-neutral append-Nil empty-set event*$_{ptick}$*.disc*(*1*)
ftF-Nil inf-bot-left is-processT1-TR length-Cons
list.size(*3*) *tF-Cons-iff tF-Nil*)
qed
next
fix *u v* **assume** ∗ : ‹*t* = *u* @ *v*› ‹*u* ∈ $\mathcal{T}$ (*Throw P A Q1*)› ‹*length u* = *1*› ‹*tF u*› ‹*ftF v*›
from ‹*length u* = *1*› ‹*tF u*› **obtain** *a* **where** ‹*u* = [*ev a*]›
by (*cases u*) (*auto simp add*: *is-ev-def*)
with ∗(*2*) **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ (*Throw P A Q2* $\downarrow$ *1*)›
by (*simp add*: ‹*t* = *u* @ *v*› *F-restriction-process*$_{ptick}$ *Throw-projs Cons-eq-append-conv*)
(*metis* (*no-types*) ∗(*3*−*5*) *One-nat-def append-Nil empty-set inf-bot-left*
insert-disjoint(*2*) *is-processT1-TR list.simps*(*15*))
qed
qed

thus ‹*P* Θ *a* ∈ *A*. *Q2 a* $\downarrow$ *1* $\sqsubseteq_{FD}$ *P* Θ *a* ∈ *A*. *Q1 a* $\downarrow$ *1*› **by** *simp*
qed

7.2 Refinement

lemma *restriction-process*$_{ptick}$*-Throw-FD* :
‹(*P* Θ *a* ∈ *A*. *Q a*) $\downarrow$ *n* $\sqsubseteq_{FD}$ (*P* $\downarrow$ *n*) Θ *a* ∈ *A*. (*Q a* $\downarrow$ *n*)› (**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
proof (*unfold refine-defs*, *safe*)
show ‹*t* ∈ $\mathcal{D}$ *?lhs*› **if** ‹*t* ∈ $\mathcal{D}$ *?rhs*› **for** *t*
proof −
from ‹*t* ∈ $\mathcal{D}$ *?rhs*›
consider *t1 t2* **where** ‹*t* = *t1* @ *t2*› ‹*t1* ∈ $\mathcal{D}$ (*P* $\downarrow$ *n*)› ‹*tF t1*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*ftF t2*›
| *t1 a t2* **where** ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ (*P* $\downarrow$ *n*)›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ $\mathcal{D}$ (*Q a* $\downarrow$ *n*)›
unfolding *D-Throw* **by** *blast*
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*
show ‹⟦*t* = *t1* @ *t2*; *t1* ∈ $\mathcal{D}$ (*P* $\downarrow$ *n*); *tF t1*; *set t1* ∩ *ev* ‘ *A* = {}; *ftF t2*⟧ ⟹ *t* ∈ $\mathcal{D}$ *?lhs*› **for** *t1 t2*
by (*elim D-restriction-process*$_{ptick}$*E*, *simp-all add*: *Throw-projs D-restriction-process*$_{ptick}$)
(*blast*, *metis* (*no-types*, *lifting*) *ftF-append inf-sup-aci*(*8*) *inf-sup-distrib2 sup-bot-right*)
next
fix *t1 a t2*
assume ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ (*P* $\downarrow$ *n*)›
‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*a* ∈ *A*› ‹*t2* ∈ $\mathcal{D}$ (*Q a* $\downarrow$ *n*)›

from ‹*t2* ∈ $\mathcal{D}$ (*Q a* ↓ *n*)› **show** ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof (*elim D-restriction-process*$_{ptick}$*E*)
from ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ (*P* ↓ *n*)› **show** ‹*t2* ∈ $\mathcal{D}$ (*Q a*) ⟹ *length t2* ≤ *n* ⟹ *t* ∈ $\mathcal{D}$ *?lhs*›
proof (*elim T-restriction-process*$_{ptick}$*E*)
from ‹*a* ∈ *A*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*t* = *t1* @ *ev a* # *t2*›
show ‹*t2* ∈ $\mathcal{D}$ (*Q a*) ⟹ *t1* @ [*ev a*] ∈ $\mathcal{T}$ *P* ⟹ *t* ∈ $\mathcal{D}$ *?lhs*›
by (*auto simp add*: *D-restriction-process*$_{ptick}$ *D-Throw*)
next
fix *u v* **assume** ‹*t2* ∈ $\mathcal{D}$ (*Q a*)› ‹*length t2* ≤ *n*› ‹*t1* @ [*ev a*] = *u* @ *v*›
‹*u* ∈ $\mathcal{T}$ *P*› ‹*length u* = *n*› ‹*tF u*› ‹*ftF v*›
from ‹*t1* @ [*ev a*] = *u* @ *v*› ‹*ftF v*› **consider** ‹*t1* @ [*ev a*] = *u*›
| *v*′ **where** ‹*t1* = *u* @ *v*′› ‹*v* = *v*′ @ [*ev a*]› ‹*ftF v*′›
by (*cases v rule*: *rev-cases*) (*simp-all add*: *ftF-append-iff*)
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*
from ‹*a* ∈ *A*› ‹*length u* = *n*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*t2* ∈ $\mathcal{D}$ (*Q a*)› ‹*tF u*› ‹*u* ∈ $\mathcal{T}$ *P*›
show ‹*t1* @ [*ev a*] = *u* ⟹ *t* ∈ $\mathcal{D}$ *?lhs*›
by (*simp add*: *D-restriction-process*$_{ptick}$ *T-Throw* ‹*t* = *t1* @ *ev a* # *t2*›)
(*metis Cons-eq-appendI D-imp-ftF append-Nil append-assoc is-processT1-TR*)
next
from ‹*ftF v*› ‹*length u* = *n*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*t* = *t1* @ *ev a* # *t2*› ‹*u* ∈ $\mathcal{T}$ *P*›
show ‹*t1* = *u* @ *v*′ ⟹ *v* = *v*′ @ [*ev a*] ⟹ *ftF v*′ ⟹ *t* ∈ $\mathcal{D}$ *?lhs*› **for** *v*′
by (*simp add*: *D-restriction-process*$_{ptick}$ *T-Throw*)
(*metis D-imp-ftF Int-assoc Un-Int-eq*(*3*) *append-assoc ftF-append ftF-nonempty-append-imp inf-bot-right list.distinct*(*1*) *same-append-eq set-append* ‹*t* ∈ $\mathcal{D}$ *?rhs*›)
qed
qed
next
from ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ (*P* ↓ *n*)›
show ‹⟦*t2* = *u* @ *v*; *u* ∈ $\mathcal{T}$ (*Q a*); *length u* = *n*; *tF u*; *ftF v*⟧ ⟹ *t* ∈ $\mathcal{D}$ *?lhs*› **for** *u v*
proof (*elim T-restriction-process*$_{ptick}$*E*)
assume ‹*t2* = *u* @ *v*› ‹*u* ∈ $\mathcal{T}$ (*Q a*)› ‹*length u* = *n*› ‹*tF u*› ‹*ftF v*› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P*› ‹*length* (*t1* @ [*ev a*]) ≤ *n*›
from ‹*a* ∈ *A*› ‹*set t1* ∩ *ev* ‘ *A* = {}› ‹*t1* @ [*ev a*] ∈ $\mathcal{T}$ *P*› ‹*u* ∈ $\mathcal{T}$ (*Q a*)›
have ‹*t1* @ *ev a* # *u* ∈ $\mathcal{T}$ (*P* Θ *a*∈*A*. *Q a*)› **by** (*auto simp add*: *T-Throw*)
moreover have ‹*n* < *length* (*t1* @ *ev a* # *u*)› **by** (*simp add*:

‹*length u = n*›)
ultimately have ‹*t1 @ ev a # u ∈ 𝒟 ?lhs*› **by** (*simp add*: *D-restriction-process*$_{ptick}$*I*)
moreover have ‹*t = (t1 @ ev a # u) @ v*› **by** (*simp add*: ‹*t = t1 @ ev a # t2*› ‹*t2 = u @ v*›)
moreover from ‹*t1 @ [ev a] ∈ 𝒯 P*› ‹*tF u*› *append-T-imp-tF*
have ‹*tF (t1 @ ev a # u)*› **by** *auto*
ultimately show ‹*t ∈ 𝒟 ?lhs*› **using** ‹*ftF v*› *is-processT7* **by** *blast*
next
fix *w x* **assume** ‹*t2 = u @ v*› ‹*u ∈ 𝒯 (Q a)*› ‹*length u = n*› ‹*tF u*› ‹*ftF v*›
‹*t1 @ [ev a] = w @ x*› ‹*w ∈ 𝒯 P*› ‹*length w = n*› ‹*tF w*› ‹*ftF x*›
from ‹*t1 @ [ev a] = w @ x*› **consider** ‹*t1 @ [ev a] = w*›
| *x'* **where** ‹*t1 = w @ x'*› ‹*x = x' @ [ev a]*›
by (*cases x rule*: *rev-cases*) *simp-all*
thus ‹*t ∈ 𝒟 ?lhs*›
proof *cases*
assume ‹*t1 @ [ev a] = w*›
with ‹*a ∈ A*› ‹*set t1 ∩ ev ‘ A = {}*› ‹*u ∈ 𝒯 (Q a)*› ‹*w ∈ 𝒯 P*›
have ‹*t1 @ ev a # u ∈ 𝒯 (P Θ a∈A. Q a)*› **by** (*auto simp add*: *T-Throw*)
moreover have ‹*n < length (t1 @ ev a # u)*› **by** (*simp add*: ‹*length u = n*›)
ultimately have ‹*t1 @ ev a # u ∈ 𝒟 ?lhs*› **by** (*blast intro*: *D-restriction-process*$_{ptick}$*I*)
moreover have ‹*t = (t1 @ ev a # u) @ v*›
by (*simp add*: ‹*t = t1 @ ev a # t2*› ‹*t2 = u @ v*›)
moreover from ‹*t1 @ [ev a] = w*› ‹*tF u*› ‹*tF w*› **have** ‹*tF (t1 @ ev a # u)*› **by** *auto*
ultimately show ‹*t ∈ 𝒟 ?lhs*› **using** ‹*ftF v*› *is-processT7* **by** *blast*
next
fix *x'* **assume** ‹*t1 = w @ x'*› ‹*x = x' @ [ev a]*›
from ‹*set t1 ∩ ev ‘ A = {}*› ‹*t1 = w @ x'*› ‹*w ∈ 𝒯 P*›
have ‹*w ∈ 𝒯 P ∧ set w ∩ ev ‘ A = {}*› **by** *auto*
hence ‹*w ∈ 𝒯 (P Θ a∈A. Q a)*› **by** (*simp add*: *T-Throw*)
with ‹*length w = n*› ‹*tF w*› **have** ‹*w ∈ 𝒟 ?lhs*›
by (*blast intro*: *D-restriction-process*$_{ptick}$*I*)
moreover have ‹*t = w @ x @ t2*›
by (*simp add*: ‹*t = t1 @ ev a # t2*› ‹*t1 @ [ev a] = w @ x*›)
moreover from *D-imp-ftF*[*OF* ‹*t ∈ 𝒟 ?rhs*›] ‹*t = t1 @ ev a # t2*› ‹*t1 = w @ x'*›
ftF-nonempty-append-imp that tF-append-iff
have ‹*ftF (x @ t2)*› **by** (*simp add*: ‹*t = w @ x @ t2*› *ftF-append-iff*)

ultimately show ‹$t \in \mathcal{D}$ *?lhs*› **by** (*simp add*: ‹*tF w*› *is-processT7*)
qed
qed
qed
qed
qed

thus ‹$(t, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow (t, X) \in \mathcal{F}$ *?lhs*› **for** *t X*
by (*meson is-processT8 le-approxD*(*2*) *mono-Throw restriction-process*$_{ptick}$*-approx-self*)
qed

7.3 Non Destructiveness

lemma *Throw-non-destructive* :
‹*non-destructive* (λ(P :: ('*a*, '*r*) *process*$_{ptick}$, Q). $P \Theta a \in A.\ Q\ a$)›
proof (*rule order-non-destructiveI*, *clarify*)
fix $P\ P'$ :: ‹('*a*, '*r*) *process*$_{ptick}$› **and** $Q\ Q'$ :: ‹'*a* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$› **and** *n*
assume ‹$(P, Q) \downarrow n = (P', Q') \downarrow n$› ‹$0 < n$›
hence ‹$P \downarrow n = P' \downarrow n$› ‹$Q \downarrow n = Q' \downarrow n$›
by (*simp-all add*: *restriction-prod-def*)
show ‹$P \Theta a \in A.\ Q\ a \downarrow n \sqsubseteq_{FD} P' \Theta a \in A.\ Q'\ a \downarrow n$›
proof (*rule leFD-restriction-process*$_{ptick}$*I*)
show ‹$t \in \mathcal{D}\ (P' \Theta a \in A.\ Q'\ a) \Longrightarrow t \in \mathcal{D}\ (P \Theta a \in A.\ Q\ a \downarrow n)$› **for** *t*
by (*metis* (*no-types*, *lifting*) *ext* ‹$P \downarrow n = P' \downarrow n$› ‹$Q \downarrow n = Q' \downarrow n$›[*unfolded restriction-fun-def*]
in-mono le-FD-D(*1*) *mono-Throw-FD restriction-process*$_{ptick}$*-FD-self*
restriction-process$_{ptick}$*-Throw-FD*)
next
show ‹$(s, X) \in \mathcal{F}\ (P' \Theta a \in A.\ Q'\ a) \Longrightarrow (s, X) \in \mathcal{F}\ (P \Theta a \in A.\ Q\ a \downarrow n)$› **for** *s X*
by (*metis* (*no-types*, *lifting*) *ext* ‹$P \downarrow n = P' \downarrow n$›
‹$Q \downarrow n = Q' \downarrow n$›[*unfolded restriction-fun-def*]
in-mono le-FD-D(*2*) *mono-Throw-FD restriction-process*$_{ptick}$*-FD-self*
restriction-process$_{ptick}$*-Throw-FD*)
qed
qed

Stronger version in Isabelle26, initially suggested to Burkhart by Claude and polished by Benoît.

lemma *ThrowR-constructive* :
‹*constructive* (λQ :: '*a* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$. $P \Theta a \in A.\ Q\ a$)›
proof (*rule order-constructiveI*)
fix $Q\ Q'$ :: ‹'*a* $\Rightarrow$ ('*a*, '*r*) *process*$_{ptick}$› **and** *n* **assume** ‹$Q \downarrow n = Q' \downarrow n$›
show ‹$(P \Theta a \in A.\ Q\ a) \downarrow Suc\ n \sqsubseteq_{FD} P \Theta a \in A.\ Q'\ a \downarrow Suc\ n$›
(**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)

proof (*rule failure-divergence-refineI-*D_{min}*-version*)
fix *t* **assume** ‹$t \in \mathcal{D}_{min}$ *?rhs*›
hence ‹$t \in \mathcal{D}_{min}$ $(P \Theta a \in A.\ Q'\ a) \wedge$ *length t* $\leq$ *Suc n* $\vee$
$t \in \mathcal{T}$ $(P \Theta a \in A.\ Q'\ a) - \mathcal{D}$ $(P \Theta a \in A.\ Q'\ a) \wedge$ *length t* = *Suc n* $\wedge$ *tF t*›
by (*simp add*: D_{min}*-restriction-process*$_{ptick}$)
thus ‹$t \in \mathcal{D}$ *?lhs*›
proof (*elim disjE conjE*)
assume ‹$t \in \mathcal{D}_{min}$ $(P \Theta a \in A.\ Q'\ a)$› ‹*length t* $\leq$ *Suc n*›
from *this*(*1*) **consider** (*D-P*) ‹$t \in \mathcal{D}_{min}$ *P*› ‹*tF t*› ‹*set t* $\cap$ *ev* ‘ *A* = {}›
| (*D-Q*) *t1 a t2* **where** ‹*t* = *t1* @ *ev a* # *t2*› ‹*t1* @ [*ev a*] $\in \mathcal{T}$ *P*›
‹*set t1* $\cap$ *ev* ‘ *A* = {}› ‹*a* $\in$ *A*› ‹*t2* $\in \mathcal{D}_{min}$ $(Q'\ a)$›
by (*blast dest*: D_{min}*-Throw-subset*[*THEN set-mp*])
thus ‹$t \in \mathcal{D}$ *?lhs*›
proof *cases*
case *D-P* **thus** ‹$t \in \mathcal{D}$ *?lhs*›
by (*intro D-restriction-process*$_{ptick}$*I*, *simp add*: *D-Throw*)
(*metis* (*mono-tags*, *lifting*) D_{min}*-D append.right-neutral ftF-Nil*)
next
case *D-Q*
from *D-Q*(*1*) ‹*length t* $\leq$ *Suc n*› **have** ‹*length t2* $\leq$ *n*› **by** *simp*
hence ‹*length t2* < *n* $\vee$ *length t2* = *n*› **by** *presburger*
thus ‹$t \in \mathcal{D}$ *?lhs*›
proof (*elim disjE*)
assume ‹*length t2* < *n*›
with *D-Q*(*5*) ‹$Q \downarrow n = Q' \downarrow n$› **have** ‹*t2* $\in \mathcal{D}$ $(Q\ a)$›
by (*metis D-restriction-process*$_{ptick}$*I Divergences*$_{min}$*-def elem-min-elems*
length-less-in-D-restriction-process$_{ptick}$ *restriction-fun-def*)
with *D-Q*(*1*−*4*) **show** ‹$t \in \mathcal{D}$ *?lhs*›
by (*auto intro*: *D-restriction-process*$_{ptick}$*I simp add*: *D-Throw*)
next
from ‹$t \in \mathcal{D}_{min}$ $(P \Theta a \in A.\ Q'\ a)$› **have** ‹*tF t*› **by** (*fact tF-mem-*D_{min})
assume ‹*length t2* = *n*›
with *D-Q*(*5*) ‹$Q \downarrow n = Q' \downarrow n$› **have** ‹*t2* $\in \mathcal{T}$ $(Q\ a)$›
by (*metis* D_{min}*-D D-T D-restriction-process*$_{ptick}$*I Orderings.order-eq-iff*
length-le-in-T-restriction-process$_{ptick}$ *restriction-fun-def*)
with *D-Q*(*1*−*4*) ‹*length t2* = *n*› **have** ‹$t \in \mathcal{T}$ $(P \Theta a \in A.\ Q\ a)$› ‹*Suc n* $\leq$ *length t*›
by (*auto simp add*: *T-Throw*)
with ‹*tF t*› **show** ‹$t \in \mathcal{D}$ *?lhs*›
by (*metis D-restriction-process*$_{ptick}$*I antisym-conv1*)
qed
qed

```
    next
      assume ‹t ∈ 𝒯 (P ⊖ a ∈ A. Q' a) − 𝒟 (P ⊖ a ∈ A. Q' a)›
‹length t = Suc n› ‹tF t›
      from this(1) consider (T-P) ‹t ∈ 𝒯 P› ‹set t ∩ ev ‘ A = {}›
        | (T-Q) t1 a t2 where ‹t = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯
P›
            ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹t2 ∈ 𝒯 (Q' a)›
        unfolding Throw-projs by fast
      thus ‹t ∈ 𝒟 ?lhs›
      proof cases
        case T-P with ‹length t = Suc n› ‹tF t› show ‹t ∈ 𝒟 ?lhs›
       by (auto intro: D-restriction-process_ptick I simp add: Throw-projs)
      next
        case T-Q
       from T-Q(1) ‹length t = Suc n› have ‹length t2 ≤ n› by simp
        with T-Q(5) ‹Q ↓ n = Q' ↓ n› have ‹t2 ∈ 𝒯 (Q a)›
          by (metis T-restriction-process_ptick I
              length-le-in-T-restriction-process_ptick restriction-fun-def)
        with T-Q(1−4) have ‹t ∈ 𝒯 (P ⊖ a ∈ A. Q a)›
          by (auto simp add: T-Throw)
        with ‹tF t› ‹length t = Suc n› show ‹t ∈ 𝒟 ?lhs›
          by (blast intro: D-restriction-process_ptick I)
      qed
    qed
  next
    fix t X assume ‹(t, X) ∈ ℱ ?rhs› ‹t ∉ 𝒟 ?rhs›
    with le-approxD(2) restriction-process_ptick-approx-self
    have ‹(t, X) ∈ ℱ (P ⊖ a ∈ A. Q' a) ∧ t ∉ 𝒟 (P ⊖ a ∈ A. Q' a)›
      by (blast intro: D-restriction-process_ptick I)
    then consider (F-P) ‹(t, X) ∈ ℱ P› ‹set t ∩ ev ‘ A = {}›
      | (F-Q) t1 a t2 where ‹t = t1 @ ev a # t2› ‹t1 @ [ev a] ∈ 𝒯 P›
        ‹set t1 ∩ ev ‘ A = {}› ‹a ∈ A› ‹(t2, X) ∈ ℱ (Q' a)›
      unfolding Throw-projs by blast
    thus ‹(t, X) ∈ ℱ ?lhs›
    proof cases
      case F-P thus ‹(t, X) ∈ ℱ ?lhs›
        by (auto intro: F-restriction-process_ptick I simp add: F-Throw)
    next
      case F-Q
      with ‹(t, X) ∈ ℱ (P ⊖ a ∈ A. Q' a) ∧ t ∉ 𝒟 (P ⊖ a ∈ A. Q'
a)›
        ‹(t, X) ∈ ℱ ?rhs› ‹t ∉ 𝒟 ?rhs› have ‹length t2 ≤ n›
        by (metis D-restriction-process_ptick I F-T add-leE length-Cons
            length-append linorder-not-le not-less-eq-eq)
      with F-Q(5) consider ‹length t2 < n› | ‹tF t2› ‹length t2 = n›
        | t2' r where ‹t2 = t2' @ [✓(r)]› ‹length t2 = n›
        by (metis is-processT2 not-tF-and-ftF order-neq-le-trans)
      thus ‹(t, X) ∈ ℱ ?lhs›
      proof cases
```

assume ‹*length t2* < *n*›
with *F-Q*(*5*) ‹*Q* ↓ *n* = *Q'* ↓ *n*› **have** ‹(*t2*, *X*) ∈ $\mathcal{F}$ (*Q a*)›
by (*metis F-restriction-process*$_{ptick}$*I length-less-in-F-restriction-process*$_{ptick}$ *restriction-fun-def*)
with *F-Q*(*1−4*) **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*auto intro*: *F-restriction-process*$_{ptick}$*I simp add*: *F-Throw*)
next
assume ‹*tF t2*› ‹*length t2* = *n*›
from *this*(*2*) *F-Q*(*5*) ‹*Q* ↓ *n* = *Q'* ↓ *n*› **have** ‹*t2* ∈ $\mathcal{T}$ (*Q a*)›
by (*metis F-T T-restriction-process*$_{ptick}$*I dual-order.refl length-le-in-T-restriction-process*$_{ptick}$ *restriction-fun-def*)
with *F-Q*(*1−4*) ‹*tF t2*› ‹*length t2* = *n*› **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*auto intro*!: *F-restriction-process*$_{ptick}$*I simp add*: *T-Throw*)
next
fix *t2' r* **assume** ‹*t2* = *t2'* @ [✓(*r*)]› ‹*length t2* = *n*›
from *this*(*2*) *F-Q*(*5*) ‹*Q* ↓ *n* = *Q'* ↓ *n*› **have** ‹*t2* ∈ $\mathcal{T}$ (*Q a*)›
by (*metis F-T T-restriction-process*$_{ptick}$*I dual-order.refl length-le-in-T-restriction-process*$_{ptick}$ *restriction-fun-def*)
with *F-Q*(*1−4*) **have** ‹*t* ∈ $\mathcal{T}$ *?lhs*›
by (*auto intro*!: *T-restriction-process*$_{ptick}$*I simp add*: *T-Throw*)
with *F-Q*(*1*) ‹*t2* = *t2'* @ [✓(*r*)]› **show** ‹(*t*, *X*) ∈ $\mathcal{F}$ *?lhs*›
by (*metis Cons-eq-appendI append-assoc tick-T-F*)
qed
qed
qed
qed

8 Non Destructiveness of Interrupt

8.1 Refinement

lemma *restriction-process*$_{ptick}$*-Interrupt-FD* :
‹*P* △ *Q* ↓ *n* ⊑$_{FD}$ (*P* ↓ *n*) △ (*Q* ↓ *n*)› (**is** ‹*?lhs* ⊑$_{FD}$ *?rhs*›)
proof (*unfold refine-defs*, *safe*)
show ∗ : ‹*t* ∈ $\mathcal{D}$ *?lhs*› **if** ‹*t* ∈ $\mathcal{D}$ *?rhs*› **for** *t*
proof −
from ‹*t* ∈ $\mathcal{D}$ *?rhs*› **consider** ‹*t* ∈ $\mathcal{D}$ (*P* ↓ *n*)›
| *u v* **where** ‹*t* = *u* @ *v*› ‹*u* ∈ $\mathcal{T}$ (*P* ↓ *n*)› ‹*tF u*› ‹*v* ∈ $\mathcal{D}$ (*Q* ↓ *n*)›
by (*simp add*: *D-Interrupt*) *blast*
thus ‹*t* ∈ $\mathcal{D}$ *?lhs*›
proof *cases*
show ‹*t* ∈ $\mathcal{D}$ (*P* ↓ *n*) ⟹ *t* ∈ $\mathcal{D}$ *?lhs*›
by (*elim D-restriction-process*$_{ptick}$*E*) (*auto simp add*: *D-restriction-process*$_{ptick}$ *Interrupt-projs*)
next
fix *u v* **assume** ‹*t* = *u* @ *v*› ‹*u* ∈ $\mathcal{T}$ (*P* ↓ *n*)› ‹*tF u*› ‹*v* ∈ $\mathcal{D}$ (*Q*

```
↓ n)›
      from ‹v ∈ 𝒟 (Q ↓ n)› show ‹t ∈ 𝒟 ?lhs›
      proof (elim D-restriction-process_ptick E)
        from ‹t = u @ v› ‹u ∈ 𝒯 (P ↓ n)› ‹tF u› show ‹v ∈ 𝒟 Q ⟹
t ∈ 𝒟 ?lhs›
         by (auto simp add: restriction-process_ptick-projs Interrupt-projs
              D-imp-ftF ftF-append)
      next
        fix w x assume ‹v = w @ x› ‹w ∈ 𝒯 Q› ‹length w = n› ‹tF
w› ‹ftF x›
        from ‹u ∈ 𝒯 (P ↓ n)› consider ‹u ∈ 𝒟 (P ↓ n)› | ‹u ∈ 𝒯 P›
‹length u ≤ n›
       by (elim T-restriction-process_ptick E) (auto simp add: D-restriction-process_ptick)
        thus ‹t ∈ 𝒟 ?lhs›
        proof cases
          assume ‹u ∈ 𝒟 (P ↓ n)›
        with D-imp-ftF ‹t = u @ v› ‹tF u› ‹v ∈ 𝒟 (Q ↓ n)› is-processT7
          have ‹t ∈ 𝒟 (P ↓ n)› by blast
          thus ‹t ∈ 𝒟 ?lhs› by (elim D-restriction-process_ptick E)
              (auto simp add: D-restriction-process_ptick Interrupt-projs)
        next
          assume ‹u ∈ 𝒯 P› ‹length u ≤ n›
          hence ‹t = take n (u @ w) @ drop (n − length u) w @ x ∧
                take n (u @ w) ∈ 𝒯 (P △ Q) ∧ length (take n (u @ w))
= n ∧
                tF (take n (u @ w)) ∧ ftF (drop (n − length u) w @ x)›
            by (simp add: ‹t = u @ v› ‹v = w @ x› ‹length w = n› ‹tF
u› T-Interrupt)
             (metis ‹ftF x› ‹tF u› ‹tF w› ‹w ∈ 𝒯 Q› append-take-drop-id
                ftF-append is-processT3-TR-append tF-append-iff)
          with D-restriction-process_ptick show ‹t ∈ 𝒟 ?lhs› by blast
        qed
      qed
    qed
  qed

  show ‹(t, X) ∈ ℱ ?rhs ⟹ (t, X) ∈ ℱ ?lhs› for t X
    by (meson ∗ is-processT8 mono-Interrupt le-approxD(2) restric-
tion-process_ptick-approx-self)
qed
```

8.2 Non Destructiveness

```
lemma Interrupt-non-destructive :
  ‹non-destructive (λ(P :: ('a, 'r) process_ptick, Q). P △ Q)›
proof (rule order-non-destructiveI, clarify)
  fix P P' Q Q' :: ‹('a, 'r) process_ptick› and n
  assume ‹(P, Q) ↓ n = (P', Q') ↓ n› ‹0 < n›
  hence ‹P ↓ n = P' ↓ n› ‹Q ↓ n = Q' ↓ n›
```

by (*simp-all add*: *restriction-prod-def*)
show ‹$P \triangle Q \downarrow n \sqsubseteq_{FD} P' \triangle Q' \downarrow n$›
proof (*rule leFD-restriction-process$_{ptick}$I*)
show ‹$t \in \mathcal{D}\ (P' \triangle Q') \Longrightarrow t \in \mathcal{D}\ (P \triangle Q \downarrow n)$› **for** t
by (*metis* (*mono-tags*, *opaque-lifting*) ‹$P \downarrow n = P' \downarrow n$› ‹$Q \downarrow n = Q' \downarrow n$›
in-mono le-FD-D(*1*) *mono-Interrupt-FD restriction-process$_{ptick}$-FD-self*
restriction-process$_{ptick}$-Interrupt-FD)
next
show ‹$(s, X) \in \mathcal{F}\ (P' \triangle Q') \Longrightarrow (s, X) \in \mathcal{F}\ (P \triangle Q \downarrow n)$› **for** s X
by (*metis* (*mono-tags*, *opaque-lifting*) ‹$P \downarrow n = P' \downarrow n$› ‹$Q \downarrow n = Q' \downarrow n$›
failure-refine-def leFD-imp-leF mono-Interrupt-FD
restriction-process$_{ptick}$-FD-self
restriction-process$_{ptick}$-Interrupt-FD subset-iff)
qed
qed

9 Non too Destructiveness of After

9.1 Equality

lemma *initials-restriction-process$_{ptick}$*: ‹$(P \downarrow n)^0 = (\mathit{if}\ n = 0\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ P^0)$›
by (*cases n*, *solves simp*)
(*auto simp add*: *initials-def T-restriction-process$_{ptick}$*,
metis append.right-neutral append-eq-conv-conj drop-Nil drop-Suc-Cons)

lemma (**in** *After*) *restriction-process$_{ptick}$-After*:
‹$P\ \mathit{after}\ e \downarrow n = (\mathit{if}\ \mathit{ev}\ e \in P^0\ \mathit{then}\ (P \downarrow \mathit{Suc}\ n)\ \mathit{after}\ e\ \mathit{else}\ \Psi\ P\ e \downarrow n)$›
proof (*split if-split*, *intro conjI impI*)
show ‹$\mathit{ev}\ e \notin P^0 \Longrightarrow P\ \mathit{after}\ e \downarrow n = \Psi\ P\ e \downarrow n$› **by** (*simp add*: *not-initial-After*)
next
assume ‹$\mathit{ev}\ e \in P^0$›
show ‹$P\ \mathit{after}\ e \downarrow n = (P \downarrow \mathit{Suc}\ n)\ \mathit{after}\ e$› (**is** ‹*?lhs* = *?rhs*›)
proof (*subst Process-eq-spec*, *safe*)
show ‹$t \in \mathcal{D}$ *?lhs* $\Longrightarrow t \in \mathcal{D}$ *?rhs*› **for** t
by (*elim D-restriction-process$_{ptick}$E*)
(*simp-all add*: ‹$\mathit{ev}\ e \in P^0$› *After-projs*
initials-restriction-process$_{ptick}$ D-restriction-process$_{ptick}$,
meson Cons-eq-appendI event$_{ptick}$.disc(*1*) *length-Cons tF-Cons-iff*)

next
show ‹$t \in \mathcal{D}$ *?rhs* $\Longrightarrow$ $t \in \mathcal{D}$ *?lhs*› **for** t
by (*auto simp add*: *D-After initials-restriction-process*$_{ptick}$ ‹*ev e* $\in P^0$›
D-restriction-process$_{ptick}$ *T-After Cons-eq-append-conv*)
next
show ‹$(t, X) \in \mathcal{F}$ *?lhs* $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?rhs*› **for** t X
by (*elim F-restriction-process*$_{ptick}E$)
(*simp-all add*: ‹*ev e* $\in P^0$› *After-projs*
initials-restriction-process$_{ptick}$ *F-restriction-process*$_{ptick}$,
meson Cons-eq-appendI event$_{ptick}$*.disc*(*1*) *length-Cons tF-Cons-iff*)
next
show ‹$(t, X) \in \mathcal{F}$ *?rhs* $\Longrightarrow$ $(t, X) \in \mathcal{F}$ *?lhs*› **for** t X
by (*auto simp add*: *F-After initials-restriction-process*$_{ptick}$ ‹*ev e* $\in P^0$›
F-restriction-process$_{ptick}$ *T-After Cons-eq-append-conv*)
qed
qed

lemma (**in** *AfterExt*) *restriction-process*$_{ptick}$*-After*$_{tick}$:
‹P *after*$_{\checkmark}$ $e \downarrow n$ =
(*case e of* ✓(*r*) ⇒ Ω P *r* $\downarrow$ *n* | *ev x* ⇒ *if* $e \in P^0$ *then* ($P \downarrow$ *Suc n*) *after*$_{\checkmark}$ *e else* Ψ P *x* $\downarrow$ *n*)›
by (*simp add*: *After*$_{tick}$*-def restriction-process*$_{ptick}$*-After split*: *event*$_{ptick}$*.split*)

lemma (**in** *AfterExt*) *restriction-process*$_{ptick}$*-After*$_{trace}$:
‹$t \in \mathcal{T}$ $P \Longrightarrow$ *tF t* $\Longrightarrow$ P *after*$_{\mathcal{T}}$ $t \downarrow n = (P \downarrow (n + length\ t))$ *after*$_{\mathcal{T}}$ *t*›
proof (*induct t arbitrary*: *n rule*: *rev-induct*)
show ‹P *after*$_{\mathcal{T}}$ $[] \downarrow n = (P \downarrow (n + length\ []))$ *after*$_{\mathcal{T}}$ $[]$› **for** n **by** *simp*
next
fix *e t n*
assume *hyp* : ‹$t \in \mathcal{T}$ $P \Longrightarrow$ *tF t* $\Longrightarrow$ P *after*$_{\mathcal{T}}$ $t \downarrow n = (P \downarrow (n + length\ t))$ *after*$_{\mathcal{T}}$ *t*› **for** *n*
assume *prems* : ‹t @ $[e] \in \mathcal{T}$ P› ‹*tF* (t @ $[e]$)›
from *prems*(*2*) **obtain** *a* **where** ‹$e = ev\ a$› **by** (*cases e*) *simp-all*
with *initials-After*$_{trace}$[*OF prems*(*1*)] **have** ‹*ev a* $\in (P\ after_{\mathcal{T}}\ t)^0$› **by** *simp*
from *prems is-processT3-TR-append* **have** ‹$t \in \mathcal{T}$ P› ‹*tF t*› **by** *auto*
from *hyp*[*OF this*] **have** ‹P *after*$_{\mathcal{T}}$ $t \downarrow$ *Suc n* = ($P \downarrow$ (*Suc n* + *length t*)) *after*$_{\mathcal{T}}$ *t*› **.**
thus ‹P *after*$_{\mathcal{T}}$ (t @ $[e]$) $\downarrow n = (P \downarrow (n + length\ (t\ @\ [e])))$ *after*$_{\mathcal{T}}$ (t @ $[e]$)›
by (*simp add*: *After*$_{trace}$*-snoc restriction-process*$_{ptick}$*-After*$_{tick}$
‹$e = ev\ a$› ‹*ev a* $\in (P\ after_{\mathcal{T}}\ t)^0$›)
qed

9.2 Non too Destructiveness

lemma (**in** *After*) *non-too-destructive-on-After* :
‹*non-too-destructive-on* (λ*P*. *P after e*) {*P*. *ev e* ∈ P^0}›
by (*auto intro*!: *non-too-destructive-onI simp add*: $restriction\text{-}process_{ptick}$-*After*)

lemma (**in** *AfterExt*) $non\text{-}too\text{-}destructive\text{-}on\text{-}After_{tick}$:
‹*non-too-destructive-on* (λ*P*. *P* $after_{✓}$ *e*) {*P*. *e* ∈ P^0}›
if ‹⋀*r*. *e* = ✓(*r*) ⟹ *non-too-destructive-on* Ω {*P*. ✓(*r*) ∈ P^0}›
proof (*intro non-too-destructive-onI*, *clarify*)
fix *P Q n* **assume** ∗ : ‹*P* ↓ *Suc n* = *Q* ↓ *Suc n*› ‹*e* ∈ P^0› ‹*e* ∈ Q^0›
show ‹*P* $after_{✓}$ *e* ↓ *n* = *Q* $after_{✓}$ *e* ↓ *n*›
proof (*cases e*)
from ∗ **show** ‹*e* = *ev a* ⟹ *P* $after_{✓}$ *e* ↓ *n* = *Q* $after_{✓}$ *e* ↓ *n*› **for** *a*
by (*simp add*: $restriction\text{-}process_{ptick}$-$After_{tick}$)
next
fix *r* **assume** ‹*e* = ✓(*r*)›
with ∗(*2*, *3*) **have** ‹*P* ∈ {*P*. ✓(*r*) ∈ P^0}› ‹*Q* ∈ {*P*. ✓(*r*) ∈ P^0}›
by *auto*
from *non-too-destructive-onD*[*OF that*[*simplified* ‹*e* = ✓(*r*)›, *OF refl*] *this* ∗(*1*)]
have ‹Ω *P* ↓ *n* = Ω *Q* ↓ *n*› **.**
with ‹*e* = ✓(*r*)› **show** ‹*P* $after_{✓}$ *e* ↓ *n* = *Q* $after_{✓}$ *e* ↓ *n*›
by (*simp add*: $restriction\text{-}process_{ptick}$-$After_{tick}$) (*metis restriction-fun-def*)
qed
qed

lemma (**in** *After*) *non-too-destructive-After* :
‹*non-too-destructive* (λ*P*. *P after e*)› **if** ∗ : ‹*non-too-destructive-on* Ψ {*P*. *ev e* ∉ P^0}›
proof (*rule non-too-destructiveI*)
fix *P Q* :: ‹('*a*, '*r*) $process_{ptick}$› **and** *n*
assume ‹*P* ↓ *Suc n* = *Q* ↓ *Suc n*›
hence ‹*ev e* ∈ P^0 ∧ *ev e* ∈ Q^0 ∨ *ev e* ∉ P^0 ∧ *ev e* ∉ Q^0›
by (*metis initials-restriction-process*$_{ptick}$ *nat.distinct*(*1*))
thus ‹*P after e* ↓ *n* = *Q after e* ↓ *n*›
proof (*elim disjE conjE*)
show ‹*ev e* ∈ P^0 ⟹ *ev e* ∈ Q^0 ⟹ *P after e* ↓ *n* = *Q after e* ↓ *n*›
by (*simp add*: $restriction\text{-}process_{ptick}$-*After* ‹*P* ↓ *Suc n* = *Q* ↓ *Suc n*›)
next
assume ‹*ev e* ∉ P^0› ‹*ev e* ∉ Q^0›
hence ‹*P after e* = Ψ *P e*› ‹*Q after e* = Ψ *Q e*›
by (*simp-all add*: *not-initial-After*)
from ‹*P* ↓ *Suc n* = *Q* ↓ *Suc n*› **have** ‹Ψ *P* ↓ *n* = Ψ *Q* ↓ *n*›

```
    by (intro ∗[THEN non-too-destructive-onD, of P Q])
      (simp-all add: ‹ev e ∉ P⁰› ‹ev e ∉ Q⁰›)
  with ‹P after e = Ψ P e› ‹Q after e = Ψ Q e›
  show ‹P after e ↓ n = Q after e ↓ n›
    by (metis restriction-fun-def)
 qed
qed


lemma (in AfterExt) non-too-destructive-After_tick :
 ‹non-too-destructive (λP. P after✓ e)›
 if ∗ : ‹⋀a. e = ev a ⟹ non-too-destructive-on Ψ {P. ev a ∉ P⁰}›
  ‹⋀r. e = ✓(r) ⟹ non-too-destructive (λP. Ω P r)›
proof (rule non-too-destructiveI)
 show ‹P after✓ e ↓ n = Q after✓ e ↓ n› if ‹P ↓ Suc n = Q ↓ Suc
n› for P Q n
 proof (cases e)
  show ‹P after✓ e ↓ n = Q after✓ e ↓ n› if ‹e = ev a› for a
    by (simp add: After_tick-def ‹e = ev a›)
      (fact non-too-destructive-After[OF ∗(1)[simplified ‹e = ev a›,
OF refl],
        THEN non-too-destructiveD, OF ‹P ↓ Suc n = Q ↓ Suc n›])
 next
  fix r assume ‹e = ✓(r)›
  from ∗(2)[unfolded ‹e = ✓(r)›, OF refl,
      THEN non-too-destructiveD, OF ‹P ↓ Suc n = Q ↓ Suc n›]
  have ‹Ω P r ↓ n = Ω Q r ↓ n› .
  thus ‹P after✓ e ↓ n = Q after✓ e ↓ n›
    by (simp add: After_tick-def ‹e = ✓(r)›)
 qed
qed


lemma (in AfterExt) restriction-shift-After_trace :
 ‹restriction-shift (λP. P after_T t) (− int (length t))›
 if ‹non-too-destructive Ψ› ‹non-too-destructive Ω›
  — We could imagine more precise assumptions, but is it useful?
proof (induct t)
 case Nil show ?case by (simp add: restriction-shiftI)
next
 case (Cons e t)
 have ‹non-too-destructive (λP. P after✓ e)›
  by (rule non-too-destructive-After_tick)
    (simp add: ‹non-too-destructive Ψ›,
      metis non-too-destructive-fun-iff ‹non-too-destructive Ω›)
 hence ∗ : ‹restriction-shift (λP. P after✓ e) (− 1)›
  unfolding non-too-destructive-def
    non-too-destructive-on-def restriction-shift-def .
 from restriction-shift-comp-restriction-shift[OF Cons.hyps ∗]
```

show *?case* **by** *simp*
qed

10 Destructiveness of Hiding

10.1 Refinement

lemma *Hiding-restriction-process$_{ptick}$-FD* : ‹$(P \downarrow n) \setminus S \sqsubseteq_{FD} P \setminus S \downarrow n$›
proof (*unfold refine-defs*, *safe*)
show $*$: ‹$t \in \mathcal{D}\ (P \setminus S \downarrow n) \Longrightarrow t \in \mathcal{D}\ ((P \downarrow n) \setminus S)$› **for** t
proof (*elim D-restriction-process$_{ptick}$E*)
show ‹$t \in \mathcal{D}\ (P \setminus S) \Longrightarrow t \in \mathcal{D}\ ((P \downarrow n) \setminus S)$›
by (*simp add*: *D-Hiding restriction-process$_{ptick}$-projs*) *blast*
next
fix $u\ v$
assume ‹$t = u\ @\ v$› ‹$u \in \mathcal{T}\ (P \setminus S)$› ‹*length* $u = n$› ‹*tF* u› ‹*ftF* v›
from ‹$u \in \mathcal{T}\ (P \setminus S)$›[*simplified T-Hiding*, *simplified*]
consider ‹$u \in \mathcal{D}\ (P \setminus S)$› | u' **where** ‹$u =$ *trace-hide* $u'\ (ev\ `\ S)$› ‹$(u', ev\ `\ S) \in \mathcal{F}\ P$›
by (*simp add*: *F-Hiding D-Hiding*) *blast*
thus ‹$t \in \mathcal{D}\ ((P \downarrow n) \setminus S)$›
proof *cases*
assume ‹$u \in \mathcal{D}\ (P \setminus S)$›
hence ‹$t \in \mathcal{D}\ (P \setminus S)$› **by** (*simp add*: ‹*ftF* v› ‹$t = u\ @\ v$› ‹*tF* u› *is-processT7*)
with *restriction-process$_{ptick}$-approx-self le-approxD*(*1*) *mono-Hiding*
show ‹$t \in \mathcal{D}\ ((P \downarrow n) \setminus S)$› **by** *blast*
next
fix u' **assume** ‹$u =$ *trace-hide* $u'\ (ev\ `\ S)$› ‹$(u', ev\ `\ S) \in \mathcal{F}\ P$›
with ‹*length* $u = n$› ‹*tF* u› *Hiding-tF length-filter-le F-T*
have ‹$n \leq$ *length* u'› ‹*tF* u'› ‹$u' \in \mathcal{T}\ P$› **by** *blast+*
with ‹$u =$ *trace-hide* $u'\ (ev\ `\ S)$›
have ‹$u' = (take\ n\ u')\ @\ (drop\ n\ u') \wedge take\ n\ u' \in \mathcal{T}\ P \wedge$
length $(take\ n\ u') = n \wedge tF\ (take\ n\ u') \wedge ftF\ (drop\ n\ u')$›
by (*simp add*: *min-def*) (*metis append-take-drop-id is-processT3-TR-append tF-append-iff tF-imp-ftF*)
with *D-restriction-process$_{ptick}$* **have** ‹$u' \in \mathcal{D}\ (P \downarrow n)$› **by** *blast*
with *Hiding-tF* ‹*ftF* v› ‹$t = u\ @\ v$› ‹$u =$ *trace-hide* $u'\ (ev\ `\ S)$› ‹*tF* u›
show ‹$t \in \mathcal{D}\ ((P \downarrow n) \setminus S)$› **by** (*simp add*: *D-Hiding*) *blast*
qed
qed

fix $s\ X$
assume ‹$(s, X) \in \mathcal{F}\ ((P \setminus S) \downarrow n)$›

then consider ‹$s \in \mathcal{D}\ ((P \setminus S) \downarrow n)$› | ‹$(s,\ X) \in \mathcal{F}\ (P \setminus S)$›
unfolding *restriction-process$_{ptick}$-projs* **by** *blast*
thus ‹$(s,\ X) \in \mathcal{F}\ ((P \downarrow n) \setminus S)$›
proof *cases*
from $*$ *D-F* **show** ‹$s \in \mathcal{D}\ ((P \setminus S) \downarrow n) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((P \downarrow n) \setminus S)$› **by** *blast*
next
show ‹$(s,\ X) \in \mathcal{F}\ (P \setminus S) \Longrightarrow (s,\ X) \in \mathcal{F}\ ((P \downarrow n) \setminus S)$›
using *restriction-process$_{ptick}$-approx-self D-F mono-Hiding le-approxD*(*2*)
by *blast*
qed
qed

10.2 Destructiveness

lemma *Hiding-destructive* :
‹$\exists P\ Q :: ('a,\ 'r)\ process_{ptick}.\ P \downarrow n = Q \downarrow n \wedge (P \setminus S) \downarrow Suc\ 0 \neq (Q \setminus S) \downarrow Suc\ 0$› **if** ‹$S \neq \{\}$›
proof −
from ‹$S \neq \{\}$› **obtain** *e* **where** ‹$e \in S$› **by** *blast*
define P :: ‹$('a,\ 'r)\ process_{ptick}$› **where** ‹$P \equiv iterate\ (Suc\ n) \cdot (\Lambda\ X.\ write0\ e\ X) \cdot (SKIP\ undefined)$›
define Q :: ‹$('a,\ 'r)\ process_{ptick}$› **where** ‹$Q \equiv iterate\ (Suc\ n) \cdot (\Lambda\ X.\ write0\ e\ X) \cdot STOP$›
have ‹$P \downarrow n = Q \downarrow n$›
unfolding *P-def Q-def* **by** (*induct n*) (*simp-all add*: *restriction-process$_{ptick}$-write0*)

have ‹$P \setminus S = SKIP\ undefined$›
unfolding *P-def* **by** (*induct n*) (*simp-all add*: *Hiding-write0-non-disjoint* ‹$e \in S$›)
hence ‹$(P \setminus S) \downarrow Suc\ 0 = SKIP\ undefined$› **by** *simp*
have ‹$Q \setminus S = STOP$›
unfolding *Q-def* **by** (*induct n*) (*simp-all add*: *Hiding-write0-non-disjoint* ‹$e \in S$›)
hence ‹$(Q \setminus S) \downarrow Suc\ 0 = STOP$› **by** *simp*

have ‹$P \downarrow n = Q \downarrow n \wedge (P \setminus S) \downarrow Suc\ 0 \neq (Q \setminus S) \downarrow Suc\ 0$›
by (*simp add*: ‹$P \downarrow n = Q \downarrow n$› ‹$(P \setminus S) \downarrow Suc\ 0 = SKIP\ undefined$› ‹$(Q \setminus S) \downarrow Suc\ 0 = STOP$›)
thus *?thesis* **by** *blast*
qed

11 Admissibility

named-theorems *restriction-adm-process$_{ptick}$-simpset*

11.1 Belonging

lemma *restriction-adm-in-D* [*restriction-adm-process$_{ptick}$-simpset] :*
‹adm$_{\downarrow}$ (λx. t ∈ 𝒟 (f x))›
and *restriction-adm-notin-D* [*restriction-adm-process$_{ptick}$-simpset] :*
‹adm$_{\downarrow}$ (λx. t ∉ 𝒟 (f x))›
and *restriction-adm-in-F* [*restriction-adm-process$_{ptick}$-simpset] :*
‹adm$_{\downarrow}$ (λx. (t, X) ∈ ℱ (f x))›
and *restriction-adm-notin-F* [*restriction-adm-process$_{ptick}$-simpset] :*
‹adm$_{\downarrow}$ (λx. (t, X) ∉ ℱ (f x))› **if** *‹cont$_{\downarrow}$ f›*
for *f* :: *‹'b :: restriction ⇒ ('a, 'r) process$_{ptick}$›*
proof (*all ‹rule restriction-adm-subst[OF ‹cont$_{\downarrow}$ f›]›*)
have ∗ : *‹σ −↓→ Σ ⟹ ∃ n0. ∀ n ≥ n0. σ n ↓ Suc (length t) = Σ ↓ Suc (length t)›*
for *σ* **and** Σ :: *‹('a, 'r) process$_{ptick}$›*
by (*metis restriction-tendstoD*)

show *‹adm$_{\downarrow}$ (λx. t ∈ 𝒟 x)› ‹adm$_{\downarrow}$ (λx. t ∉ 𝒟 x)›*
by (*rule restriction-admI*,
metis (*no-types*) ∗ *D-restriction-process$_{ptick}$-Suc-length-iff-D dual-order.refl*)+

show *‹adm$_{\downarrow}$ (λx. (t, X) ∈ ℱ x)› ‹adm$_{\downarrow}$ (λx. (t, X) ∉ ℱ x)›*
by (*rule restriction-admI*,
metis (*no-types*) ∗ *F-restriction-process$_{ptick}$-Suc-length-iff-F dual-order.refl*)+
qed

corollary *restriction-adm-in-T* [*restriction-adm-process$_{ptick}$-simpset]*
:
‹cont$_{\downarrow}$ f ⟹ adm$_{\downarrow}$ (λx. t ∈ 𝒯 (f x))›
and *restriction-adm-notin-T* [*restriction-adm-process$_{ptick}$-simpset] :*
‹cont$_{\downarrow}$ f ⟹ adm$_{\downarrow}$ (λx. t ∉ 𝒯 (f x))›
by (*fact restriction-adm-in-F*[*of f t ‹{}›, simplified T-F-spec*])
(*fact restriction-adm-notin-F*[*of f t ‹{}›, simplified T-F-spec*])

corollary *restriction-adm-in-initials* [*restriction-adm-process$_{ptick}$-simpset]*
:
‹cont$_{\downarrow}$ f ⟹ adm$_{\downarrow}$ (λx. e ∈ (f x)^0)›
and *restriction-adm-notin-initials* [*restriction-adm-process$_{ptick}$-simpset]*
:
‹cont$_{\downarrow}$ f ⟹ adm$_{\downarrow}$ (λx. e ∉ (f x)^0)›
by (*simp-all add*: *initials-def restriction-adm-in-T restriction-adm-notin-T*)

11.2 Refining

corollary *restriction-adm-leF* [*restriction-adm-process$_{ptick}$-simpset*] :
‹$cont_{\downarrow}\ f \Longrightarrow cont_{\downarrow}\ g \Longrightarrow adm_{\downarrow}\ (\lambda x.\ f\ x \sqsubseteq_{F}\ g\ x)$›
by (*simp add*: *failure-refine-def subset-iff restriction-adm-process$_{ptick}$-simpset*)

corollary *restriction-adm-leD* [*restriction-adm-process$_{ptick}$-simpset*] :
‹$cont_{\downarrow}\ f \Longrightarrow cont_{\downarrow}\ g \Longrightarrow adm_{\downarrow}\ (\lambda x.\ f\ x \sqsubseteq_{D}\ g\ x)$›
by (*simp add*: *divergence-refine-def subset-iff restriction-adm-process$_{ptick}$-simpset*)

corollary *restriction-adm-leT* [*restriction-adm-process$_{ptick}$-simpset*] :
‹$cont_{\downarrow}\ f \Longrightarrow cont_{\downarrow}\ g \Longrightarrow adm_{\downarrow}\ (\lambda x.\ f\ x \sqsubseteq_{T}\ g\ x)$›
by (*simp add*: *trace-refine-def subset-iff restriction-adm-process$_{ptick}$-simpset*)

corollary *restriction-adm-leFD* [*restriction-adm-process$_{ptick}$-simpset*]
:
‹$cont_{\downarrow}\ f \Longrightarrow cont_{\downarrow}\ g \Longrightarrow adm_{\downarrow}\ (\lambda x.\ f\ x \sqsubseteq_{FD}\ g\ x)$›
by (*simp add*: *failure-divergence-refine-def restriction-adm-process$_{ptick}$-simpset*)

corollary *restriction-adm-leDT* [*restriction-adm-process$_{ptick}$-simpset*]
:
‹$cont_{\downarrow}\ f \Longrightarrow cont_{\downarrow}\ g \Longrightarrow adm_{\downarrow}\ (\lambda x.\ f\ x \sqsubseteq_{DT}\ g\ x)$›
by (*simp add*: *trace-divergence-refine-def restriction-adm-process$_{ptick}$-simpset*)

11.2.1 Transitions

lemma (**in** *After*) *restriction-cont-After* [*restriction-adm-simpset*] :
‹$cont_{\downarrow}\ (\lambda x.\ f\ x\ after\ a)$› **if** ‹$cont_{\downarrow}\ f$› **and** ‹$cont_{\downarrow}\ \Psi$›
— We could imagine more precise assumptions, but is it useful?
proof (*rule restriction-cont-comp*[*OF* - ‹$cont_{\downarrow}\ f$›])
show ‹$cont_{\downarrow}\ (\lambda P.\ P\ after\ a)$›
proof (*rule restriction-contI*)
show ‹$(\lambda n.\ \sigma\ n\ after\ a) \ -\downarrow\rightarrow\ \Sigma\ after\ a$› **if** ‹$\sigma\ -\downarrow\rightarrow\ \Sigma$› **for** $\sigma\ \Sigma$
proof (*rule restriction-tendstoI*)
fix n
from ‹$\sigma\ -\downarrow\rightarrow\ \Sigma$› **obtain** *n0*
where $*$: ‹$\forall\, k{\geq}n0.\ \Sigma \downarrow Suc\ n = \sigma\ k \downarrow Suc\ n$›
by (*blast dest*: *restriction-tendstoD*)
consider ‹$ev\ a \in \Sigma^0$› ‹$\forall\, n{\geq}Suc\ n0.\ ev\ a \in (\sigma\ n)^0$›
| ‹$ev\ a \notin \Sigma^0$› ‹$\forall\, n{\geq}Suc\ n0.\ ev\ a \notin (\sigma\ n)^0$›
by (*metis* $*$ *Suc-leD initials-restriction-process$_{ptick}$ nat.distinct*(*1*))
thus ‹$\exists\, n0.\ \forall\, k{\geq}n0.\ \Sigma\ after\ a \downarrow n = \sigma\ k\ after\ a \downarrow n$›
proof *cases*
assume ‹$ev\ a \in \Sigma^0$› ‹$\forall\, n{\geq}Suc\ n0.\ ev\ a \in (\sigma\ n)^0$›
hence ‹$\forall\, k{\geq}Suc\ n0.\ \Sigma\ after\ a \downarrow n = \sigma\ k\ after\ a \downarrow n$›
by (*metis* $*$ *Suc-leD restriction-process$_{ptick}$-After*)
thus ‹$\exists\, n0.\ \forall\, k{\geq}n0.\ \Sigma\ after\ a \downarrow n = \sigma\ k\ after\ a \downarrow n$› **..**
next
assume ‹$ev\ a \notin \Sigma^0$› ‹$\forall\, n{\geq}Suc\ n0.\ ev\ a \notin (\sigma\ n)^0$›
hence ‹$\Sigma\ after\ a = \Psi\ \Sigma\ a$› ‹$\forall\, k{\geq}Suc\ n0.\ \sigma\ k\ after\ a = \Psi\ (\sigma$

```
k) a›
        by (simp-all add: not-initial-After)
      moreover from ‹cont↓ Ψ›[THEN restriction-contD]
      obtain n1 where ‹∀k≥n1. Ψ Σ ↓ n = Ψ (σ k) ↓ n›
        by (blast intro: ‹σ −↓→ Σ› dest: restriction-tendstoD)
      ultimately have ‹∀k≥max n1 (Suc n0). Σ after a ↓ n = σ k
after a ↓ n›
        by simp (metis restriction-fun-def)
      thus ‹∃n0. ∀k≥n0. Σ after a ↓ n = σ k after a ↓ n› ..
    qed
  qed
 qed
qed

lemma (in AfterExt) restriction-cont-After_tick [restriction-adm-simpset]
:
 ‹cont↓ (λx. f x after✓ e)› if ‹cont↓ f› and ‹cont↓ Ψ› and ‹cont↓ Ω›
 — We could imagine more precise assumptions, but is it useful?
proof (cases e)
 show ‹e = ev a ⟹ cont↓ (λx. f x after✓ e)› for a
  by (simp add: After_tick-def restriction-cont-After ‹cont↓ f› ‹cont↓
Ψ›)
next
 fix r assume ‹e = ✓(r)›
 hence ‹(λx. f x after✓ e) = (λx. Ω (f x) r)› by (simp add: Af-
ter_tick-def)
 thus ‹cont↓ (λx. f x after✓ e)›
  by (metis restriction-cont-comp restriction-cont-fun-imp that(1,3))
qed

lemma (in AfterExt) restriction-cont-After_trace [restriction-adm-simpset]
:
 ‹cont↓ (λx. f x after𝒯 t)› if ‹cont↓ f› and ‹cont↓ Ψ› and ‹cont↓ Ω›
 — We could imagine more precise assumptions, but is it useful?
proof (rule restriction-cont-comp[OF - ‹cont↓ f›])
 show ‹cont↓ (λP. P after𝒯 t)›
 proof (induct t)
  show ‹cont↓ (λP. P after𝒯 [])› by simp
 next
  fix e t assume ‹cont↓ (λP. P after𝒯 t)›
  show ‹cont↓ (λP. P after𝒯 (e # t))›
    by (simp, rule restriction-cont-comp[OF ‹cont↓ (λP. P after𝒯
t)›])
      (simp add: restriction-cont-After_tick ‹cont↓ Ψ› ‹cont↓ Ω›)
 qed
qed


lemma (in OpSemTransitions) restriction-adm-weak-ev-trans [restriction-adm-process_ptick-simpset]:
```

— Could be weakened to a continuity assumption on Ψ.
fixes $f\ g :: \langle 'b :: \mathit{restriction} \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}\rangle$
assumes τ-*trans-restriction-adm*:
$\langle \bigwedge f\ g :: 'b \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}.\ \mathit{cont}_{\downarrow}\ f \Longrightarrow \mathit{cont}_{\downarrow}\ g \Longrightarrow \mathit{adm}_{\downarrow}$
$(\lambda x.\ f\ x \rightsquigarrow_{\tau} g\ x)\rangle$
and $\langle \mathit{cont}_{\downarrow}\ f\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ g\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Psi\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Omega\rangle$
shows $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ f\ x \rightsquigarrow_{e} g\ x)\rangle$
proof (*unfold ev-trans-def*, *intro restriction-adm-conj*)
show $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ \mathit{ev}\ e \in (f\ x)^0)\rangle$
by (*simp add*: $\langle \mathit{cont}_{\downarrow}\ f\rangle$ *restriction-adm-in-initials*)
next
show $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ f\ x\ \mathit{after}_{\checkmark}\ \mathit{ev}\ e \rightsquigarrow_{\tau} g\ x)\rangle$
proof (*rule* τ-*trans-restriction-adm*[*OF* - $\langle \mathit{cont}_{\downarrow}\ g\rangle$],
rule restriction-cont-comp[*OF* - $\langle \mathit{cont}_{\downarrow}\ f\rangle$])
show $\langle \mathit{cont}_{\downarrow}\ (\lambda x.\ x\ \mathit{after}_{\checkmark}\ \mathit{ev}\ e)\rangle$
by (*simp add*: $\langle \mathit{cont}_{\downarrow}\ \Psi\rangle$ $\langle \mathit{cont}_{\downarrow}\ \Omega\rangle$ *restriction-cont-After*$_{tick}$)
qed
qed

lemma (**in** *OpSemTransitions*) *restriction-adm-weak-tick-trans* [*restriction-adm-process*$_{ptick}$*-simpset*]:
fixes $f\ g :: \langle 'b :: \mathit{restriction} \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}\rangle$
assumes τ-*trans-restriction-adm*:
$\langle \bigwedge f\ g :: 'b \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}.\ \mathit{cont}_{\downarrow}\ f \Longrightarrow \mathit{cont}_{\downarrow}\ g \Longrightarrow \mathit{adm}_{\downarrow}$
$(\lambda x.\ f\ x \rightsquigarrow_{\tau} g\ x)\rangle$
and $\langle \mathit{cont}_{\downarrow}\ f\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ g\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Psi\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Omega\rangle$
shows $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ f\ x \rightsquigarrow_{\checkmark r}\ (g\ x))\rangle$
proof (*unfold tick-trans-def*, *intro restriction-adm-conj*)
show $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ \checkmark(r) \in (f\ x)^0)\rangle$
by (*simp add*: $\langle \mathit{cont}_{\downarrow}\ f\rangle$ *restriction-adm-in-initials*)
next
show $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ f\ x\ \mathit{after}_{\checkmark}\ \checkmark(r) \rightsquigarrow_{\tau} g\ x)\rangle$
proof (*rule* τ-*trans-restriction-adm*[*OF* - $\langle \mathit{cont}_{\downarrow}\ g\rangle$],
rule restriction-cont-comp[*OF* - $\langle \mathit{cont}_{\downarrow}\ f\rangle$])
show $\langle \mathit{cont}_{\downarrow}\ (\lambda x.\ x\ \mathit{after}_{\checkmark}\ \checkmark(r))\rangle$
by (*simp add*: $\langle \mathit{cont}_{\downarrow}\ \Psi\rangle$ $\langle \mathit{cont}_{\downarrow}\ \Omega\rangle$ *restriction-cont-After*$_{tick}$)
qed
qed

lemma (**in** *OpSemTransitions*) *restriction-adm-weak-trace-trans* [*restriction-adm-process*$_{ptick}$*-simpset*]:
fixes $f\ g :: \langle 'b :: \mathit{restriction} \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}\rangle$
assumes τ-*trans-restriction-adm*:
$\langle \bigwedge f\ g :: 'b \Rightarrow ('a, 'r)\ \mathit{process}_{ptick}.\ \mathit{cont}_{\downarrow}\ f \Longrightarrow \mathit{cont}_{\downarrow}\ g \Longrightarrow \mathit{adm}_{\downarrow}$
$(\lambda x.\ f\ x \rightsquigarrow_{\tau} g\ x)\rangle$
and $\langle \mathit{cont}_{\downarrow}\ f\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ g\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Psi\rangle$ **and** $\langle \mathit{cont}_{\downarrow}\ \Omega\rangle$
shows $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ f\ x \rightsquigarrow^{*}\ t\ (g\ x))\rangle$
proof (*subst trace-trans-iff-T-and-After*$_{trace}$*-*τ*-trans*, *intro restriction-adm-conj*)
show $\langle \mathit{adm}_{\downarrow}\ (\lambda x.\ t \in \mathcal{T}\ (f\ x))\rangle$ **by** (*simp add*: $\langle \mathit{cont}_{\downarrow}\ f\rangle$ *restriction-adm-in-T*)
next

show ‹$adm_{\downarrow}$ ($\lambda x.$ f x $after_{\mathcal{T}}$ t $\rightsquigarrow_{\tau}$ g x)›
proof (*rule* τ-*trans-restriction-adm*[*OF* - ‹$cont_{\downarrow}$ g›])
show ‹$cont_{\downarrow}$ ($\lambda x.$ f x $after_{\mathcal{T}}$ t)›
by (*simp add*: ‹$cont_{\downarrow}$ f› ‹$cont_{\downarrow}$ Ψ› ‹$cont_{\downarrow}$ Ω› *restriction-cont-After*$_{trace}$)
qed
qed

declare *restriction-adm-process*$_{ptick}$-*simpset* [*simp*]

12 Higher-Order Rules

This is the main entry point. We configure the simplifier below.

named-theorems *restriction-shift-process*$_{ptick}$-*simpset*

12.1 Prefixes

lemma *Mprefix-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$-*simpset*]
:
‹*constructive* ($\lambda x.$ $\Box a \in A \rightarrow f\ a\ x$)› **if** ‹($\bigwedge a.$ $a \in A \Longrightarrow$ *non-destructive* (f a))›
proof −
have $*$: ‹$\Box a \in A \rightarrow f\ a\ x = \Box a \in A \rightarrow$ (*if* $a \in A$ *then* f a x *else* *STOP*)› **for** x
by (*auto intro*: *mono-Mprefix-eq*)
show ‹*constructive* ($\lambda x.$ $\Box a \in A \rightarrow f\ a\ x$)›
by (*subst* $*$, *rule constructive-comp-non-destructive*
[*OF Mprefix-constructive*, *of* ‹λx a. *if* $a \in A$ *then* f a x *else* *STOP*›])
(*auto intro*: *that*)
qed

lemma *Mndetprefix-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$-*simpset*]
:
‹*constructive* ($\lambda x.$ $\sqcap a \in A \rightarrow f\ a\ x$)› **if** ‹($\bigwedge a.$ $a \in A \Longrightarrow$ *non-destructive* (f a))›
proof −
have $*$: ‹$\sqcap a \in A \rightarrow f\ a\ x = \sqcap a \in A \rightarrow$ (*if* $a \in A$ *then* f a x *else* *STOP*)› **for** x
by (*auto intro*: *mono-Mndetprefix-eq*)
show ‹*constructive* ($\lambda x.$ $\sqcap a \in A \rightarrow f\ a\ x$)›
by (*subst* $*$, *rule constructive-comp-non-destructive*
[*OF Mndetprefix-constructive*, *of* ‹λx a. *if* $a \in A$ *then* f a x *else* *STOP*›])
(*auto intro*: *that*)

qed

corollary *write0-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *constructive* (λ*x*. *a* → *f x*)›
by (*simp add*: *write0-def Mprefix-restriction-shift-process*$_{ptick}$)

corollary *write-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *constructive* (λ*x*. *c***!***a* → *f x*)›
by (*simp add*: *write-def Mprefix-restriction-shift-process*$_{ptick}$)

corollary *read-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹(⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*f a*)) ⟹ *constructive* (λ*x*. *c***?***a* ∈
A → *f a x*)›
by (*simp add*: *read-def Mprefix-restriction-shift-process*$_{ptick}$ *inv-into-into*)

corollary *ndet-write-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹(⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*f a*)) ⟹ *constructive* (λ*x*. *c***!!***a* ∈
A → *f a x*)›
by (*simp add*: *ndet-write-def Mndetprefix-restriction-shift-process*$_{ptick}$
inv-into-into)

12.2 Choices

lemma *Guard-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *non-destructive* (λ*x*. *b* **&** *f x*)›
‹*constructive f* ⟹ *constructive* (λ*x*. *b* **&** *f x*)›
by (*auto intro*!: *non-destructiveI constructiveI*
simp add: *restriction-process*$_{ptick}$*-Guard dest*!: *non-destructiveD*
constructiveD)

lemma *GlobalNdet-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹(⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*f a*)) ⟹ *non-destructive* (λ*x*. ⊓*a*
∈ *A*. *f a x*)›
‹(⋀*a*. *a* ∈ *A* ⟹ *constructive* (*f a*)) ⟹ *constructive* (λ*x*. ⊓*a* ∈ *A*.
f a x)›
proof −
have ∗ : ‹⊓*a* ∈ *A*. *f a x* = ⊓*a* ∈ *A*. (*if a* ∈ *A then f a x else STOP*)›
for *x*
by (*auto intro*: *mono-GlobalNdet-eq*)

show ‹(⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*f a*)) ⟹ *non-destructive*
(λ*x*. ⊓*a* ∈ *A*. *f a x*)›
by (*subst* ∗, *rule non-destructive-comp-non-destructive*

[*OF GlobalNdet-non-destructive*, *of* ‹$\lambda x\ a.$ *if* $a \in A$ *then* $f\ a\ x$ *else STOP*›]) *auto*

show ‹$(\bigwedge a.\ a \in A \Longrightarrow$ *constructive* $(f\ a)) \Longrightarrow$ *constructive* $(\lambda x.\ \sqcap a \in A.\ f\ a\ x)$›
by (*subst* $*$, *rule non-destructive-comp-constructive*
[*OF GlobalNdet-non-destructive*, *of* ‹$\lambda x\ a.$ *if* $a \in A$ *then* $f\ a\ x$ *else STOP*›]) *auto*
qed

lemma *GlobalDet-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹$(\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(f\ a)) \Longrightarrow$ *non-destructive* $(\lambda x.\ \square a \in A.\ f\ a\ x)$›
‹$(\bigwedge a.\ a \in A \Longrightarrow$ *constructive* $(f\ a)) \Longrightarrow$ *constructive* $(\lambda x.\ \square a \in A.\ f\ a\ x)$›
proof −
have $*$: ‹$\square a \in A.\ f\ a\ x = \square a \in A.$ (*if* $a \in A$ *then* $f\ a\ x$ *else STOP*)›
for x
by (*auto intro*: *mono-GlobalDet-eq*)

show ‹$(\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(f\ a)) \Longrightarrow$ *non-destructive* $(\lambda x.\ \square a \in A.\ f\ a\ x)$›
by (*subst* $*$, *rule non-destructive-comp-non-destructive*
[*OF GlobalDet-non-destructive*, *of* ‹$\lambda x\ a.$ *if* $a \in A$ *then* $f\ a\ x$ *else STOP*›]) *auto*

show ‹$(\bigwedge a.\ a \in A \Longrightarrow$ *constructive* $(f\ a)) \Longrightarrow$ *constructive* $(\lambda x.\ \square a \in A.\ f\ a\ x)$›
by (*subst* $*$, *rule non-destructive-comp-constructive*
[*OF GlobalDet-non-destructive*, *of* ‹$\lambda x\ a.$ *if* $a \in A$ *then* $f\ a\ x$ *else STOP*›]) *auto*
qed

lemma *Ndet-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive* $f \Longrightarrow$ *non-destructive* $g \Longrightarrow$ *non-destructive* $(\lambda x.\ f\ x \sqcap g\ x)$›
‹*constructive* $f \Longrightarrow$ *constructive* $g \Longrightarrow$ *constructive* $(\lambda x.\ f\ x \sqcap g\ x)$›
by (*auto intro*!: *non-destructiveI constructiveI*
simp add: *restriction-process*$_{ptick}$*-Ndet dest*!: *non-destructiveD constructiveD*)

lemma *Det-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive* $f \Longrightarrow$ *non-destructive* $g \Longrightarrow$ *non-destructive* $(\lambda x.\ f\ x \ \square\ g\ x)$›
‹*constructive* $f \Longrightarrow$ *constructive* $g \Longrightarrow$ *constructive* $(\lambda x.\ f\ x \ \square\ g\ x)$›

by (*auto intro*!: *non-destructiveI constructiveI*
simp add: *restriction-process*$_{ptick}$*-Det dest*!: *non-destructiveD constructiveD*)

lemma *Sliding-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *non-destructive g* ⟹ *non-destructive* (λ*x*. *f x* ▷ *g x*)›
‹*constructive f* ⟹ *constructive g* ⟹ *constructive* (λ*x*. *f x* ▷ *g x*)›
by (*auto intro*!: *non-destructiveI constructiveI*
simp add: *restriction-process*$_{ptick}$*-Sliding dest*!: *non-destructiveD constructiveD*)

12.3 Renaming

lemma *Renaming-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive P* ⟹ *non-destructive* (λ*x*. *Renaming* (*P x*) *f g*)›
‹*constructive P* ⟹ *constructive* (λ*x*. *Renaming* (*P x*) *f g*)›
by (*auto intro*!: *non-destructiveI constructiveI*
simp add: *restriction-process*$_{ptick}$*-Renaming dest*!: *non-destructiveD constructiveD*)

12.4 Sequential Composition

lemma *Seq-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *non-destructive g* ⟹ *non-destructive* (λ*x*. *f x* ; *g x*)›
‹*constructive f* ⟹ *constructive g* ⟹ *constructive* (λ*x*. *f x* ; *g x*)›
by (*fact non-destructive-comp-non-destructive*[*OF Seq-non-destructive non-destructive-prod-codomain*, *simplified*])
(*fact non-destructive-comp-constructive*[*OF Seq-non-destructive constructive-prod-codomain*, *simplified*])

lemma *MultiSeq-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹(⋀*l*. *l* ∈ *set L* ⟹ *non-destructive* (*f l*)) ⟹ *non-destructive* (λ*x*. *SEQ l* ∈@ *L*. *f l x*)›
‹(⋀*l*. *l* ∈ *set L* ⟹ *constructive* (*f l*)) ⟹ *constructive* (λ*x*. *SEQ l* ∈@ *L*. *f l x*)›
by (*induct L rule*: *rev-induct*; *simp add*: *Seq-restriction-shift-process*$_{ptick}$)+

corollary *MultiSeq-non-destructive* : ‹*non-destructive* (λ*P*. *SEQ l* ∈@ *L*. *P l*)›
by (*simp add*: *MultiSeq-restriction-shift-process*$_{ptick}$(*1*)[*of L* ‹λ*l x*. *x l*›])

12.5 Synchronization Product

lemma *Sync-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *non-destructive g* ⟹ *non-destructive* (λ*x*. *f x* ⟦*S*⟧ *g x*)›
‹*constructive f* ⟹ *constructive g* ⟹ *constructive* (λ*x*. *f x* ⟦*S*⟧ *g x*)›
by (*fact non-destructive-comp-non-destructive*[*OF Sync-non-destructive non-destructive-prod-codomain*, *simplified*])
(*fact non-destructive-comp-constructive*[*OF Sync-non-destructive constructive-prod-codomain*, *simplified*])

lemma *MultiSync-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹(⋀*m*. *m* ∈# *M* ⟹ *non-destructive* (*f m*)) ⟹ *non-destructive* (λ*x*. ⟦*S*⟧ *m* ∈# *M*. *f m x*)›
‹(⋀*m*. *m* ∈# *M* ⟹ *constructive* (*f m*)) ⟹ *constructive* (λ*x*. ⟦*S*⟧ *m* ∈# *M*. *f m x*)›
by (*induct M rule*: *induct-subset-mset-empty-single*;
simp add: *Sync-restriction-shift-process*$_{ptick}$)+

corollary *MultiSync-non-destructive* : ‹*non-destructive* (λ*P*. ⟦*S*⟧ *m* ∈# *M*. *P m*)›
by (*rule MultiSync-restriction-shift-process*$_{ptick}$(*1*)[*of M* ‹λ*m x*. *x m*›]) *simp*

12.6 Throw

lemma *Throw-constructive-continuation* :
‹(⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*g a*)) ⟹ *constructive* (λ*x*. *f x* Θ *a* ∈ *A*. *g a x*)›
if ‹*constructive f*› **and** ‹⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*g a*)›
proof −
let *?g* = ‹λ*a x*. *if a* ∈ *A then g a x else STOP*›
have ∗ : ‹*f x* Θ *a* ∈ *A*. *g a x* = *f x* Θ *a* ∈ *A*. *?g a x*› **for** *x*
by (*auto intro*: *mono-Throw-eq*)
have *ndg* : ‹*non-destructive* (*?g a*)› **for** *a* **by** (*auto intro*: *that*(*2*))
show ‹*constructive* (λ*x*. *f x* Θ *a* ∈ *A*. *g a x*)›
proof (*subst* ∗, *rule constructiveI*)
fix *n* **and** *x y* **assume** ‹*x* ↓ *n* = *y* ↓ *n*›
and ‹*f x* Θ *a* ∈ *A*. *?g a x* ≠ *f y* Θ *a* ∈ *A*. *?g a y*›
have ‹(*f x* Θ *a* ∈ *A*. *?g a x*) ↓ *Suc n* = (*f y* Θ *a* ∈ *A*. *?g a x*) ↓ *Suc n*›
by (*auto intro*!: *non-destructiveD*[*OF Throw-non-destructive*,
of ‹(*f x*, λ*a*. *?g a x*)› - ‹(*f y*, λ*a*. *?g a x*)›, *simplified*]
constructiveD[*OF that*(*1*) ‹*x* ↓ *n* = *y* ↓ *n*›] *simp add*: *restriction-prod-def*)
also have ‹. . . = (*f y* Θ *a* ∈ *A*. *?g a y*) ↓ *Suc n*›

by (*auto intro*!: *constructiveD*[*OF ThrowR-constructive*]
non-destructiveD[*OF that*(*2*)] ‹$x \downarrow n = y \downarrow n$› *simp add*: *restriction-fun-def*)
finally show ‹$(f\ x\ \Theta\ a \in A.\ ?g\ a\ x) \downarrow Suc\ n = (f\ y\ \Theta\ a \in A.\ ?g\ a\ y) \downarrow Suc\ n$› **.**
qed
qed

lemma *Throw-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* $\Longrightarrow$ $(\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(g\ a)) \Longrightarrow$ *non-destructive* $(\lambda x.\ f\ x\ \Theta\ a \in A.\ g\ a\ x)$›
‹*constructive f* $\Longrightarrow$ $(\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(g\ a)) \Longrightarrow$ *constructive* $(\lambda x.\ f\ x\ \Theta\ a \in A.\ g\ a\ x)$›
proof −
let *?g* = ‹$\lambda a\ x.$ *if* $a \in A$ *then* $g\ a\ x$ *else STOP*›
have $*$: ‹$f\ x\ \Theta\ a \in A.\ g\ a\ x = f\ x\ \Theta\ a \in A.\ ?g\ a\ x$› **for** x
by (*auto intro*: *mono-Throw-eq*)

show ‹*non-destructive f* $\Longrightarrow$ $(\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(g\ a))$ $\Longrightarrow$ *non-destructive* $(\lambda x.\ f\ x\ \Theta\ a \in A.\ g\ a\ x)$›
by (*subst* $*$, *erule non-destructive-comp-non-destructive*
[*OF Throw-non-destructive non-destructive-prod-codomain*, *simplified*]) *auto*

show ‹*constructive* $(\lambda x.\ f\ x\ \Theta\ a \in A.\ g\ a\ x)$›
if ‹*constructive f*› ‹$\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(g\ a)$›
proof (*subst* $*$, *rule constructiveI*)
fix n **and** $x\ y$ **assume** ‹$x \downarrow n = y \downarrow n$›
and ‹$f\ x\ \Theta\ a \in A.\ ?g\ a\ x \neq f\ y\ \Theta\ a \in A.\ ?g\ a\ y$›
have ‹$(f\ x\ \Theta\ a \in A.\ ?g\ a\ x) \downarrow Suc\ n = (f\ y\ \Theta\ a \in A.\ ?g\ a\ x) \downarrow Suc\ n$›
by (*auto intro*!: *non-destructiveD*[*OF Throw-non-destructive*,
of ‹$(f\ x, \lambda a.\ ?g\ a\ x)$› - ‹$(f\ y, \lambda a.\ ?g\ a\ x)$›, *simplified*]
constructiveD[*OF that*(*1*) ‹$x \downarrow n = y \downarrow n$›] *simp add*: *restriction-prod-def*)
also have ‹$\ldots = (f\ y\ \Theta\ a \in A.\ ?g\ a\ y) \downarrow Suc\ n$›
by (*auto intro*!: *constructiveD*[*OF ThrowR-constructive*]
non-destructiveD[*OF that*(*2*)] ‹$x \downarrow n = y \downarrow n$› *simp add*: *restriction-fun-def*)
finally show ‹$(f\ x\ \Theta\ a \in A.\ ?g\ a\ x) \downarrow Suc\ n = (f\ y\ \Theta\ a \in A.\ ?g\ a\ y) \downarrow Suc\ n$› **.**
qed
qed

12.7 Interrupt

lemma *Interrupt-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-destructive f* ⟹ *non-destructive g* ⟹ *non-destructive* (λ*x*. *f x* △ *g x*)›
‹*constructive f* ⟹ *constructive g* ⟹ *constructive* (λ*x*. *f x* △ *g x*)›
by (*fact non-destructive-comp-non-destructive*[*OF Interrupt-non-destructive non-destructive-prod-codomain*, *simplified*])
(*fact non-destructive-comp-constructive*[*OF Interrupt-non-destructive constructive-prod-codomain*, *simplified*])

12.8 After

lemma (**in** *After*) *After-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹*non-too-destructive* Ψ ⟹ *constructive f* ⟹ *non-destructive* (λ*x*. *f x after a*)›
‹*non-too-destructive* Ψ ⟹ *non-destructive f* ⟹ *non-too-destructive* (λ*x*. *f x after a*)›
by (*auto intro*!: *non-too-destructive-comp-constructive*[*OF non-too-destructive-After*]
non-too-destructive-comp-non-destructive[*OF non-too-destructive-After*])

lemma (**in** *AfterExt*) *After*$_{tick}$*-restriction-shift-process*$_{ptick}$ [*restriction-shift-process*$_{ptick}$*-simpset*]
:
‹⟦*non-too-destructive* Ψ; *non-too-destructive* Ω; *constructive f*⟧
⟹ *non-destructive* (λ*x*. *f x after*$_{✓}$ *e*)›
‹⟦*non-too-destructive* Ψ; *non-too-destructive* Ω; *non-destructive f*⟧
⟹ *non-too-destructive* (λ*x*. *f x after*$_{✓}$ *e*)›
by (*auto intro*!: *non-too-destructive-comp-constructive*[*OF non-too-destructive-After*$_{tick}$]
non-too-destructive-comp-non-destructive[*OF non-too-destructive-After*$_{tick}$]
simp add: *non-too-destructive-fun-iff*)

12.9 Illustration

declare *restriction-shift-process*$_{ptick}$*-simpset* [*simp*]

notepad begin
fix *e f g* :: *'a* **fix** *r s* :: *'r*
fix *A B C* :: ‹*'a set*›
fix *S* :: ‹*'a* ⇒ *'a* ⇒ *'a* ⇒ *'a set*›
define *P* **where** ‹*P* ≡ υ *X*. ((□*a* ∈ *A* → *X* ⊓ *SKIP r*) △ (*f* → *STOP*))
□ (*g* → *X*)
□ ((*f* → *e* → (⊥ ⊓ (*e* → *X*))) Θ *b* ∈ *insert e B*. (*e* → *SKIP s*))›
(**is** ‹*P* ≡ υ *X*. *?f X*›)
have ‹*constructive ?f*› **by** *simp*
have ‹*cont ?f*› **by** *simp*

have ‹$P = ?f\ P$›
unfolding *P-def* **by** (*subst restriction-fix-eq*) *simp-all*

define Q **where** ‹$Q \equiv \upsilon\ X.\ (\lambda\sigma\ \sigma'\ \sigma''\ .\ e \rightarrow \sqcap\ b \in S\ \sigma\ \sigma'\ \sigma''.\ X\ b\ b\ b \sqcap SKIP\ r)$› (**is** ‹$Q \equiv \upsilon\ X.\ ?g\ X$›)
have ‹*constructive ?g*› **by** *simp*

define R **where** ‹$R \equiv \upsilon\ (x,\ y).\ (e \rightarrow y \sqcap SKIP\ r,\ \Box a \in A \rightarrow x)$›
(**is** ‹$R \equiv \upsilon\ (x,\ y).\ (?h\ y,\ ?i\ x)$›)
have ‹*snd* $R = \Box a \in A \rightarrow fst\ R$›
by (*unfold R-def*, *subst restriction-fix-eq*)
(*simp-all add*: *case-prod-beta'*)

end

Finally, we add the following new law. A similar result involving the least fixed-point operator inherited from `HOLCF` was already available: *Renaming-fix*.

lemma *Renaming-restriction-fix* :
‹*Renaming* $(\upsilon\ X.\ \varphi\ X)\ f\ g = (\upsilon\ X.\ ((\lambda P.\ Renaming\ P\ f\ g) \circ \varphi \circ (\lambda P.\ Renaming\ P\ (inv\ f)\ (inv\ g)))\ X)$›
(**is** ‹*Renaming* $(\upsilon\ X.\ \varphi\ X)\ f\ g = (\upsilon\ X.\ ?\varphi'\ X)$›) **if** ‹*constructive* φ› **and** ‹*inj* f› **and** ‹*inj* g› **for** φ :: ‹$('a,\ 'r)\ process_{ptick} \Rightarrow ('a,\ 'r)\ process_{ptick}$›
proof (*rule restriction-fix-unique*[*symmetric*])
show ‹*constructive* $?\varphi'$›
by (*simp add*: *comp-def constructiveD constructiveI*
restriction-process$_{ptick}$*-Renaming* ‹*constructive* φ›)
next
from ‹*inj* f› ‹*inj* g›
have ‹*inv* $f \circ f = id$› ‹*inv* $g \circ g = id$› **unfolding** *inj-iff* **.**
thus ‹$?\varphi'\ (Renaming\ (\upsilon\ X.\ \varphi\ X)\ f\ g) = Renaming\ (\upsilon\ X.\ \varphi\ X)\ f\ g$›
by (*simp add*: *Renaming-id flip*: *Renaming-comp restriction-fix-eq*[*OF* ‹*constructive* φ›])
qed

end

Part I
Instantiation of the Generic Fixrec Package for Restriction-Spaces

theory *GenFixrec−RS*
imports *HOL−CSP.GenFixrec*
HOL−CSP-RS

begin

default-sort *type* — no cpo default sorts are imposed on users of the instance

12.10 Example setup

lemma *def-constructive-fix-eq* :
‹$f \equiv (\upsilon\ x.\ F\ x) \Longrightarrow$ *constructive* $F \Longrightarrow f = F\ f$›
by(*simp*, *rule restriction-fix-eq*,*simp*)

Case distinctions on conditions not depending on the recursion variable (the rules of `Restriction_Spaces` state the condition as $P\ x$, which the simplifier cannot match):

lemma *non-destructive-if* [*simp*] :
‹($b \Longrightarrow$ *non-destructive* f)
$\Longrightarrow (\neg\ b \Longrightarrow$ *non-destructive* g)
$\Longrightarrow$ *non-destructive* $(\lambda x.$ *if* b *then* $f\ x$ *else* $g\ x)$›
by (*cases* b) *simp-all*

lemma *constructive-if* [*simp*] :
‹($b \Longrightarrow$ *constructive* f)
$\Longrightarrow (\neg\ b \Longrightarrow$ *constructive* g)
$\Longrightarrow$ *constructive* $(\lambda x.$ *if* b *then* $f\ x$ *else* $g\ x)$›
by (*cases* b) *simp-all*

The rules on the constructiveness of *Throw*, in particular in its continuation (*ThrowR-constructive*, *Throw-constructive-continuation*), are in *HOL−CSP-RS.Throw-Non-Destructive*; the constructiveness proofs of the RS instance use:

declare *Throw-constructive-continuation* [*simp*]

lemma *def-restriction-fix-ind* :
$[\![f \equiv \upsilon\ x.\ F\ x;$ *constructive* F; $adm_{\downarrow}\ P$; $P\ x$; $\bigwedge x.\ P\ x \Longrightarrow P\ (F\ x)]\!]$
$\Longrightarrow P\ f$

by (*simp add*: *restriction-fix-ind*)

ML‹
structure RS-instance =
struct

local open GenFixrec in

val E = @{*term restriction-fix*};
val a-ty = (*snd o dest-funT o fastype-of*) *E*;
fun mk-rs-fix [] *E* = *let val a-ty* = (*fst o dest-funT o fastype-of*) *E*
in Const (***const-name*** ‹*complete-restriction-space-class.restriction-fix*›,

(*a-ty* −−> *a-ty*) −−> *a-ty*)
$ *E*
end

fun dest-rs-fix (*Const* (***const-name*** ‹*complete-restriction-space-class.restriction-fix*›,
t) $ *E*) = *E*

(∗ *constructiveness proofs*: *the shift rules of HOL−CSP-RS are simp rules*; *case-prod-beta turns*
continuations with tuple patterns (*e.g. of read prefixes*) *into terms they apply to* ∗)
fun set-context-fix ctxt = *ctxt* |> *fold Splitter.add-split* @{*thms prod.split*}
|> *fold Simplifier.add-simp* [@{*thm case-prod-beta*}]

val def-rs-fix-eq = @{*thm def-constructive-fix-eq*}

val aux = (*Goal.prove-global* @{*theory*} [] []
(@{*prop* ‹*s* ≡ *s*′ ⟹ *s*′ = *t* ⟹ *s* = *t*›})
(*fn* {*context*,...} =>*auto-tac context*))

val aux2 = (*Goal.prove-global* @{*theory*} [] []
(@{*prop* ‹*s* = *s*′ ⟹ *P s*′ = *t* ⟹ *P s* = *t*›})
(*fn* {*context*,...} =>*auto-tac context*))

val fst-conv = @{*thm fst-conv*}
val snd-conv = @{*thm snd-conv*}

val add-prod-splits = *fold Splitter.add-split* @{*thms prod.split*};
fun metis-step ctxt defs = *Metis-Tactic.metis-tac* [] *combs*
ctxt (*fst-conv* :: *snd-conv* :: *defs*)

val aux2′ = *meta-eq-to-obj-eq RS aux2*

(∗ *structural projection proof* (*core*), *heuristic proof as fallback*;

```
  aux2' (instead of aux) also covers shifted equations  c x1 ... xn = E
*)
fun prove-projection  defs unfold ctxt def =
      structural-projection-tac defs unfold ctxt def
      ORELSE
      EVERY [resolve0-tac [def RS aux2'] 1,
            resolve0-tac [unfold RS aux2] 1,
            auto-tac (add-prod-splits ctxt),
             TRY (metis-step ctxt defs 1)]  (* auto may already solve
the goal *)

local open HOLogic in

fun substitute-curried-pred ctxt vartab thm =
    let val frees = map (fn((s,t,-),-) => Free(Binding.name-of s,t))
vartab;
        val vars-proj = map (fn((s,t,-),-) => Var((Binding.name-of
s,0),t)) vartab;
       val Var ((P, idx), ty') $ S = dest-Trueprop(Thm.concl-of thm);
       val prodT'= strip-tupleT (domain-type ty')
      val frees'= map (fn (Free(s,-),t) => Free(s,t)) (frees ~~ prodT')

       val frees'' = map (fn (Free(s,-),t) => Var((s^0,0),t)) (frees ~~
prodT')
       val tuple-witness = HOLogic.mk-tuple frees''

       val P' = list-comb (Var((P^',idx),prodT' ---> boolT), frees')
       val projT' =  tupled-lambda (mk-tuple frees') P'
      val subst = Vars.make2(((P,idx),ty'), Thm.cterm-of ctxt projT')
                             (((x,0),domain-type ty'), Thm.cterm-of ctxt
tuple-witness);
   in  thm |> Thm.instantiate(TVars.empty,subst) end

end

val def-restriction-fix-ind = @{thm def-restriction-fix-ind}

fun add-case-prod-beta ctxt = Simplifier.add-simp @{thm case-prod-beta}
ctxt (* case-prod-beta*)

val rs =    {mk-fix    = mk-rs-fix,
            dest-fix  = dest-rs-fix,
           lambda'   = HOLogic.tupled-lambda, (* no inner application;
never used *)
            apply     = Bound 0, (* dummy *)
            shift-tac = (fn {context,...} => all-tac), (* dummy *)
            fix-tac   = (fn {context,prems} =>
                          resolve0-tac [hd prems RS def-rs-fix-eq] 1
                                THEN (SOLVED' (asm-full-simp-tac
```

```
(set-context-fix context)) 1
                                        ORELSE auto-tac (set-context-fix
context))),
          proj-tac  = (fn {proj-def, projs, unfold} =>
                         fn {context,prems} => prove-projection projs
unfold context proj-def),
          fixind-tac = (fn {eqns, varstab, projs, rec-def, unfold} =>
                                fn {context,prems} => ((rec-def RS
def-restriction-fix-ind)
                                                    |> substitute-curried-pred
context varstab
                                                |> Simplifier.asm-full-simplify
                                                        (add-case-prod-beta
context)
                                                    |> Local-Defs.fold context
projs
                                                |> Simplifier.asm-full-simplify
context
                                                    ))
        } : language;

fun define-rs-space thy =  Context.Theory thy
                       |> update-language (Binding.make(RS,@{here}))
rs
                           |> Context.the-theory;
end (* local *)
end (* struct *)
›
```

setup ‹*RS-instance.define-rs-space*›

end

Part II
Conceptual Tests: the Restriction-Space Instance of `Fixrec`

theory *Fixrec-RS*
  imports *GenFixrec−RS*
begin

The tests are organised by concept and mirror the theory `Fixrec_HOLCF`. Expected failures are checked by *expect-failure*, which runs the

command semantics on the current theory and asserts that it fails with a message containing a given fragment.

ML ‹
fun expect-failure langs vars eqns fragment =
let val spec = (map (fn (n, T) => (Binding.name n, SOME T, NoSyn)) vars,
map (fn (n, e) => (false, ((Binding.name n, []), e))) eqns)
val msg-of = Protocol-Message.clean-output o Runtime.exn-message
in case Exn.result (GenFixrec.add-fixrec-langs-global (map (rpair Position.none) langs) spec)
(Context.the-global-context ()) of
Exn.Res - => error (expected failure, but Fixrec succeeded: ^ commas langs)
| Exn.Exn exn => if String.isSubstring fragment (msg-of exn) then ()
else error (unexpected failure message:\n ^ msg-of exn)
end
›

13 Language Selection

A language that always fails to prove the fixpoint unfolding; used to test fallback.

setup ‹*fn thy => Context.Theory thy*
*|> GenFixrec.update-language (Binding.make (BROKEN, **here**))*
{mk-fix = #mk-fix RS-instance.rs, dest-fix = #dest-fix RS-instance.rs, lambda' = #lambda' RS-instance.rs,
apply = #apply RS-instance.rs, shift-tac = #shift-tac RS-instance.rs, fix-tac = K no-tac,
proj-tac = #proj-tac RS-instance.rs, fixind-tac = #fixind-tac RS-instance.rs}
|> Context.the-theory›

Explicit selection:

Fixrec [*RS*] *l2* :: ‹*int process*› **where** *l2-eq* : ‹*l2 = 2 → l2*›

Languages are tried from left to right; the first one that succeeds is taken:

Fixrec [*BROKEN*, *RS*] *l3* :: ‹*int process*› **where** *l3-eq* : ‹*l3 = 3 → l3*›

ML ‹
val proc = int process
(∗ *the default language HOLCF is not available in this instance* ∗)
val - = expect-failure [*HOLCF*] [(*e0*, *proc*)] [(*e0-eq*, *e0 = 1 → e0*)]

Undefined Fixrec language: *HOLCF*\
(∗ *all languages fail*: *every failure is reported* ∗)
val - = expect-failure [*BROKEN*] [(*e1*, *proc*)] [(*e1-eq*, *e1 = 1 → e1*)]
no language succeeded
(∗ *unknown and misspelled* (*case−sensitive*) *language names* ∗)
val - = expect-failure [*FOO*] [(*e2*, *proc*)] [(*e2-eq*, *e2 = 1 → e2*)]
Undefined Fixrec language: *FOO*\
val - = expect-failure [*rs*] [(*e3*, *proc*)] [(*e3-eq*, *e3 = 1 → e3*)]
Undefined Fixrec language: *rs*\
val - = expect-failure [] [(*e4*, *proc*)] [(*e4-eq*, *e4 = 1 → e4*)]
no language given
›

14 Unshifted Equations

Fixrec [*RS*] *u1* :: ‹*int* ⇒ *int process*› **where** *u1-eq* : ‹*u1* = (λ*x*. *x* → *u1* (*x*+*1*))›

15 Generic Shift over HOL Application

The shift over HOL application is provided by the core, independently of the language; no continuous function space is required.

Fixrec [*RS*] *s1* :: ‹*int* ⇒ *int process*› **where** *s1-eq* : ‹*s1 x* = *x* → *s1* (*x*+*1*)›
Fixrec [*RS*] *s2* :: ‹*int* ⇒ *nat* ⇒ *int process*› **where** *s2-eq* : ‹*s2 x n* = *x* → *s2* (*x*+*1*) (*Suc n*)›
Fixrec [*RS*] *s3* :: ‹*int* × *nat* ⇒ *int process*› **where** *s3-eq* : ‹*s3* (*x*,*n*) = *x* → *s3* (*x*+*1*,*n*)›
Fixrec [*RS*] *s4* :: ‹*int* × *int* × *int* ⇒ *int process*› **where** *s4-eq* : ‹*s4* (*x*,*y*,*z*) = *x* → *s4* (*y*,*z*,*x*)›
Fixrec [*RS*] *s5* :: ‹*int* × *nat* ⇒ *bool* × *int* ⇒ *int process*›
where *s5-eq* : ‹*s5* (*x*,*n*) (*b*,*y*) = *x* → *s5* (*y*,*Suc n*) (¬*b*,*x*)›

The shift is equivalent to the unshifted form:

lemma ‹*s1* = (λ*x*. *x* → *s1* (*x*+*1*))› **using** *s1-eq* **by** *blast*

ML ‹
val - = expect-failure [*RS*] [(*e5*, *int* ⇒ *int process*)] [(*e5-eq*, *e5 0 = 0* → *e5 0*)]
args must be free variables or tuples of them
val - = expect-failure [*RS*] [(*e6*, *int* ⇒ *int process*)]
[(*e6-eq*, (*if True then e6 else e6*) *x* = *x* → *e6 x*)]
lhs head must be one of the declared constants
›

16 Domain-Specific Shift

The restriction-space instance has no application/abstraction pair of its own; a lhs using the HOLCF application · is not recognised:

ML ‹
val - = expect-failure [*RS*] [(*e7*, *int process* → *int process*)] [(*e7-eq*, *e7*·*P* = *1* → *e7*·*P*)]
lhs head must be one of the declared constants
›

17 Mutual Recursion

Fixrec [*RS*] *m1* :: ‹*int* ⇒ *int process*› **and** *m2* :: ‹*int* ⇒ *int process*›
where *m1-eq* : ‹*m1 x* = *1* → *m2 x*›
| *m2-eq* : ‹*m2 x* = *2* → *m1* (*x*+*1*)›

Fixrec [*RS*] *m3* :: ‹*int* ⇒ *int process*› **and** *m4* :: ‹*int process*›
where *m3-eq* : ‹*m3 x* = *x* → *m4*›
| *m4-eq* : ‹*m4* = *0* → *m3 1*›

Equations that are not (self-)recursive, alone or within a system:

Fixrec [*RS*] *n1* :: ‹*int* ⇒ *int process*› **where** *n1-eq* : ‹*n1 x* = *x* → *Skip*›
Fixrec [*RS*] *n2* :: ‹*int* ⇒ *int process*› **and** *n3* :: ‹*int* ⇒ *int process*›
where *n2-eq* : ‹*n2 x* = *x* → *n3 x*›
| *n3-eq* : ‹*n3 x* = *x* → *n3* (*x*+*1*)›

A larger mutual system (guarded equations, choices, finite nondeterminism, a member that is not self-recursive):

Fixrec [*RS*] *t1* :: ‹*int* ⇒ *int process*› **and** *t2* :: ‹*int* ⇒ *int process*›
and *t3* :: ‹*int* ⇒ *int process*› **and** *t4* :: ‹*int process*› **and** *t5* :: ‹*int* ⇒ *int process*›
where *t1-eq* : ‹*t1 x* = *1* → *t2 x*›
| *t2-eq* : ‹*t2 x* = (*2* → *t3 x*) □ (*3* → *t1* (*x*+*1*))›
| *t3-eq* : ‹*t3 x* = (⊓ *a* ∈ {*0*..*x*}. *a* → *t4*)›
| *t4-eq* : ‹*t4* = *0* → *t5 1*›
| *t5-eq* : ‹*t5 x* = (*x* → *t1 x*) ⊓ (*x* → *t4*)›

Recursion through the continuation of the exception operator Θ (FDR's [| *A* |>), also when the exception event is initial (as in the crash handling of Dali):

Fixrec [*RS*] *cr* :: ‹*int process*› **and** *cr′* :: ‹*int process*›
where *cr-eq* : ‹*cr* = ((*1* → *Skip*) ||| (*0* → *Skip*)) Θ *a* ∈ {*0*}. *cr′*›
| *cr′-eq* : ‹*cr′* = ((*2* → *Skip*) ||| (*0* → *Skip*)) Θ *a* ∈ {*0*}. *cr′*›

Recursion in both operands of Θ (guarded in the left one), with a parametrised continuation:

Fixrec [*RS*] *tb* :: ‹*int* ⇒ *int process*›
  where *tb-eq* : ‹*tb x* = (*x* → *tb* (*x*+*1*) ||| (*0* → *Skip*)) Θ *a* ∈ {*0*}. *tb* (*x*−*1*)›

18 Polymorphism

Fixrec [*RS*] *p1* :: ‹'*a* ⇒ '*a process*› **where** *p1-eq* : ‹*p1 x* = *x* → *p1 x*›

Fixrec [*RS*] *p2* :: ‹'*a* ⇒ '*a process*› **and** *p3* :: ‹'*a* ⇒ '*a process*›
  where *p2-eq* : ‹*p2 x* = *x* → *p3 x*›
  | *p3-eq* : ‹*p3 x* = *x* → *p2 x*›

lemma ‹*p1 x* = *x* → *p1 x*› **by** (*fact p1-eq*)

Type variables are of sort *type*, not *cpo*/*domain*:

lemma ‹*p1* (*n*::*nat*) = *n* → *p1 n*› **by** (*fact p1-eq*)

19 Generated Theorems

thm *m1-m2-rec-def m1-m2-rec-unfold m1-def m2-def m1-eq m2-eq m1-m2-induct*

lemma ‹*s5* (*x*,*n*) (*b*,*y*) = *x* → *s5* (*y*,*Suc n*) (¬*b*,*x*)› **by** (*fact s5-eq*)

Fixpoint induction over the restriction fixpoint: admissibility, an arbitrary start value, step.

ML ‹
val - = @{*assert*} (*Thm.nprems-of* @{*thm s1-induct*} = *3*)
val - = @{*assert*} (*Thm.nprems-of* @{*thm m1-m2-induct*} = *3*)
›

Unlike HOLCF's `fixrec`, the equations are *not* simp rules by default (unguarded use of recursive equations by the simplifier loops); [*simp*] declares them explicitly:

Fixrec [*RS*] *sa* :: ‹*int* ⇒ *int process*› **where** *sa-eq* : ‹*sa x* = *x* → *Skip*›
Fixrec [*RS*] *sb* :: ‹*int* ⇒ *int process*› **where** *sb-eq* [*simp*] : ‹*sb x* = *x* → *Skip*›

lemma ‹*sb 1* = *1* → *Skip*› **by** *simp*

ML ‹
val - = @{*assert*} (*is-none* (*try* (*Goal.prove* ***context*** [] [] @{*prop* ‹*sa 1* = *1* → *Skip*›})
                                (*fn* {*context*, ...} => *asm-full-simp-tac context 1*)))
›

20 Semantic Properties of the Instance

In restriction spaces, the fixpoint exists only for constructive (guarded) equations; an unguarded equation is rejected:

ML ‹
val - = expect-failure [RS] [(d1, int ⇒ int process)] [(d1-eq, d1 x = d1 (x+1))]
constructive
›

21 Local Contexts: Locales, Contexts, Interpretation

`Fixrec` is a local-theory command: it can be used inside locales and other local contexts. Locale parameters may occur in the equations; the generated constants are abstracted over the parameters they depend on, and all theorems are available under interpretation.

locale *L1* = **fixes** *k* :: *int* **assumes** *k-pos* : ‹*0 < k*›
begin
Fixrec [*RS*] *r1* :: ‹*int ⇒ int process*› **where** *r1-eq* : ‹*r1 x = x → r1 (x + k)*›
Fixrec [*RS*] *r2* :: ‹*int ⇒ int process*› **and** *r3* :: ‹*int process*›
  where *r2-eq* : ‹*r2 x = x → r3*›
  | *r3-eq* : ‹*r3 = k → r2 k*›
lemma ‹*r1 x = x → r1 (x + k)*› **by** (*fact r1-eq*)
thm *r1-rec-def r1-rec-unfold r1-def r1-induct r2-r3-induct*
end

The global constants take the parameter as argument:

term ‹*L1.r1* :: *int ⇒ int ⇒ int process*›

Re-entering the locale:

context *L1* **begin**
lemma ‹*r3 = k → r2 k*› **by** (*fact r3-eq*)
end

Interpretation:

interpretation *L1-5* : *L1 5* **by** *standard simp*
lemma ‹*L1-5.r1 x = x → L1-5.r1 (x + 5)*› **by** (*fact L1-5.r1-eq*)

Unnamed local contexts:

context fixes *m* :: *int* **begin**
Fixrec [*RS*] *r4* :: ‹*int process*› **where** *r4-eq* : ‹*r4 = m → r4*›
end

term ‹*r4* :: *int* ⇒ *int process*›

Polymorphic locales:

locale *L3* = **fixes** *e* :: ‹'*a*›
begin
Fixrec [*RS*] *r5* :: ‹'*a process*› **where** *r5-eq* : ‹*r5* = *e* → *r5*›
end

In restriction spaces, ⊓ *a* ∈ *A*. - preserves constructiveness for arbitrary *A*: no finiteness assumption is needed (in contrast to the HOLCF instance).

locale *L2* = **fixes** *A* :: ‹*int set*›
begin
Fixrec [*RS*] *r6* :: ‹*int process*› **where** *r6-eq* : ‹*r6* = (⊓ *a* ∈ *A*. *a* → *r6*)›
end

Part III
Tests Moved from the Instance Theory (verbatim)

22 Test

Fixrec [*RS*] *g* :: ‹'*a* ⇒ *int process*›
  and *h* :: ‹'*a* ⇒ *int process*›
  and *j* :: ‹'*a* ⇒ *int process*›
 where *a* : ‹*g* = (λ *x*. □ *a* ∈ *UNIV* → *g x*)›
 | *b* : ‹*h* = (λ *x*. *4* → *j x*)›
 | *c* : ‹*j* = (λ *x*. *5* → *h x*)›

term *g* **term** *h* **term** *j*

thm *g-h-j-rec-def g-h-j-rec-unfold g-def h-def j-def a b c g-h-j-induct*

lemma *g-h-j-induct'* :
 assumes ∗ : ‹*adm*↓ (λ(*g*::'*a* ⇒ *int process*, *h*::'*a* ⇒ *int process*, *j*::'*a* ⇒ *int process*). *P' g h j*)›
  and ∗∗: ‹*P' g*0 *h*0 *j*0›
  and ∗∗∗: (⋀*g*0 *h*0 *j*0. *P' g*0 *h*0 *j*0 ⟹ *P'* (λ*x*. □ *a* ∈ *UNIV* → *g*0 *x*)
                                                (λ *x*. *4* → *j*0 *x*)(λ *x*. *5* → *h*0 *x*))
 shows‹*P' g h j*›

proof (*rule def-restriction-fix-ind*[*OF g-h-j-rec-def*,
where $P = \langle\lambda(g,\ h,\ j).\ P'\ g\ h\ j\rangle$
and $x = \langle(g_0,h_0,j_0)\rangle$,*simplified*, *simplified case-prod-beta*,
folded g-def h-def j-def, *simplified*])
show $\langle adm_{\downarrow}\ (\lambda(x,\ xa,\ y).\ P'\ x\ xa\ y)\rangle$ **by** *fact*
next
show $\langle P'\ g_0\ h_0\ j_0\rangle$ **by** *fact*
next
from $***$ **show** $\langle P'\ (fst\ x)\ (fst\ (snd\ x))\ (snd\ (snd\ x)) \Longrightarrow$
$P'\ (\lambda xa.\ \square a{\in}UNIV \rightarrow fst\ x\ xa)$
$(\lambda xa.\ 4 \rightarrow snd\ (snd\ x)\ xa)$
$(\lambda xa.\ 5 \rightarrow fst\ (snd\ x)\ xa)\rangle$ **for** x **by** *simp*
qed

23 Simulation and Tests

ML‹
val t = @{*term* ‹g_0›}

val s = $g\hat{}_0$
›

lemma *aux* : $s \equiv s' \Longrightarrow s' = t \Longrightarrow s = t$ **by** *auto*
lemma *aux2*: $s = s' \Longrightarrow P\ s' = t \Longrightarrow P\ s = t$ **by** *auto*

lemma a' : $\langle g = (\lambda x.\ \square\ a \in UNIV \rightarrow g\ x)\rangle$
thm *g-def*[*THEN aux*]
apply (*rule g-def*[*THEN aux*])
thm *g-h-j-rec-unfold*[*THEN aux2*]
apply (*rule g-h-j-rec-unfold*[*THEN aux2*])
apply (*auto split*: *prod.split*)
by (*metis g-def fst-conv snd-conv*)

lemma a'' : $\langle g = (\lambda x.\ \square\ a \in UNIV \rightarrow g\ x)\rangle$
by(*tactic* ‹*RS-instance.prove-projection* [@{*thm g-def*},@{*thm h-def*},@{*thm j-def*}]
@{*thm g-h-j-rec-unfold*}
@{*context*}
@{*thm g-def*}›)

lemma b'' : $\langle h = (\lambda x.\ (4\ ::\ int) \rightarrow j\ x)\rangle$
by(*tactic* ‹*RS-instance.prove-projection* [@{*thm g-def*},@{*thm h-def*},@{*thm j-def*}]
@{*thm g-h-j-rec-unfold*}
@{*context*}

@{*thm h-def*}›)

lemma *c''* : ‹*j* = (λ *x*. *5* → *h* *x*)›
by(*tactic* ‹*RS-instance.prove-projection* [@{*thm g-def*},@{*thm h-def*},@{*thm j-def*}]
@{*thm g-h-j-rec-unfold*}
@{*context*}
@{*thm j-def*}›)

lemma *restriction-fix-eq-defined* : ‹*P* = (υ *X*. *f X*) ⟹ *constructive f* ⟹ *P* = *f P*›
using *restriction-fix-eq* **by** *blast*

definition *g-h-j'* :: ‹('*b* ⇒ *nat process*) × ('*b* ⇒ *nat process*) × ('*b* ⇒ *nat process*)›
where ‹*g-h-j'* ≡ υ *x*. *case x of*
(*g'*, *h'*, *j'*) ⇒ (λ*x*. (*3* :: *nat*) → *g' x*,
λ *x*. (*4* :: *nat*) → *j' x*, λ *x*. *5* → *h' x*)›

definition *g-h-j* **where** ‹*g-h-j* ≡ υ (*g*, *h*, *j*). (λ *x*. □ *a* ∈ *UNIV* → *g x*,
λ *x*. *4* → *j* *x*,
λ *x*. *5* → *h* *x*)›

lemma *g-h-j'-unfold* : ‹*g-h-j'* = (*case g-h-j' of*
(*g'*, *h'*, *j'*) ⇒ (λ *x*. *3* → *g' x*,
λ *x*. *4* → *j' x*,
λ *x*. *5* → *h' x*))›
apply (*rule restriction-fix-eq-defined*)
apply (*fact g-h-j'-def*[*THEN meta-eq-to-obj-eq*])
by *auto*

lemma [*simp*] : ‹*constructive f* ⟹ *non-destructive* (λ*x*. *f* (*x*, *y*))›
by (*simp add*: *constructive-prod-domain-iff*)

end

Part IV
Annex: Running Example with Buffer over infinite Alphabet

This is a variant of the classical Copy-Buffer example. The processes that do not use hiding are defined in the restriction-space instance of `Fixrec`, i.e., as unique fixpoints of constructive functionals. Since these functionals are also continuous, the unique fixpoints coincide with the least fixpoints of `HOLCF`. The proofs involving hiding (which is not non-destructive, hence has no place in restriction spaces) are then done in the HOLCF setting, exactly as in the standard example.

24 Defining the Copy-Buffer Example

datatype *'a channel = left 'a | right 'a | mid 'a | ack*

definition *SYN* :: ‹*'a channel set*›
  where ‹*SYN* ≡ *range mid* ∪ {*ack*}›

Fixrec [*RS*] *COPY* :: ‹*'a channel process*› **where** *COPY-rec*: ‹*COPY* = *left*?*x* → *right*!*x* → *COPY*›
Fixrec [*RS*] *SEND* :: ‹*'a channel process*› **where** *SEND-rec*: ‹*SEND* = *left*?*x* → *mid*!*x* → *ack* → *SEND*›
Fixrec [*RS*] *REC* :: ‹*'a channel process*› **where** *REC-rec*: ‹*REC* = *mid*?*x* → *right*!*x* → *ack* → *REC*›

definition *SYSTEM* :: ‹*'a channel process*›
  where ‹*SYSTEM* ≡ *SEND* ⟦ *SYN* ⟧ *REC* \ *SYN*›

25 Unique Fixpoints and Least Fixpoints

For a functional that is constructive and continuous, the unique fixpoint (restriction spaces) and the least fixpoint (HOLCF) coincide: both are fixpoints, and a constructive functional has only one.

lemma *constructive-cont-fix-unique* :
  ‹*X* = (*μ x*. *F x*)› **if** ‹*constructive F*› ‹*cont F*› ‹*X* = *F X*›
  for *F* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$›
proof −

have ‹(∪ x. F x) = X› **by** (*rule restriction-fix-unique*[*OF that*(*1*)])
(*simp add*: *that*(*3*)[*symmetric*])
moreover have ‹(∪ x. F x) = (μ x. F x)›
by (*rule restriction-fix-unique*[*OF that*(*1*)]) (*use cont-process-rec*[*OF refl that*(*2*)] **in** *simp*)
ultimately show *?thesis* **by** *simp*
qed

lemma *COPY-is-fix* : ‹*COPY* = (μ *COPY*. *left*?*x* → *right*!*x* → *COPY*)›
by (*rule constructive-cont-fix-unique*) (*simp*, *simp*, *fact COPY-rec*)

lemma *SEND-is-fix* : ‹*SEND* = (μ *SEND*. *left*?*x* → *mid*!*x* → *ack* → *SEND*)›
by (*rule constructive-cont-fix-unique*) (*simp*, *simp*, *fact SEND-rec*)

lemma *REC-is-fix* : ‹*REC* = (μ *REC*. *mid*?*x* → *right*!*x* → *ack* → *REC*)›
by (*rule constructive-cont-fix-unique*) (*simp*, *simp*, *fact REC-rec*)

26 The Standard Proof

26.1 Channels and Synchronization Sets

First part: abstract properties for these events to SYN. This kind of stuff could be automated easily by some extra-syntax for channels and SYN-sets.

lemma *simplification-lemmas* [*simp*] :
‹*range left* ∩ *SYN* = {}› ‹*range right* ∩ *SYN* = {}› ‹*ack* ∈ *SYN*› ‹*range mid* ⊆ *SYN*›
‹*mid x* ∈ *SYN*› ‹*right x* ∉ *SYN*› ‹*left x* ∉ *SYN*› ‹*inj mid*›
by (*auto simp*: *SYN-def inj-on-def*)

lemma ‹*finite* (*SYN*::'*a channel set*) ⟹ *finite* {(*t*::'*a*). *True*}›
by (*metis* (*no-types*) *SYN-def UNIV-def channel.inject*(*3*) *finite-Un finite-imageD inj-on-def*)

26.2 Setup for Rewriting

The recursive equations *COPY-rec*, *SEND-rec*, *REC-rec* are generated by `Fixrec`.

lemmas *Sync-rules* = *read-Sync-read-subset-forced-read-same-chan*
read-Sync-read-left read-Sync-read-right
write-Sync-read-left write-Sync-read-right
read-Sync-write-left read-Sync-write-right
write-Sync-write-subset
write-Sync-read-subset read-Sync-write-subset

write0-Sync-write-right write0-Sync-write0

lemmas *Hiding-rules = Hiding-read-disjoint Hiding-write-subset Hiding-write-disjoint*
Hiding-write0-non-disjoint Hiding-write0-disjoint

lemmas *mono-rules = mono-read-FD mono-write-FD mono-write0-FD*

26.3 Various Samples of Refinement Proofs

An example for a very explicit structured proof. Slow-motion for presentations. Note that the proof makes no assumption over the structure of the content of the channel; it is truly polymorphic wrt. the channel type $'\alpha$.

lemma *impl-refines-spec'* : (*COPY* :: *'a channel process*) $\sqsubseteq_{FD}$ *SYSTEM*
unfolding *SYSTEM-def COPY-is-fix*
proof (*rule fix-ind*)
show *adm* ($\lambda a.$ $a \sqsubseteq_{FD}$ (*SEND* $[\![SYN]\!]$ *REC* $\setminus$ *SYN*))
by (*simp add*: *cont2mono*)
next
show $\bot \sqsubseteq_{FD}$ (*SEND* $[\![SYN]\!]$ *REC* $\setminus$ *SYN*)
by *simp*
next
fix *x* :: *'a channel process*
assume *hyp* : $x \sqsubseteq_{FD}$ (*SEND* $[\![SYN]\!]$ *REC* $\setminus$ *SYN*)
show (Λ *x*. *left*?*xa* $\rightarrow$ *right*!*xa* $\rightarrow$ *x*)·*x* $\sqsubseteq_{FD}$ (*SEND* $[\![SYN]\!]$ *REC* $\setminus$ *SYN*)
apply (*subst SEND-rec*, *subst REC-rec*)
apply (*simp add*: *cont-fun*)
proof −
have *left*?*x* $\rightarrow$ *mid*!*x* $\rightarrow$ *ack* $\rightarrow$ *SEND* $[\![SYN]\!]$ *mid*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ *ack* $\rightarrow$ *REC* $\setminus$ *SYN*
= *left*?*x* $\rightarrow$ (*mid*!*x* $\rightarrow$ *ack* $\rightarrow$ *SEND* $[\![SYN]\!]$ *mid*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ *ack* $\rightarrow$ *REC* $\setminus$ *SYN*)
(**is** *?lhs* = -)
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* $\rightarrow$ (*mid*!*x* $\rightarrow$ (*ack* $\rightarrow$ *SEND* $[\![SYN]\!]$ *right*!*x* $\rightarrow$ *ack* $\rightarrow$ *REC*) $\setminus$ *SYN*)
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* $\rightarrow$ (*ack* $\rightarrow$ *SEND* $[\![SYN]\!]$ *right*!*x* $\rightarrow$ *ack* $\rightarrow$ *REC*) $\setminus$ *SYN*
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ (*ack* $\rightarrow$ *SEND* $[\![SYN]\!]$ *ack* $\rightarrow$ *REC*) $\setminus$ *SYN*
by (*simp add*: *Sync-rules Hiding-rules*)
also have ... = *left*?*x* $\rightarrow$ *right*!*x* $\rightarrow$ (*SEND* $[\![SYN]\!]$ *REC* $\setminus$ *SYN*) (**is** - = *?rhs*)

```
    by (simp add: Sync-rules Hiding-rules)
  finally have * :  ?lhs = ?rhs .
  show     left?xa → right!xa → x  ⊑FD  ?lhs
    by (simp only : * mono-read-FD mono-write-FD hyp)
 qed
qed
```

26.4 An Example for a Highly Automated Proof.

Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

```
lemma impl-refines-spec : COPY ⊑FD SYSTEM
 unfolding SYSTEM-def COPY-is-fix
 apply(rule fix-ind, auto intro: le-FD-adm simp:  cont-fun monofunI)
 apply(subst SEND-rec, subst REC-rec)
 apply (simp add: Sync-rules Hiding-rules mono-read-FD mono-write-FD)

 oops


lemma spec-refines-impl :
 assumes fin: finite (SYN:: 'a channel set)
 shows       SYSTEM ⊑FD (COPY :: 'a channel process)
 apply(simp add: SYSTEM-def SEND-is-fix)
 apply(rule fix-ind, simp-all)
  apply (intro le-FD-adm)
   apply (simp add: fin)
  apply (simp add: cont2mono)
 apply (simp add: Sync-commute)
 apply (subst COPY-rec, subst REC-rec)
 apply (simp add: read-def write-def comp-def)
 apply (subst Mprefix-Sync-Mprefix-right) apply auto[2]
 apply (simp add: Sync-rules Hiding-rules comp-def)
 apply (subst Hiding-Mprefix-disjoint) apply auto[1]
 apply (intro mono-Mprefix-FD ballI)
 apply (subst Mprefix-Sync-Mprefix-subset) apply auto[2]
 apply (subst Hiding-Mprefix-non-disjoint) apply auto[1]
 apply simp
 apply (fold write0-def)
 by (simp add: Hiding-write0-disjoint Hiding-write0-non-disjoint Sync-commute
mono-write0-FD write0-Sync-write0)
```

Note that this was actually proven for the Process ordering, not the refinement ordering. But the former implies the latter. And due to anti-symmetry, equality follows for the case of finite alphabets ...

```
lemma spec-equal-impl :
 assumes fin:  finite (SYN::('a channel)set)
```

shows $SYSTEM = (COPY::'a\ channel\ process)$
by (*simp add*: *FD-antisym fin impl-refines-spec' spec-refines-impl*)

26.5 Deadlock Freeness Proof

HOL-CSP can be used to prove deadlock-freeness of processes with infinite alphabet. In the case of the $COPY$ - process, this can be formulated as the following refinement problem:

lemma *DF-COPY* : $(DF\ (range\ left \cup range\ right)) \sqsubseteq_{FD} COPY$
apply(*simp add*:*DF-def*,*rule fix-ind2*)
proof −
show *adm* $(\lambda a.\ a \sqsubseteq_{FD} COPY)$ **by**(*rule le-FD-adm*, *simp-all add*: *monofunI*)
next
show $\bot \sqsubseteq_{FD} COPY$ **by** *fastforce*
next
have *1*: $(\sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow \bot) \sqsubseteq_{FD} (\sqcap xa{\in}\ range\ left \rightarrow \bot)$
by (*simp add*: *Mndetprefix-FD-subset*)
have *2*: $(\sqcap xa{\in}\ range\ left \rightarrow \bot) \sqsubseteq_{FD} (left?x \rightarrow \bot)$
unfolding *read-def*
by (*meson Mndetprefix-FD-Mprefix BOT-leFD mono-Mndetprefix-FD trans-FD*)
show $(\Lambda\ x.\ \sqcap xa{\in}(range\ left \cup range\ right) \rightarrow x){\cdot}\bot \sqsubseteq_{FD} COPY$
by *simp* (*metis* (*mono-tags*, *lifting*) *1 2 COPY-rec mono-read-FD BOT-leFD trans-FD*)
next
fix $P::'a\ channel\ process$
assume $*$: $P \sqsubseteq_{FD} COPY$ **and** $**$: $(\Lambda\ x.\ \sqcap xa{\in}(range\ left \cup range\ right) \rightarrow x){\cdot}P \sqsubseteq_{FD} COPY$
have *1*:$\sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow P \sqsubseteq_{FD} \sqcap xa{\in}\ range\ right \rightarrow P$
by (*simp add*: *Mndetprefix-FD-subset*)
have *2*:$\sqcap xa{\in}\ range\ right \rightarrow P \sqsubseteq_{FD} right!x \rightarrow P$ **for** x
apply (*unfold write-def*, *rule trans-FD*[*OF Mndetprefix-FD-subset*[*of* ‹{*right x*}› ‹*range right*›]])
by (*simp-all add*: *Mndetprefix-FD-Mprefix Mprefix-singl*)
from *1 2* **have** *ab*:$\sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow P \sqsubseteq_{FD} right!x \rightarrow P$ **for** x
using *trans-FD* **by** *blast*
hence *3*:$left?x \rightarrow \sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow P \sqsubseteq_{FD} left?x \rightarrow right!x \rightarrow P$
by (*simp add*: *mono-read-FD*)
have *4*:$\bigwedge X.\ \sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow X \sqsubseteq_{FD} \sqcap xa{\in}\ range\ left \rightarrow X$
by (*rule Mndetprefix-FD-subset*, *simp*, *blast*)
have *5*:$\bigwedge X.\ \sqcap xa{\in}\ range\ left \rightarrow X \sqsubseteq_{FD} left?x \rightarrow X$
by (*unfold read-def*, *subst K-record-comp*, *fact Mndetprefix-FD-Mprefix*)
from *3 4*[*of* $\sqcap xa{\in}\ (range\ left \cup range\ right) \rightarrow P$]

5 [*of* $\sqcap xa \in$ (*range left* $\cup$ *range right*) $\rightarrow$ *P*]
have *6*:$\sqcap xa \in$ (*range left* $\cup$ *range right*) $\rightarrow$
$\sqcap xa \in$ (*range left* $\cup$ *range right*) $\rightarrow$ *P* $\sqsubseteq_{FD}$ *left*?$x \rightarrow$ *right*!$x \rightarrow$ *P*
using *trans-FD* **by** *blast*
from $*$ $**$ **have** *7*:*left*?$x \rightarrow$ *right*!$x \rightarrow$ *P* $\sqsubseteq_{FD}$ *left*?$x \rightarrow$ *right*!$x \rightarrow$ *COPY*
by (*simp add*: *mono-read-FD mono-write-FD*)

show (Λ *x*. $\sqcap xa \in$(*range left* $\cup$ *range right*) $\rightarrow$ *x*)$\cdot$
((Λ *x*. $\sqcap xa \in$(*range left* $\cup$ *range right*) $\rightarrow$ *x*)$\cdot$*P*) $\sqsubseteq_{FD}$ *COPY*
by *simp* (*metis* (*mono-tags*, *lifting*) *6 7 COPY-rec trans-FD*)
qed

Part V
Annex: A Note on State-based Processes in HOL-CSP (Restriction Spaces)

theory *FunSeriesInHOL−CSP-RS*
imports *HOL−CSP HOL.Transcendental GenFixrec−RS*
begin

This section is intended to give an intuition to the power of HOL-CSP processes by a set of examples. In our view, it can be stated that HOL-CSP are a more powerful modeling instrument for indefinite behavior than, for example, co-inductive definitions, since they posses a theory of synchronisation and monadic sequential composition that co-inductively defined streams lack. On the other hand, there is no denying that the entire machinery is more complex.

This is the variant of the theory `FunSeriesInHOL-CSP` in which all processes are defined as unique fixpoints of constructive functionals (restriction spaces, υ) instead of least fixpoints of continuous functionals (HOLCF, μ). Uniqueness replaces fixpoint induction in proofs of process equalities (see *parallel-exec-Join* below).

27 The 'Random' Key-Generator

27.1 Example for the basic Construction with υ

abbreviation *key-generator-fun* :: ‹(*'α set* ⇒ *'α process*) ⇒ (*'α set* ⇒ *'α process*)›
  where ‹*key-generator-fun X A* ≡ ⊓*a* ∈ *A* → *X* (*A* − {*a*})›

definition *key-generator* :: ‹*'α set* ⇒ *'α process*› (‹*KEY*›)
  where ‹*key-generator* ≡ υ *X*. (λ *A*. ⊓*a* ∈ *A* → *X* (*A* − {*a*}))›

lemma *key-generator-fun-constructive*[*simp*]: ‹*constructive key-generator-fun*›
  by *simp*

lemma *key-generator-unfold*: ‹*KEY A* = ⊓*a* ∈ *A* → *KEY* (*A* − {*a*})›
  unfolding *key-generator-def* **by** (*subst restriction-fix-eq*) *simp-all*

Now, the process *KEY* ℕ denotes the set of all traces that enumerate the natural numbers without repetition...

27.2 Example in High-level Notation

The *Fixrec* ... *where* ... notation provided by the generic fixpoint-support package has the same effect as the construction above:

Fixrec [*RS*] *keygen* ::*'α set* ⇒ *'α process*
  where *kg-rec* : *keygen A* = (⊓*a* ∈ *A* → *keygen* (*A* − {*a*}))

28 Cauchy-Series as Processes

28.1 The Construction in basic Steps (Again)

It is kind of fun - but also instructive wrt. the power and generality of the process type - to see Cauchy-sequences of numerical approximations in the number system with base 10 as processes. This shows that processes in HOL-CSP are not finite, not rational, but transcendent objects with a non-enumerable cardinality.

definition *digits* :: ‹*real* ⇒ *int process*›
  where ‹*digits* ≡ υ *X*. (λ*x*. ⌊*x*⌋ → *X* ((*frac x*) ∗ *10*)) ›

lemma *digits-unfold*: ‹*digits x* = ⌊*x*⌋ → *digits* (*frac x* ∗ *10*)›
  unfolding *digits-def* **by** (*subst restriction-fix-eq*) *simp-all*

Note that the floor function ⌊*x*⌋ yields the largest integer smaller or equal *x*, while *frac x* returns the rest between ⌊*x*⌋ and *x*.

Now, the process *digits pi* enumerates the representation at base 10 of the transcendental number π, so: *3* → *1* → *4* → *1* → *5* → *9* → ..., while while *digits* (*exp 1*) does the same for the

transcendental number *e*. While these two perfectly well-defined mathematical objects imported from the *HOL.Transcendental* theory have as such no computational content, their approximation by a HOL-CSP process has. Although we do not yet have a code-generator that shows this... ;-)

28.2 The Example again in High-level Notation

The *Fixrec*-package supports this construction directly:

Fixrec [*RS*] *digits′* ::*real* ⇒ *int process*
  where *digits′-rec* : *digits′ x* = ⌊*x*⌋ → *digits′* (*frac x* ∗ *10*)

29 The Collatz-Function as Process

Note that the Collatz function is unknown to be terminating; a standard definition in HOL via the recursive function definitions such as **primrec** or **fun** is therefore out of reach. This example shows, however, that it is perfectly possible to represent the Collatz process as a recursive HOL-CSP process, since HOL-CSP is built for modeling potentially non-terminating computations.

abbreviation *Even* **where** *Even* ≡ *Collect even* — the set of even numbers
abbreviation *Odd* **where** *Odd* ≡ *Collect odd* — the set of odd numbers
abbreviation *Div3* **where** *Div3* ≡ {*x*. *x mod 3* = *0*}

We use the deterministic choice to express a case-distinction:

Fixrec [*RS*] *Collatz* ::*nat* ⇒ *nat process*
  where *Collatz-rec* : *Collatz n* = (□*x* ∈ ({*0*, *1*} ∩ {*n*}) → *Skip*)
□ (□*x* ∈ ((*Even* − {*0*}) ∩ {*n*}) → *Collatz* (*x div 2*))
□ (□*x* ∈ ((*Odd* − {*1*}) ∩ {*n*}) → *Collatz* (*3* ∗ *x* + *1*))

30 True Concurrent Evolution of States.

It is a common misconception of CSP that it can not handle "true-concurrent' computations given that its fundamental concept of *events*: They are "indivisible obervations on a specific time-point". However, the contrary is the case: Since HOL-CSP allows for modeling non-determinism and potentially infinite states, true-concurrent calculations over non-overlapping parts of a state are possible between synchronisation points. If portions of a state are overlapping and computations do not

agree — so: races will occur — then these situations are represented in HOL-CSP as deadlock.

Fixrec [*RS*] $Twos_L :: \langle nat \Rightarrow (nat \times nat)\ process\rangle$
where $Twos_L\text{-}unfold$: $\langle Twos_L\ n_1 = (\Box\sigma \in (\{n_1<..\} \cap Even) \times UNIV \rightarrow Twos_L\ (fst\ \sigma))\rangle$

Fixrec [*RS*] $Threes_R :: \langle nat \Rightarrow (nat \times nat)\ process\rangle$
where $Threes_R\text{-}unfold$: $\langle Threes_R\ n_2 = (\Box\sigma \in UNIV \times (\{n_2<..\} \cap Div3) \rightarrow Threes_R\ (snd\ \sigma))\rangle$

These two processes work on a state σ, consisting of a pair of two counters n_1 and n_2. While $Twos_L$ choses an even number larger than its current local state n_1, $Threes_R$ chooses a number devisable by three larger than its current local state n_2. Putting these two together via:

$Twos_L\ 0\ [\![Id]\!]\ Threes_R\ 0$

... where Id is the identity relation, this synchronization over these two processes calculate independently two values but have to agree on a joint state. A trace of this combined processes is, for example:

$(2,6) \rightarrow (24,24) \rightarrow (16,33) \rightarrow ...$

Of course, this example is a bit simplistic since the the global state σ=(n_1,n_2) has a simple structure, the access to the global variables is distinct and the underlying choices are deter- ministic, so to speak, *angelic*. An obvious generalization of this handling of the common state are algebraic lenses as provided by the theory `Optics` in the AFP. Wrt. to the choice, one might model a more aggressive, i.e. *demonic* behaviour as follows:

Fixrec [*RS*] $Twos_L' :: \langle nat \Rightarrow (nat \times nat)\ process\rangle$
where $Twos_L'\text{-}unfold$: $\langle Twos_L'\ n_1 = (\sqcap\sigma \in (\{n_1<..\} \cap Even) \times UNIV \rightarrow Twos_L'\ (fst\ \sigma))\rangle$

Fixrec [*RS*] $Threes_R' :: \langle nat \Rightarrow (nat \times nat)\ process\rangle$
where $Threes_R'\text{-}unfold$: $\langle Threes_R'\ n_2 = (\sqcap\sigma \in UNIV \times (\{n_2<..\} \cap Div3) \rightarrow Threes_R'\ (snd\ \sigma))\rangle$

The intersection may again be defined via:

$Twos_L'\ 0\ [\![Id]\!]\ Threes_R'\ 0$

but the traces of this combined process will contain explicit deadlocks on all places where an agreement could not be reached due to "ruthless behaviour" demonic processes that chose deviant values from each other... Which means that races occurred.

One might want to prove these facts, i.e. that the synchronized processes produce only streams enjoying the conjunction

of the devisability properties and the agreement is angelic or demonic. The handy recursive equations needed for this, e.g. *Twos$_L$-unfold* and *Threes$_R$'-unfold*, are provided by the *Fixrec*-package.

Of course, we could use this modeling technique — which is roughly similar to the conjunction of two preconditions — by adding a process that ensures that first and second component of the joint agree in their value, that is:
(*6*,*6*) → (*24*,*24*) → (*36*,*36*) → ...
We bind the combined process to the constant *Join*:

Fixrec [*RS*] *Join* :: ‹*nat* × *nat* ⇒ (*nat* × *nat*) *process*›
  where *Join-unfold*: ‹*Join* (*n*$_1$, *n*$_2$) = (□*σ* ∈ ({*n*$_1$<..} ∩ *Even*) × ({*n*$_2$<..} ∩ *Div3*) → *Join σ*)›

And there we go: We state that *Twos$_L$* and *Threes$_R$* in parallel composition are the same as *Join*, which does these computations in one step. In restriction spaces, no fixpoint induction is needed: the parallel composition satisfies the recursive equation of *Join*, whose functional is constructive, hence has a unique fixpoint.

lemma *parallel-exec-Join*: ‹(*Twos$_L$* (*fst σ*) || *Threes$_R$* (*snd σ*)) = *Join σ*›
proof −
  let *?F* = ‹λ*X*. λ(*n*$_1$, *n*$_2$). □*σ* ∈ ({*n*$_1$<..} ∩ *Even*) × ({*n*$_2$<..} ∩ *Div3*) → *X σ*›
  let *?P* = ‹λ*σ*. *Twos$_L$* (*fst σ*) || *Threes$_R$* (*snd σ*)›
  have *eqP* : ‹*?F ?P* = *?P*›
  proof (*rule ext*)
    fix *σ* :: ‹*nat* × *nat*›
    obtain *n*$_1$ *n*$_2$ **where** *σ* : ‹*σ* = (*n*$_1$, *n*$_2$)› **by** (*cases σ*)
   have ‹*?P* (*n*$_1$, *n*$_2$) = (□*σ* ∈ ({*n*$_1$<..} ∩ *Even*) × *UNIV* → *Twos$_L$* (*fst σ*))
                || (□*σ* ∈ *UNIV* × ({*n*$_2$<..} ∩ *Div3*) → *Threes$_R$* (*snd σ*))›
        by (*simp only*: *fst-conv snd-conv*, *subst Twos$_L$-unfold*, *subst Threes$_R$-unfold*, *rule refl*)
    also have ‹... = □*σ* ∈ (({*n*$_1$<..} ∩ *Even*) × *UNIV* ∩ *UNIV* × ({*n*$_2$<..} ∩ *Div3*))
                → (*Twos$_L$* (*fst σ*) || *Threes$_R$* (*snd σ*))›
      by (*rule Mprefix-Par-Mprefix*)
   also have ‹... = *?F ?P* (*n*$_1$, *n*$_2$)› **by** (*simp add*: *Times-Int-Times*)
    finally show ‹*?F ?P σ* = *?P σ*› **by** (*simp only*: *σ*)
  qed
  have *constr* : ‹*constructive ?F*› **by** (*simp add*: *case-prod-beta*)
  have ‹(*υ X*. *?F X*) = *?P*› **using** *restriction-fix-unique*[*OF constr eqP*] **.**
  hence ‹*Join* = *?P*› **by** (*simp add*: *Join-def Join-rec-def*)

thus *?thesis* **by** *metis*
qed

... which is set under the constraint as follows:
Join σ || (υ *X*. ($\lambda\sigma$. *Mprefix Id X*)) σ
We refrain from a formal proof which will follow the lines of *parallel-exec-Join*.

end

Part VI
Example: Dining Philosophers

31 Classic version

We formalize here the Dining Philosophers problem with a locale.

locale *DiningPhilosophers* =

fixes *N*::*nat*
assumes *N-g1*[*simp*] : ‹*N* > *1*›
— We assume that we have at least one right handed philosophers (so at least two philosophers with the left handed one).

begin

We use a datatype for representing the dinner's events.

datatype *dining-event* = *picks* (*phil*:*nat*) (*fork*:*nat*)
| *putsdown* (*phil*:*nat*) (*fork*:*nat*)

We introduce the right handed philosophers, the left handed philosopher and the forks.

definition *RPHIL*:: ‹*nat* ⇒ *dining-event process*›
where ‹*RPHIL i* ≡ μ *X*. (*picks i i* → (*picks i* ((*i*−*1*) *mod N*) →
(*putsdown i* ((*i*−*1*) *mod N*) → (*putsdown i i* →
X))))›

definition *LPHIL0*:: ‹*dining-event process*›
where ‹*LPHIL0* ≡ μ *X*. (*picks 0* (*N*−*1*) → (*picks 0 0* →
(*putsdown 0 0* → (*putsdown 0* (*N*−*1*) → *X*))))›

definition *FORK* :: ‹*nat* ⇒ *dining-event process*›
where ‹*FORK i* ≡ μ *X*. (*picks i i* → (*putsdown i i* → *X*)) □
(*picks* ((*i*+*1*) *mod N*) *i* → (*putsdown* ((*i*+*1*) *mod N*) *i* → *X*))›

Now we use the architectural operators for modelling the interleaving of the philosophers, and the interleaving of the forks.

definition ‹*PHILS* ≡ ||| *P* ∈# *add-mset LPHIL0* (*mset* (*map RPHIL* [*1*..< *N*])). *P*›
definition ‹*FORKS* ≡ ||| *P* ∈# *mset* (*map FORK* [*0*..< *N*]). *P*›

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL 1* ||| *RPHIL 2*)›
— just a test
unfolding *PHILS-def* **by** (*simp add*: *eval-nat-numeral upt-rec Sync-assoc*)

Finally, the dinner is obtained by putting forks and philosophers in parallel.

definition *DINING* :: ‹*dining-event process*›
where ‹*DINING* = (*FORKS* || *PHILS*)›

end

32 Formalization with the Fixrec package

The `Fixrec` package provides a more readable syntax (essentially, it allows us to "get rid of μ" in equations like $\mu\ x.\ P\ x$). In contrast to the `fixrec` package of `HOLCF`, it does not require continuous function spaces: the processes are ordinary (total) HOL functions, and no *cpo* instances of the argument types (like *nat*) are needed. `Fixrec` works inside the locale, the equations may refer to its parameter *N*.

locale *DiningPhilosophers-fixrec* =

fixes *N*::*nat*
assumes *N-g1*[*simp*] : ‹*N* > *1*›
— We assume that we have at least one right handed philosophers (so at least two philosophers with the left handed one).

begin

We use a datatype for representing the dinner's events.

datatype *dining-event* = *picks* (*phil*:*nat*) (*fork*:*nat*)
| *putsdown* (*phil*:*nat*) (*fork*:*nat*)

We introduce the right handed philosophers, the left handed philosopher and the forks.

Fixrec [*RS*, *HOLCF*]
RPHIL :: ‹*nat* ⇒ *dining-event process*›

and *LPHIL0* :: ‹*dining-event process*›
and *FORK* :: ‹*nat* ⇒ *dining-event process*›
where
RPHIL-rec : ‹*RPHIL i* = (*picks i i* → (*picks i* (*i*−*1*) →
(*putsdown i* (*i*−*1*) → (*putsdown i i* →
RPHIL i))))›
| *LPHIL0-rec* : ‹*LPHIL0* = (*picks 0* (*N*−*1*) → (*picks 0 0* →
(*putsdown 0 0* → (*putsdown 0* (*N*−*1*) →
LPHIL0))))›
| *FORK-rec* : ‹*FORK i* = (*picks i i* → (*putsdown i i* → *FORK i*))
□
(*picks* ((*i*+*1*) *mod N*) *i* → (*putsdown* ((*i*+*1*)
mod N) *i* → *FORK i*))›

Now we use the architectural operators for modelling the interleaving of the philosophers, and the interleaving of the forks.

definition ‹*PHILS* ≡ ||| *P* ∈# *add-mset LPHIL0* (*mset* (*map RPHIL* [*1*..<*N*])). *P*›
definition ‹*FORKS* ≡ ||| *P* ∈# *mset* (*map FORK* [*0*..<*N*]). *P*›

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL 1* ||| *RPHIL 2*)›
— just a test
unfolding *PHILS-def* **by** (*simp add*: *eval-nat-numeral upt-rec Sync-assoc*)

Finally, the dinner is obtained by putting forks and philosophers in parallel.

definition *DINING* :: ‹*dining-event process*›
where ‹*DINING* = (*FORKS* || *PHILS*)›

end

Part VII
Example: Plain Old Telephone System

The "Plain Old Telephone Service is a standard medium-size example for architectural modeling of a concurrent system.

Plain old telephone service (POTS), or plain ordinary telephone system,[1] is a retronym for voice-grade telephone service employing analog signal transmission over copper loops. POTS was the standard service offering from telephone companies from

1876 until 1988[2] in the United States when the Integrated Services Digital Network (ISDN) Basic Rate Interface (BRI) was introduced, followed by cellular telephone systems, and voice over IP (VoIP). POTS remains the basic form of residential and small business service connection to the telephone network in many parts of the world. The term reflects the technology that has been available since the introduction of the public telephone system in the late 19th century, in a form mostly unchanged despite the introduction of Touch-Tone dialing, electronic telephone exchanges and fiber-optic communication into the public switched telephone network (PSTN).

C.f. wikipedia https://en.wikipedia.org/wiki/Plain_old_telephone_service.

33 The Alphabet and Basic Types of POTS

Underlying terminology apparent in the acronyms:

1. T-side (target side, callee side)
2. O-side (originator (?) side, caller side)

datatype *MtcO* = *Osetup* | *Odiscon-o*
datatype *MctO* = *Obusy* | *Oalert* | *Oconnect* | *Odiscon-t*
datatype *MtcT* = *Tbusy* | *Talert* | *Tconnect* | *Tdiscon-t*
datatype *MctT* = *Tsetup* | *Tdiscon-o*

type-synonym *Phones* = ‹*int*›

datatype *channels* = *tcO* ‹*Phones* × *MtcO*› —
| *ctO* ‹*Phones* × *MctO*›
| *tcT* ‹*Phones* × *MtcT* × *Phones*›
| *ctT* ‹*Phones* × *MctT* × *Phones*›
| *tcOdial* ‹*Phones* × *Phones*›
| *StartReject Phones* — phone x rejects from now on to be called
| *EndReject Phones* — phone x accepts from now on to be called
| *terminal Phones*
| *off-hook Phones*
| *on-hook Phones*
| *digits* ‹*Phones* × *Phones*› — communication relation: x calls y

| *tone-ring Phones*
| *tone-quiet Phones*
| *tone-busy Phones*

| *tone-dial Phones*
| *connected Phones*

locale *POTS* =
fixes *min-phones* :: *int*
and *max-phones* :: *int*
and *VisibleEvents* :: ‹*channels set*›
assumes *min-phones-g-1*[*simp*] : ‹*1* ≤ *min-phones*›
and *max-phones-g-min-phones*[*simp*] : ‹*min-phones* < *max-phones*›
begin

definition *phones* :: ‹*Phones set*› **where** ‹*phones* ≡ {*min-phones* .. *max-phones*}›

lemma *nonempty-phones*[*simp*]: ‹*phones* ≠ {}›
and *finite-phones*[*simp*]: ‹*finite phones*›
and *at-least-two-phones*[*simp*]: ‹*2* ≤ *card phones*›
and *not-singl-phone*[*simp*]: ‹*phones* − {*p*} ≠ {}›
apply (*simp-all add*: *phones-def*)
using *max-phones-g-min-phones* **apply** *linarith*+
by (*metis atLeastAtMost-iff less-le-not-le max-phones-g-min-phones order-refl singletonD subsetD*)

definition *EventsIPhone* :: ‹*Phones* ⇒ *channels set*›
where ‹*EventsIPhone u* ≡ {*tone-ring u*, *tone-quiet u*, *tone-busy u*, *tone-dial u*, *connected u*}›
definition *EventsUser* :: ‹*Phones* ⇒ *channels set*›
where ‹*EventsUser u* ≡ {*off-hook u*, *on-hook u*} ∪ {*x* . ∃ *n*. *x* = *digits* (*u*, *n*)}›

34 Auxilliaries to Substructure the Specification

abbreviation
Tside-connected :: ‹*Phones* ⇒ *Phones* ⇒ *channels process*›
where ‹*Tside-connected ts os* ≡
(*ctT*!(*ts*,*Tdiscon-o*,*os*) → *tcT*!(*ts*,*Tdiscon-t*,*os*) → *EndReject*!*ts*→*Skip*)
▷ (*tcT*!(*ts*,*Tdiscon-t*,*os*) → *ctT*!(*ts*,*Tdiscon-o*,*os*) → *EndReject*!*ts*→*Skip*)›

abbreviation
Oside-connected :: ‹*Phones* ⇒ *channels process*›
where ‹*Oside-connected ts* ≡
(*ctO*!(*ts*,*Odiscon-t*) → *tcO*!(*ts*,*Odiscon-o*) → *EndReject*!*ts*→*Skip*)

$\triangleright$ ($tcO!(ts, Odiscon\text{-}o) \rightarrow ctO!(ts, Odiscon\text{-}t) \rightarrow EndReject!ts \rightarrow Skip$)›

abbreviation
Oside1 :: ‹[*Phones*, *Phones*] ⇒ *channels process*›
where
‹*Oside1 ts p* ≡ *tcOdial!*(*ts*,*p*)
→ (*ctO!*(*ts*,*Oalert*)
→ *ctO!*(*ts*,*Oconnect*)
→ (*Oside-connected ts*))
□(*ctO!*(*ts*,*Oconnect*) →(*Oside-connected ts*))
□(*ctO!*(*ts*,*Obusy*) → *tcO!*(*ts*,*Odiscon-o*) → *EndReject!ts*
→ *Skip*)›

definition
ITside-connected :: ‹[*Phones*,*Phones*,*channels process*] ⇒ *channels process*›
where
‹*ITside-connected ts os IT* ≡ (*ctT*(*ts*,*Tdiscon-o*,*os*)
→((*tone-busy!ts*
→ *on-hook!ts*
→ *tcT!*(*ts*,*Tdiscon-t*,*os*)
→ *EndReject!ts*
→ *IT*)
□ (*on-hook!ts*
→ *tcT!*(*ts*,*Tdiscon-t*,*os*)
→ *EndReject!ts*
→ *IT*)
))
□ (*on-hook!ts*
→ *tcT!*(*ts*,*Tdiscon-t*,*os*)
→ *ctT!*(*ts*,*Tdiscon-o*,*os*)
→ *EndReject!ts*
→*IT*)›

35 A Telephone

Fixrec [*RS*, *HOLCF*] *T* :: ‹*Phones* ⇒ *channels process*›
and *Oside* :: ‹*Phones* ⇒ *channels process*›
and *Tside* :: ‹*Phones* ⇒ *channels process*›
and *NoReject* :: ‹*Phones* ⇒ *channels process*›
and *Reject* :: ‹*Phones* ⇒ *channels process*›
where
T-rec [*simp del*]: ‹*T ts* = (*Tside ts* ; *T ts*) ▷ (*Oside ts* ; *T ts*)›
| *Oside-rec* [*simp del*]: ‹*Oside ts* = *StartReject!ts*
→ *tcO!*(*ts*,*Osetup*)

```
                                        → (⊓ p ∈ phones. Oside1 ts p)›
 | Tside-rec   [simp del]: ‹Tside ts   = ctT?(y,z,os)|((y,z)=(ts,Tsetup))

                                        → StartReject!ts
                                        → (   tcT!(ts,Talert,os)
                                            → tcT!(ts,Tconnect,os)
                                            →(Tside-connected ts os)
                                          ⊓ (tcT!(ts,Tconnect,os)
                                          → (Tside-connected ts os)))›
 | NoReject-rec [simp del]: ‹NoReject ts = StartReject!ts → Reject ts›
 | Reject-rec  [simp del]: ‹Reject ts  = ctT?(y,z,os)|(y=ts ∧ z=Tsetup
∧ os∈phones ∧ os≠ts)
                                      →     (tcT!(ts,Tbusy,os) → Reject
ts)
                                      □  (EndReject!ts → NoReject ts)›
```

definition *Tel*:: ‹*Phones* ⇒ *channels process*›
where ‹*Tel p* ≡ (*T p* ⟦{*StartReject p*, *EndReject p*}⟧ *NoReject p*) \ {*StartReject p*, *EndReject p*}›

36 A Connector with the Network

```
Fixrec [RS, HOLCF]     Call      :: ‹Phones ⇒ channels process›
 and BUSY      :: ‹Phones ⇒ Phones ⇒ channels process›
 and Connected :: ‹Phones ⇒ Phones ⇒ channels process›
 where
   Call-rec  [simp del]: ‹Call os      = (tcO!  (os,Osetup) → tcO-
dial?(x,ts)|(x=os) → (BUSY os ts)) ; Call os›
 | BUSY-rec  [simp del]: ‹BUSY os ts  = (if ts = os
                        then ctO!(os,Obusy) → tcO!(os,Odiscon-o) →
Skip
                         else ctT!(ts,Tsetup,os)
                             →( (tcT!(ts,Tbusy,os)
                                  → ctO!(os,Obusy)
                                  → tcO!(os,Odiscon-o) → Skip)
                               □
                                (tcT ! (ts,Talert,os)
                                  → ctO!(os,Oalert)
                                  → tcT!(ts,Tconnect,os)
                                  → ctO!(os,Oconnect)
                                  → Connected os ts)
                               □
                                (tcT!(ts,Tconnect,os)
                                  → ctO!(os,Oconnect)
                                  → Connected os ts)))›
```

```
  | Connected-rec [simp del]: ‹Connected os ts =  (tcO!(os,Odiscon-o)
→
                        (( (ctT!(ts,Tdiscon-o,os) → tcT!(ts,Tdiscon-t,os)
→ Skip)
                             □
                           (tcT!(ts,Tdiscon-t,os)→ ctT!(ts,Tdiscon-o,os)
→ Skip)
                           )
                           ; (ctO!(os,Odiscon-t) → Skip)))
                           □
                          (tcT!(ts,Tdiscon-t,os) →
                                (  (ctO!(os,Odiscon-t)
                                   → ctT!(ts,Tdiscon-o,os)
                                   → tcO!(os,Odiscon-o)
                                   → Skip )
                                  □
                                  (tcO!(os,Odiscon-o)
                                   → ctT!(ts,Tdiscon-o,os)
                                   → ctO!(os,Odiscon-t)
                                   → Skip)
                                )
                          )›
```

37 Combining NETWORK and TELEPHONES to a SYSTEM

definition *NETWORK* :: ‹*channels process*›
where ‹*NETWORK* ≡ (||| *os* ∈# (*mset-set phones*). *Call os*)›

definition *TELEPHONES* :: ‹*channels process*›
where ‹*TELEPHONES* ≡ (||| *ts* ∈# (*mset-set phones*). *Tel ts*)›

definition *SYSTEM* :: ‹*channels process*›
where ‹*SYSTEM* ≡ *NETWORK* ⟦*VisibleEvents*⟧ *TELEPHONES*›

We underline here the usefulness of the architectural operators, especially *MultiSync* but also *GlobalNdet* which appears in *Oside* recursive definition.

38 A simple Model of a User

Fixrec [*RS*, *HOLCF*] *User* :: ‹*Phones* ⇒ *channels process*›
and *UserSCon* :: ‹*Phones* ⇒ *channels process*›
where

User-rec : ‹*User u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon u*)
□ (*connected*!*u*→*UserSCon u*)
□ (*tone-busy*!*u*→*on-hook*!*u*→*User u*)
)
)
)
□ (*connected*!*u* → *UserSCon u*)
)
□ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*UserSCon u*)›
| *UserSCon-rec*: ‹*UserSCon u* = (*tone-busy*!*u* → *on-hook*!*u* → *User u*) ▷ (*on-hook*!*u* → *User u*)›

Fixrec [*RS*, *HOLCF*] *User-Ndet* :: ‹*Phones* ⇒ *channels process*›
and *UserSCon-Ndet* :: ‹*Phones* ⇒ *channels process*›
where
User-Ndet-rec : ‹*User-Ndet u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon-Ndet u*)
⊓ (*connected*!*u*→*UserSCon-Ndet u*)
⊓ (*tone-busy*!*u*→*on-hook*!*u*→*User-Ndet u*)
)
)
)
⊓ (*connected*!*u* → *UserSCon-Ndet u*)
)
⊓ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*User-SCon-Ndet u*)›
| *UserSCon-Ndet-rec*: ‹*UserSCon-Ndet u* = (*tone-busy*!*u* → *on-hook*!*u* → *User-Ndet u*)
⊓ (*on-hook*!*u* → *User-Ndet u*)›

definition *ImplementT* :: ‹*Phones* ⇒ *channels process*›
where ‹*ImplementT ts* ≡ ((*Tel ts*) ⟦*EventsIPhone ts* ∪ *EventsUser ts*⟧ (*User ts*))
\ (*EventsIPhone ts* ∪ *EventsUser ts*)›

39 Toplevel Proof-Goals

This has been proven in an ancient FDR model for *max-phones* = *5* ...

lemma ‹∀ *p* ∈ *phones*. *deadlock-free* (*Tel p*)› **oops**
lemma ‹∀ *p* ∈ *phones*. *deadlock-free-v2* (*Call p*)› **oops**
lemma ‹*deadlock-free-v2 NETWORK*› **oops**
lemma ‹*deadlock-free-v2 SYSTEM*› **oops**
lemma ‹*lifelock-free SYSTEM*› **oops**
lemma ‹∀ *p* ∈ *phones*. *lifelock-free* (*ImplementT p*)› **oops**
lemma ‹∀ *p* ∈ *phones*. *Tel p* $\sqsubseteq_{FD}$ *ImplementT p*› **oops**

lemma ‹∀ *p* ∈ *phones*. *Tel′ p* $\sqsubseteq_{F}$ *RUN UNIV*› **oops**

this should represent "deterministic" in process-algebraic terms.
. .
end

theory *Dali-Combined*
imports *HOL−CSPM HOL−CSP.GenFixrec−HOLCF GenFixrec−RS*
begin

This model is based on the IFM 2025 Article: "Model-Checking Buffered Durable Linearizability in CSP" by Edmonds, C., Derrick, J., Dongol, B., Schellhorn, G., Wehrheim, H.

A Zenodo Distribution on the sources can be found under: https://zenodo.org/records/15041794.

Meanwhile, a corrigendum appeared.

type-synonym *NodeIds* = *nat*
type-synonym *Keys* = *nat*
type-synonym *Pointer* = *nat*
type-synonym *Values* = *nat*
type-synonym *Tids* = *nat*
type-synonym *Epoches* = *nat*

The data of the model are indexed by natural numbers; their ranges are bounded by the parameters of the locale *Dali* below (instead of an ad-hoc finitization by enumeration types).

datatype *NodeIDType* = *Null* | *N NodeIds*
datatype *KeyType* = *K Keys*
datatype *Pointers* = *Bot* | *P Pointer*
datatype *ValType* = *TNull* | *V Values*
datatype *ThreadID* = *Th Tids*
datatype *BucketEpochType* = *BE Epoches*

Names for the first elements, as in the original (finite) model:

abbreviation ‹*N0* ≡ *N 0*› **abbreviation** ‹*N1* ≡ *N 1*› **abbreviation** ‹*N2* ≡ *N 2*›
abbreviation ‹*K0* ≡ *K 0*› **abbreviation** ‹*K1* ≡ *K 1*›
abbreviation ‹*P0* ≡ *P 0*› **abbreviation** ‹*P1* ≡ *P 1*› **abbreviation** ‹*P2* ≡ *P 2*›
abbreviation ‹*V0* ≡ *V 0*› **abbreviation** ‹*V1* ≡ *V 1*›
abbreviation ‹*T0* ≡ *Th 0*› **abbreviation** ‹*T1* ≡ *Th 1*› **abbreviation** ‹*T2* ≡ *Th 2*›
abbreviation ‹*BE0* ≡ *BE 0*› **abbreviation** ‹*BE1* ≡ *BE 1*› **abbreviation** ‹*BE2* ≡ *BE 2*›

datatype *channels* = *initNode* ‹*NodeIDType* × *KeyType* × *ValType*›
| *getVal* ‹*NodeIDType* × *ValType*›
| *getKey* ‹*NodeIDType* × *KeyType*›
| *getNext* ‹*NodeIDType* × *NodeIDType*›
| *setNext* ‹*NodeIDType* × *NodeIDType*›
| *noFreeNode*
| *persistNode NodeIDType*
| *crash*

| *setPointerNode* ‹*Pointers* × *NodeIDType*›
| *getPointerNode* ‹*Pointers* × *NodeIDType*›

| *getActiveNode Pointers*
| *getCommittedNode Pointers*
| *getFlightNode Pointers*
| *getBucketEpoch BucketEpochType*
| *setStat* ‹*Pointers* × *Pointers* × *Pointers* × *BucketEpochType*›

| *lock*
| *unlock*

| *invRead* ‹*ThreadID* × *KeyType*›
| *resRead* ‹*ThreadID* × *ValType*›

| *invUpdate* ‹*ThreadID* × *KeyType* × *ValType*›
| *resUpdate ThreadID*
| *resUpdateFull ThreadID*

| *getGlobalEpoch* ‹*ThreadID* × *BucketEpochType*›
| *getGlobalEpochW BucketEpochType*
| *incGlobalEpoch*
| *persistGlobalEpoch*
| *setEpochLock bool*

| *addFailure BucketEpochType*
| *persistFailureList*
| *ssFail* ‹*BucketEpochType* × *bool*›

| *getTIDTransaction* ‹*ThreadID* × *BucketEpochType*›

| *pickFreeTID ThreadID*

| *doRead* ‹*ThreadID* × *KeyType* × *ValType*›
| *doUpdate* ‹*ThreadID* × *KeyType* × *ValType*›
| *doUpdateFull ThreadID*
| *persist*
| *getUsedThreads* ‹*ThreadID set*›

40 Auxiliaries: Traces and Hiding in the Traces Model

FDR checks trace assertions in the traces model, where hiding is just the projection of traces; divergence is irrelevant there. *traces-hiding* captures this notion of hiding.

lemma *replicate-setinterleaves*:
‹$x \in S \Longrightarrow$ *replicate n x setinterleaves* ((*replicate n x*, *replicate n x*), *S*)›
by (*induct n*) *auto*

lemma *replicate-strict-mono*: ‹*strict-mono* (λi. *replicate i x*)›
proof (*rule strict-monoI*)
fix *i j* :: *nat* **assume** ‹$i < j$›
hence ‹*replicate j x* = *replicate i x* @ *replicate* ($j - i$) *x*›
by (*simp add*: *replicate-add*[*symmetric*])
moreover from ‹$i < j$› **have** ‹*replicate i x* ≠ *replicate j x*›
by (*metis length-replicate less-irrefl-nat*)
ultimately show ‹*replicate i x* < *replicate j x*›
by (*auto simp*: *less-list-def less-eq-list-def prefix-def*)
qed

lemma *T-SyncI-right*: ‹$u \in \mathcal{T}$ *Proc2* $\Longrightarrow \forall x{\in}$*set u*. $x \notin$ *range tick* $\cup$ *ev* ‘ *S* $\Longrightarrow u \in \mathcal{T}$ (*Proc1* $[\![S]\!]$ *Proc2*)›
by (*rule T-SyncI*[*OF Nil-elem-T*], *assumption*, *rule emptyLeftSelf*) *blast*
lemma *T-SyncI-left*: ‹$t \in \mathcal{T}$ *Proc1* $\Longrightarrow \forall x{\in}$*set t*. $x \notin$ *range tick* $\cup$ *ev* ‘ *S* $\Longrightarrow t \in \mathcal{T}$ (*Proc1* $[\![S]\!]$ *Proc2*)›
by (*metis Sync-commute T-SyncI-right*)

lemma *T-GlobalDetI*: ‹$x \in A \Longrightarrow s \in \mathcal{T}$ (*Pf x*) $\Longrightarrow s \in \mathcal{T}$ ($\Box x \in A$. *Pf x*)›
by (*auto simp*: *T-GlobalDet′*)

locale *Dali* =
fixes *MaxNIds MaxKeys MaxcPtr MaxVal MaxTid MaxEpoch* :: *nat*
assumes *MaxcPtr-ge-2* : ‹*2* ≤ *MaxcPtr*›
— at least three pointers (active, in-flight, committed)
begin

definition *NodeID* :: ‹*NodeIDType set*›
where ‹*NodeID* = *N* ‘ {*0* .. *MaxNIds*}›

definition *Key* :: ‹*KeyType set*›
where ‹*Key* = *K* ‘ {*0* .. *MaxKeys*}›

definition *Val* :: ‹*ValType set*›
where ‹*Val* = *V* ‘ {*0* .. *MaxVal*}›

definition *Pointer* :: ‹*Pointers set*›
where ‹*Pointer* = *P* ‘ {*0* .. *MaxcPtr*}›

definition *ThreadIDs* :: ‹*ThreadID set*›
where ‹*ThreadIDs* = *Th* ‘ {*0* .. *MaxTid*}›

definition *BucketEpochs* :: ‹*BucketEpochType set*›
where ‹*BucketEpochs* ≡ *BE* ‘ {*0* .. *MaxEpoch*}›

lemma *NodeID-nonempty*[*simp*]: ‹*NodeID* ≠ {}›
and *Null-notin-NodeID*[*simp*]: ‹*Null* ∉ *NodeID*›
and *finite-NodeID*[*simp*]: ‹*finite NodeID*›
and *finite-Key*[*simp*]: ‹*finite Key*›
and *finite-Val*[*simp*]: ‹*finite Val*›
and *finite-Pointer*[*simp*]: ‹*finite Pointer*›
and *finite-ThreadIDs*[*simp*]: ‹*finite ThreadIDs*›
and *finite-BucketEpochs*[*simp*]: ‹*finite BucketEpochs*›
by (*auto simp*: *NodeID-def Key-def Val-def Pointer-def ThreadIDs-def BucketEpochs-def*)

lemma *NodeID-iff*: ‹*x* ∈ *NodeID* ⟷ (∃ *i* ≤ *MaxNIds*. *x* = *N i*)›
and *ThreadIDs-iff*: ‹*t* ∈ *ThreadIDs* ⟷ (∃ *i* ≤ *MaxTid*. *t* = *Th i*)›
by (*auto simp*: *NodeID-def ThreadIDs-def*)

Epochs are counted cyclically modulo *MaxEpoch* + *1*:

fun *epochToInt* :: ‹*BucketEpochType* ⇒ *int*› **where**
‹*epochToInt* (*BE n*) = *int n*›

fun *prevEpoch* :: ‹*BucketEpochType* ⇒ *BucketEpochType*› **where**
‹*prevEpoch* (*BE n*) = *BE* ((*n* + *MaxEpoch*) *mod Suc MaxEpoch*)›

fun *nextEpoch* :: ‹*BucketEpochType* ⇒ *BucketEpochType*› **where**

‹*nextEpoch* (*BE n*) = *BE* (*Suc n mod Suc MaxEpoch*)›

fun *intToEpoch* :: ‹*int* ⇒ *BucketEpochType*› **where**
‹*intToEpoch x* = (*if* 0 ≤ *x* ∧ *x* ≤ *int MaxEpoch then BE* (*nat x*) *else BE MaxEpoch*)›

The first pointer that is neither the active nor the committed one:

definition *freePointer* :: ‹*Pointers* ⇒ *Pointers* ⇒ *Pointers*›
where ‹*freePointer a c* ≡ *P* (*LEAST i*. *P i* ≠ *a* ∧ *P i* ≠ *c*)›

Fixrec [*RS*, *HOLCF*] *NodeHandler* :: ‹*NodeIDType* ⇒ *KeyType* ⇒ *ValType* ⇒ *NodeIDType* ⇒ *bool* ⇒ *channels process*›
and *InitNode* :: ‹*NodeIDType* ⇒ *channels process*›
where *NodeHandler-rec* [*simp del*]: ‹*NodeHandler me k v nxt pers* =
(*getKey*!(*me*,*k*) → *NodeHandler me k v nxt pers*)
□ (*getVal*!(*me*,*v*) → *NodeHandler me k v nxt pers*)
□ (*getNext*!(*me*,*nxt*) → *NodeHandler me k v nxt pers*)
□ (*setNext*?(*m*,*n*)|(*m* = *me* ∧ *n* ∈ *NodeID*) → *NodeHandler me k v n pers*)
□ (*persistNode*!*me* → *NodeHandler me k v nxt True*)
□ (*crash* →(*if pers*
then NodeHandler me k v nxt pers
else InitNode me))›
| *InitNode-rec* [*simp del*]: ‹*InitNode me* =
(*initNode*?(*n*,*k*,*v*)|(*n* = *me*) → *NodeHandler me k v Null False*)
□ (*crash* → *InitNode me*)›

definition *alphaNode* :: ‹*NodeIDType* ⇒ *channels set*›
where ‹*alphaNode me* ≡
{*x*. ∃ *k v*. *x* = *initNode* (*me*,*k*,*v*)}
∪ {*x*. ∃ *v*. *x* = *getVal* (*me*,*v*)}
∪ {*x*. ∃ *k*. *x* = *getKey* (*me*,*k*)}
∪ {*x*. ∃ *n*. *x* = *getNext* (*me*,*n*)}
∪ {*x*. ∃ *n*. *x* = *setNext* (*me*,*n*)}
∪ {*noFreeNode*}
∪ {*persistNode me*}
∪ {*crash*}›

definition *AllNodes* :: ‹*channels process*›
where ‹*AllNodes* ≡ ||| *n* ∈# *mset-set NodeID*. *InitNode n*›

definition *nodeSyncSet* :: ‹*channels set*›
where ‹*nodeSyncSet* ≡ {*x*. ∃ *n*∈*NodeID*. ∃ *k v*. *x* = *initNode* (*n*,*k*,*v*)}
∪ {*x*. ∃ *n*∈*NodeID*. ∃ *k*. *x* = *getKey* (*n*,*k*)}

$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, v.\ x = getVal\ (n,v)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, nxt.\ x = getNext\ (n,nxt)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, nxt.\ x = setNext\ (n,nxt)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ x = persistNode\ n\}$
$\cup \{noFreeNode\}$
$\cup \{crash\}\rangle$

definition *nodeSyncSeti* :: ‹*channels set*›
where ‹$nodeSyncSeti \equiv \{x.\ \exists\, n{\in}NodeID.\ \exists\, k\ v.\ x = initNode\ (n,k,v)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, k.\ x = getKey\ (n,k)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, v.\ x = getVal\ (n,v)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, nxt.\ x = getNext\ (n,nxt)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ \exists\, nxt.\ x = setNext\ (n,nxt)\}$
$\cup \{x.\ \exists\, n{\in}NodeID.\ x = persistNode\ n\}$
$\cup \{noFreeNode\}\rangle$

Fixrec [*RS*, *HOLCF*] *PointerHandler* :: ‹*Pointers* ⇒ *NodeIDType* ⇒ *channels process*›
where *PointerHandler-rec* [*simp del*]: ‹*PointerHandler me node* =
(*setPointerNode*?(*p*,*n*)|(*p* = *me*) → *PointerHandler me n*)
□ (*getPointerNode*!(*me*,*node*) → *PointerHandler me node*)›

definition *Pointers* :: ‹*channels process*›
where ‹*Pointers* ≡ ||| *p* ∈# *mset-set Pointer*. *PointerHandler p Null*›

definition *pointerSyncSet* :: ‹*channels set*›
where ‹*pointerSyncSet* ≡ {*x*. ∃ *p n*. *x* = *setPointerNode* (*p*,*n*)}
∪ {*x*. ∃ *p n*. *x* = *getPointerNode* (*p*,*n*)}›

definition *maxEpoch* :: *BucketEpochType*
where ‹*maxEpoch* ≡ *BE MaxEpoch*›

Fixrec [*RS*, *HOLCF*] *BucketVarHandler* :: ‹*Pointers* ⇒ *Pointers* ⇒ *Pointers* ⇒ *BucketEpochType* ⇒ *channels process*›
where *BucketVarHandler-rec* [*simp del*]: ‹*BucketVarHandler a f c ss* = (*getBucketEpoch*!*ss* → *BucketVarHandler a f c ss*)
□ (*getActiveNode*!*a* → *BucketVarHandler a f c ss*)
□ (*getFlightNode*!*f* → *BucketVarHandler a f c ss*)
□ (*getCommittedNode*!*c* → *BucketVarHandler a f c ss*)
□ (*setStat*?(*newA*,*newF*,*newC*,*newSS*) → *BucketVarHandler newA newF newC newSS*)›

definition *InitBucket* :: ‹*channels process*›
where ‹*InitBucket* ≡ *BucketVarHandler P0 Bot P2 BE0*›

definition *alphaBucket* :: ‹*channels set*›
where ‹*alphaBucket* ≡ {*x*. ∃ *p*. *x* = *getActiveNode p*}
∪ {*x*. ∃ *p*. *x* = *getFlightNode p*}
∪ {*x*. ∃ *p*. *x* = *getCommittedNode p*}
∪ {*x*. ∃ *e*. *x* = *getBucketEpoch e*}
∪ {*x*. ∃ *a f c e*. *x* = *setStat* (*a*,*f*,*c*,*e*)}›

definition *alphaLock* :: ‹*channels set*›
where ‹*alphaLock* ≡ {*lock*, *unlock*}›

Fixrec [*RS*, *HOLCF*] *BucketLockHandler* :: ‹*channels process*›
where *BucketLockHandler-rec* [*simp del*]: ‹*BucketLockHandler* = (*lock* → *unlock* → *BucketLockHandler*)›

Fixrec [*RS*, *HOLCF*] *FailureListHandler* :: ‹*BucketEpochType set* ⇒ *BucketEpochType set* ⇒ *channels process*›
where *FailureListHandler-rec* [*simp del*]: ‹*FailureListHandler fs pfs* = (*addFailure*?*f*
→ *FailureListHandler* (*insert f fs*) *pfs*)
□ (*persistFailureList* → *FailureListHandler fs fs*)
□ (*ssFail*?(*f*,*b*)|(*b* = (*f* ∈ *pfs*)) → *FailureListHandler fs pfs*)
□ (*crash* → *FailureListHandler pfs pfs*)›

definition *InitFailureList* :: ‹*channels process*›
where ‹*InitFailureList* ≡ *FailureListHandler* {} {}›

definition *alphaFailureList* :: ‹*channels set*›
where ‹*alphaFailureList* ≡
{*x*. ∃ *f*. *x* = *addFailure f*}
∪ {*persistFailureList*}
∪ {*x*. ∃ *f b*. *x* = *ssFail* (*f*,*b*)}›

Fixrec [*RS*, *HOLCF*] *GlobalEpochHandler* :: ‹*BucketEpochType* ⇒ *BucketEpochType* ⇒ *bool* ⇒ *channels process*›
where *GlobalEpochHandler-rec* [*simp del*]: ‹*GlobalEpochHandler e pe lt* =
((*if lt*
then (*getGlobalEpoch*?(*t*,*v*)|((*v* = *pe*)) → *GlobalEpochHandler e pe lt*)
else (*getGlobalEpoch*?(*t*,*v*)|((*v* = *e*)) → *GlobalEpochHandler e pe lt*)))
□ (*getGlobalEpochW*!*e* → *GlobalEpochHandler e pe lt*)
□ (*incGlobalEpoch* →
(*if e* = *maxEpoch*

then Skip
else GlobalEpochHandler (nextEpoch e) pe lt))
□ *(persistGlobalEpoch → GlobalEpochHandler e e lt)*
□ *(crash → GlobalEpochHandler pe pe False)*
□ *(setEpochLock?l → GlobalEpochHandler e pe l)*›

definition *InitGlobalEpoch* :: ‹*channels process*›
where ‹*InitGlobalEpoch ≡ GlobalEpochHandler BE0 BE0 False*›

definition *alphaGlobalEpoch* :: ‹*channels set*›
where ‹*alphaGlobalEpoch* ≡
{*x*. ∃ *t e*. *x* = *getGlobalEpoch* (*t*,*e*)}
∪ {*x*. ∃ *e*. *x* = *getGlobalEpochW e*}
∪ {*incGlobalEpoch*}
∪ {*persistGlobalEpoch*}
∪ {*x*. ∃ *l*. *x* = *setEpochLock l*}
∪ {*crash*}›

Fixrec [*RS*, *HOLCF*] *GlobalArrayHandler* :: ‹(*ThreadID* ⇒ *BucketEpochType*) ⇒ *channels process*›
where *GlobalArrayHandler-rec* [*simp del*]: ‹*GlobalArrayHandler tm* =
(*getTIDTransaction*?(*t*,*e*)|(*t* ∈ *ThreadIDs* ∧ *e* = *tm t*) → *GlobalArrayHandler tm*)
□ (*getGlobalEpoch*?(*t*,*e*) → *GlobalArrayHandler* (*tm*(*t* := *e*)))
□ (*resRead*?(*t*,*v*) → *GlobalArrayHandler* (*tm*(*t* := *BE0*)))
□ (*resUpdate*?*t* → *GlobalArrayHandler* (*tm*(*t* := *BE0*)))
□ (*resUpdateFull*?*t* → *GlobalArrayHandler* (*tm*(*t* := *BE0*)))›

definition *initMap* :: ‹*ThreadID* ⇒ *BucketEpochType*›
where ‹*initMap* ≡ (λ-. *BE0*)›

definition *InitGlobalArray* :: ‹*channels process*›
where ‹*InitGlobalArray ≡ GlobalArrayHandler initMap*›

definition *transSyncSet* :: ‹*channels set*›
where ‹*transSyncSet* ≡
{*x*. ∃ *t e*. *x* = *getTIDTransaction* (*t*,*e*)}
∪ {*x*. ∃ *t e*. *x* = *getGlobalEpoch* (*t*,*e*)}
∪ {*x*. ∃ *t v*. *x* = *resRead* (*t*,*v*)}
∪ {*x*. ∃ *t*. *x* = *resUpdate t*}
∪ {*x*. ∃ *t*. *x* = *resUpdateFull t*}›

definition *alphaTransArray* :: ‹*channels set*›
where ‹*alphaTransArray* ≡ {*x*. ∃ *t e*. *x* = *getTIDTransaction* (*t*,*e*)}›

Fixrec [*RS*, *HOLCF*] *FreeTIDsHandler* :: ‹*ThreadID set* ⇒ *channels process*›
  where *FreeTIDsHandler-rec* [*simp del*]: ‹*FreeTIDsHandler F* =
      (*pickFreeTID*?*t*|(*t* ∈ *F*) → *FreeTIDsHandler* (*F* − {*t*}))›

definition *FreeTIDs* :: ‹*channels process*›
  where ‹*FreeTIDs* ≡ *FreeTIDsHandler ThreadIDs*›

definition *SyncSetFreeTIDs* :: ‹*channels set*›
  where ‹*SyncSetFreeTIDs* ≡ {*x*. ∃ *t*. *x* = *pickFreeTID t*}›

41 APPLICATION THREAD Methods

The application thread loops: after a response, it continues as *Thread* (as in *concrete.csp*); the thread methods form one system of mutually recursive processes.

Fixrec [*RS*, *HOLCF*] *ReadSearch* :: ‹*ThreadID* ⇒ *KeyType* ⇒ *NodeIDType* ⇒ *channels process*›
  and *Read* :: ‹*ThreadID* ⇒ *KeyType* ⇒ *channels process*›
  and *SetStat* :: ‹*ThreadID* ⇒ *BucketEpochType* ⇒ *Pointers* ⇒ *Pointers* ⇒ *Pointers* ⇒ *channels process*›
  and *Lookup* :: ‹*ThreadID* ⇒ *BucketEpochType* ⇒ *bool* ⇒ *bool* ⇒ *NodeIDType* ⇒ *Pointers* ⇒ *BucketEpochType* ⇒ *channels process*›
  and *Update′* :: ‹*ThreadID* ⇒ *NodeIDType* ⇒ *BucketEpochType* ⇒ *channels process*›
  and *Update* :: ‹*ThreadID* ⇒ *KeyType* ⇒ *ValType* ⇒ *channels process*›
  and *Thread* :: ‹*ThreadID* ⇒ *channels process*›
  where *ReadSearch-rec* [*simp del*]: ‹*ReadSearch me key validHead* =
      (*if validHead* = *Null*
      then resRead!(*me*,*TNull*) → *unlock* → *Thread me*
      else (*getKey*?(*n*,*k*)|(*n* = *validHead* ∧ *k* ∈ *Key*)
          → (*if key* = *k*
          then (*getVal*?(*n*,*value*)|(*n* = *validHead*) → *resRead*!(*me*,*value*)
              → *unlock* → *Thread me*)
          else (*getNext*?(*n*,*next*)|(*n* = *validHead*) → *ReadSearch me key next*))))›
  | *Read-rec* [*simp del*]: ‹*Read me key* = *lock*
    → *getGlobalEpoch*?(*t*,*e*)|(*t* = *me*) → *getBucketEpoch*?*SS*
    → (*if SS* = *BE0*
      then ReadSearch me key Null
      else ssFail?(*f*,*currFail*)|(*f* = *SS*)
    → (*if* ¬ *currFail*
      then (*getActiveNode*?*headNode* → *getPointerNode*?(*p*,*h*)|(*p* = *headNode*)
          → *ReadSearch me key h*)
      else ssFail?(*f2*,*prevFail*)|(*f2* = *prevEpoch SS*) → *getFlightN-*

ode?*headNodef*
→ (*if* ((¬ *prevFail*) ∧ (*headNodef* ≠ *Bot*))
then getPointerNode?(*p*,*h*)|(*p* = *headNodef*) → *ReadSearch me key h*
else getCommittedNode?*headNodec* → *getPointerNode*?(*p*,*h*)|(*p* = *headNodec*)
→ *ReadSearch me key h*)))›
| *SetStat-rec* [*simp del*]: ‹*SetStat me epoch newA newF newC* = *setStat*!(*newA*,*newF*,*newC*,*epoch*)
→ *resUpdate*!*me*
→ *unlock*
→ *Thread me*›
| *Lookup-rec* [*simp del*]: ‹*Lookup me SS currFail prevFail newNode headNode epoch* =
getPointerNode?(*p*,*h*)|(*p* = *headNode*)
→ *setNext*!(*newNode*,*h*)
→ *getActiveNode*?*oldA*
→ *getFlightNode*?*oldF*
→ *getCommittedNode*?*oldC*
→ (*if SS* = *epoch*
then setPointerNode!(*oldA*,*newNode*) → *SetStat me epoch oldA oldF oldC*
else if (*SS* = *prevEpoch epoch* ∧ (¬ *currFail*) ∧ (¬ *prevFail*) ∧ (*oldF* ≠ *Bot*))
then setPointerNode!(*oldC*,*newNode*) → *SetStat me epoch oldC oldA oldF*
else if (*SS* = *prevEpoch epoch* ∧ (¬ *currFail*) ∧ (*oldF* = *Bot*))
then (*setPointerNode*!(*freePointer oldA oldC*,*newNode*)
→ *SetStat me epoch* (*freePointer oldA oldC*) *oldA oldC*)
else if (*SS* = *prevEpoch epoch* ∧ (¬ *currFail*))
then setPointerNode!(*oldF*,*newNode*) → *SetStat me epoch oldF oldA oldC*
else if (*SS* = *prevEpoch epoch*)
then setPointerNode!(*oldA*,*newNode*) → *SetStat me epoch oldA Bot oldC*
else if (*epochToInt SS* < *epochToInt* (*prevEpoch epoch*) ∧ (¬ *currFail*))
then setPointerNode!(*oldC*,*newNode*) → *SetStat me epoch oldC Bot oldA*
else if (*epochToInt SS* < *epochToInt* (*prevEpoch epoch*) ∧ *currFail* ∧ (¬ *prevFail*) ∧ (*oldF* ≠ *Bot*))
then setPointerNode!(*oldA*,*newNode*) → *SetStat me epoch oldA Bot oldF*
else setPointerNode!(*oldA*,*newNode*) → *SetStat me epoch oldA Bot oldC*)›
| *Update′-rec* [*simp del*]: ‹*Update′ me node gepoch* = *getBucketEpoch*?*SS* →
(*if SS* = *BE0*
then getActiveNode?*headNode* → *Lookup me SS False False*

node headNode gepoch
else ssFail?(f,currFail)|(f = SS) → ssFail?(f2,prevFail)|(f2 = prevEpoch SS)
→ (if ¬ currFail
then getActiveNode?headNode → Lookup me SS currFail prevFail node headNode gepoch
else getFlightNode?headNodef
→ (if ((¬ prevFail) ∧ (headNodef ≠ Bot))
then Lookup me SS currFail prevFail node headNodef gepoch
else getCommittedNode?headNodec
→ Lookup me SS currFail prevFail node headNodec gepoch)))›
| Update-rec [simp del]: ‹Update me key val = lock → getGlobalEpoch?(t,e)|(t = me)
→ ((initNode?(node,k,v)|(k = key ∧ v = val) → Update′ me node e)
□ (noFreeNode → resUpdateFull!me → unlock → Thread me))›
| Thread-rec [simp del]: ‹Thread me =(invRead?(t,k)|(t = me) → Read me k)
□ (invUpdate?(t,k,v)|(t = me) → Update me k v)›

42 WORKER THREAD METHODS

Fixrec [*RS*, *HOLCF*] *WaitOnTransactions* :: ‹*ThreadID set* ⇒ *BucketEpochType* ⇒ *channels process*›
where *WaitOnTransactions-rec* [*simp del*]: ‹*WaitOnTransactions Threads e* =
(*if Threads* = {}
then Skip
else getTIDTransaction?(*t*,*v*)|(*t* ∈ *Threads*) →
(*if v* ≠ *e*
then WaitOnTransactions (*Threads* − {*t*}) *e*
else WaitOnTransactions Threads e))›

Fixrec [*RS*, *HOLCF*] *WriteBackLoop* :: ‹*NodeIDType* ⇒ *NodeIDType* ⇒ *channels process*›
where *WriteBackLoop-rec* [*simp del*]: ‹*WriteBackLoop cnode n* =
(*if n* = *Null*
then Skip
else persistNode!n
→ *getNext?*(*x*,*n2*)|(*x* = *n*)
→ (*if n2* = *cnode*
then Skip
else WriteBackLoop cnode n2))›

Fixrec [*RS*, *HOLCF*] *WriteBack* :: ‹*channels process*›
where *WriteBack-rec* [*simp del*]: ‹*WriteBack* =
getCommittedNode?cnode
→ *getActiveNode?n*

$\rightarrow$ *getPointerNode*?(*p1*,*c*)|(*p1* = *cnode*)
$\rightarrow$ *getPointerNode*?(*p2*,*a*)|(*p2* = *n*)
$\rightarrow$ *WriteBackLoop c a*›

Fixrec [*RS*, *HOLCF*] *WorkerThread* :: ‹*channels process*›
where *WorkerThread-rec* [*simp del*]: ‹*WorkerThread* =
getGlobalEpochW?*e*
$\rightarrow$ *setEpochLock*!*True*
$\rightarrow$ *incGlobalEpoch*
$\rightarrow$ *persistGlobalEpoch*
$\rightarrow$ *setEpochLock*!*False*
$\rightarrow$ ((*WaitOnTransactions ThreadIDs e*)
; *WriteBack*
; *WorkerThread*)›

definition *ApplicationThreads* :: ‹*channels process*›
where ‹*ApplicationThreads* ≡ (*pickFreeTID*?*t1* $\rightarrow$ *Thread t1*) ||| (*pickFreeTID*?*t2* $\rightarrow$ *Thread t2*)›

definition *AllThreads* :: ‹*channels process*›
where ‹*AllThreads* ≡ *ApplicationThreads* ||| *WorkerThread*›

definition *Recovery* :: ‹*channels process*›
where ‹*Recovery* ≡ *getGlobalEpochW*?*F*
$\rightarrow$ *addFailure*!*F*
$\rightarrow$ *addFailure*!(*prevEpoch F*)
$\rightarrow$ *persistFailureList*
$\rightarrow$ *setEpochLock*!*True*
$\rightarrow$ *incGlobalEpoch*
$\rightarrow$ *persistGlobalEpoch*
$\rightarrow$ *setEpochLock*!*False*
$\rightarrow$ *Skip*›

definition *alphaCrash* :: ‹*channels set*›
where ‹*alphaCrash* ≡ {*crash*}›

definition *alphaGlobal* :: ‹*channels set*›
where ‹*alphaGlobal* ≡ *alphaFailureList* ∪ *alphaGlobalEpoch*›

definition *alphaG* :: ‹*channels set*›
where ‹*alphaG* ≡ *alphaGlobal*
∪ *nodeSyncSet*
∪ *alphaBucket*
∪ *SyncSetFreeTIDs*
∪ *alphaCrash*
∪ *pointerSyncSet*›

definition *GlobalProcesses* :: ‹*channels process*›
where ‹*GlobalProcesses* ≡ (((*AllNodes* ⟦*alphaCrash*⟧ *InitGlobalEpoch*)
⟦*alphaCrash*⟧
InitFailureList)
||| *InitBucket*
||| *FreeTIDs*
||| *Pointers*)›

definition *DaliArray* :: ‹*channels process*›
where ‹*DaliArray* ≡ ((*AllThreads* ⟦*transSyncSet*⟧ *InitGlobalArray*)
\ *alphaTransArray*)›

definition *DaliMethods* :: ‹*channels process*›
where ‹*DaliMethods* ≡ ((*DaliArray* ⟦*alphaLock*⟧ *BucketLockHandler*)
\ *alphaLock*)›

definition *internalEvents* :: ‹*channels set*›
where ‹*internalEvents* ≡ *alphaGlobal*
∪ *nodeSyncSeti*
∪ *alphaBucket*
∪ *SyncSetFreeTIDs*
∪ *pointerSyncSet*›

definition *DaliL* :: ‹*channels process*›
where ‹*DaliL* ≡ ((*DaliMethods* ⟦*alphaG*⟧ *GlobalProcesses*) \ *internalEvents*)›

definition *DaliRecovery* :: ‹*channels process*›
where ‹*DaliRecovery* ≡ *Recovery* **;** *DaliMethods*›

Fixrec [*RS*, *HOLCF*] *DaliCrash* :: ‹*channels process*›
and *DaliCrashR* :: ‹*channels process*›
where *DaliCrash-rec* [*simp del*]: ‹*DaliCrash* =
((*DaliMethods* ||| (*crash* → *Skip*))
Θ *a* ∈ *alphaCrash*. *DaliCrashR*)›
| *DaliCrashR-rec* [*simp del*]:
‹*DaliCrashR* =
((*DaliRecovery* ||| (*crash* → *Skip*))
Θ *a* ∈ *alphaCrash*. *DaliCrashR*)›

definition *Dali* :: ‹*channels process*›
where ‹*Dali* ≡ ((*DaliCrash* ⟦*alphaG*⟧ *GlobalProcesses*) \ *internalEvents*)›

definition *DaliE* :: ‹*channels process*›
where ‹*DaliE* ≡ *Dali* \ {*x*. ∃ *t k*. *x* = *invRead* (*t*,*k*)} \ {*x*. ∃ *t k v*. *x* = *invUpdate* (*t*,*k*,*v*)}›

43 ABSTRACT MAP SPECIFICATION

definition *maxNodes* :: *int*
where ‹*maxNodes* ≡ *int* (*card NodeID*)›

type-synonym *MapMem* = ‹*KeyType* ⇒ *ValType*›

type-synonym *MapState* = ‹*MapMem* × *MapMem* × *int* × *int*›

definition *emptyMap* :: *MapMem*
where ‹*emptyMap* ≡ (λ-. *TNull*)›

definition *mapLookupNull* :: ‹*MapMem* ⇒ *KeyType* ⇒ *ValType*›
where ‹*mapLookupNull m k* ≡ *m k*›

definition *mapUpdate* :: ‹*MapMem* ⇒ *KeyType* ⇒ *ValType* ⇒ *MapMem*›
where ‹*mapUpdate m k v* ≡ *m*(*k* := *v*)›

Fixrec [*RS*, *HOLCF*] *MapOps* :: ‹*MapState* ⇒ *channels process*›
where *MapOps-rec* [*simp del*]: ‹*MapOps* (*vmem*, *pmem*, *c*, *pmemc*) =
((*if vmem* = *emptyMap*
then (*doRead*?(*t*,*k*,*v*)|(*v* = *TNull*) → *MapOps* (*vmem*, *pmem*, *c*, *pmemc*))
else (*doRead*?(*t*,*k*,*v*)|(*v* = *mapLookupNull vmem k*) → *MapOps* (*vmem*, *pmem*, *c*, *pmemc*))))
□ ((*if c* < *maxNodes*
then (*doUpdate*?(*t*,*k*,*v*) → *MapOps* (*mapUpdate vmem k v*, *pmem*, *c* + *1*, *pmemc*))
else (*doUpdateFull*?*t* → *MapOps* (*vmem*, *pmem*, *c*, *pmemc*))))
□ (*crash* → (□*x* ∈ {*pmemc*..*c*}. *MapOps* (*pmem*, *pmem*, *x*, *x*)))
□ (*persist* → *MapOps* (*vmem*, *vmem*, *c*, *c*))›

definition *MapSpec0* :: ‹*channels process*›
where ‹*MapSpec0* ≡ *MapOps* (*emptyMap*, *emptyMap*, *0*, *0*)›

definition *MapSpec* :: ‹*channels process*›
where ‹*MapSpec* ≡ *MapSpec0*›

Fixrec [*RS*, *HOLCF*] *Persist* :: ‹*channels process*›
where *Persist-rec* : ‹*Persist* = *persist* → *Persist*›

Fixrec [*RS*, *HOLCF*] *AllThreadHandler* :: ‹*ThreadID set* ⇒ *channels process*›

where *AllThreadHandler-rec* : ‹*AllThreadHandler usedThreads* =
(*invRead*?(*t*,*k*) → *AllThreadHandler* (*insert t usedThreads*))
□ (*invUpdate*?(*t*,*k*,*v*) → *AllThreadHandler* (*insert t usedThreads*))
□ (*getUsedThreads*!*usedThreads* → *AllThreadHandler usedThreads*)›

Fixrec [*RS*, *HOLCF*] *MapThread* :: ‹*ThreadID* ⇒ *channels process*›
where *MapThread-rec* : ‹*MapThread me* = (*invRead*?(*t*,*key*)|(*t* = *me*)
→ *doRead*?(*t2*,*k2*,*value*)|(*t2* = *me* ∧ *k2* = *key*)
→ *resRead*!(*me*,*value*) → *MapThread me*)
□ (*invUpdate*?(*t*,*key*,*value*)|(*t* = *me*)
→ ((*doUpdateFull*!*me*
→ *resUpdateFull*!*me* → *MapThread me*)
□ (*doUpdate*!(*me*,*key*,*value*)
→ *resUpdate*!*me*
→ *MapThread me*)))›

Fixrec [*RS*, *HOLCF*] *AllMapThreads* :: ‹*channels process*›
where *AllMapThreads-rec* : ‹*AllMapThreads* = *getUsedThreads*?*ts*
→ (||| *t* ∈# *mset-set* (*ThreadIDs* − *ts*). *MapThread t*)›

definition *specSyncSet* :: ‹*channels set*›
where ‹*specSyncSet* ≡ {*x*. ∃ *t k v*. *x* = *doUpdate* (*t*,*k*,*v*)}
∪ {*x*. ∃ *t k v*. *x* = *doRead* (*t*,*k*,*v*)}
∪ {*x*. ∃ *t*. *x* = *doUpdateFull t*}
∪ {*crash*}
∪ {*persist*}›

definition *internal* :: ‹*channels set*›
where ‹*internal* ≡ {*x*. ∃ *t k v*. *x* = *doUpdate* (*t*,*k*,*v*)}
∪ {*x*. ∃ *t k v*. *x* = *doRead* (*t*,*k*,*v*)}
∪ {*x*. ∃ *t*. *x* = *doUpdateFull t*}
∪ {*persist*}›

definition *threadSyncSet* :: ‹*channels set*›
where ‹*threadSyncSet* ≡ {*x*. ∃ *t k*. *x* = *invRead* (*t*,*k*)}
∪ {*x*. ∃ *t k v*. *x* = *invUpdate* (*t*,*k*,*v*)}
∪ {*x*. ∃ *ts*. *x* = *getUsedThreads ts*}›

definition *internalSpec* :: ‹*channels set*›
where ‹*internalSpec* ≡ {*x*. ∃ *t k v*. *x* = *doUpdate* (*t*,*k*,*v*)}
∪ {*x*. ∃ *t k v*. *x* = *doRead* (*t*,*k*,*v*)}

$\cup \{x.\ \exists\, t.\ x = doUpdateFull\ t\}$
$\cup \{persist\}$›

definition *alphaThread* :: ‹*channels set*›
where ‹*alphaThread* ≡ $\{x.\ \exists\, ts.\ x = getUsedThreads\ ts\}$›

definition *SpecThreadsL* :: ‹*channels process*›
where ‹*SpecThreadsL* ≡ ((*AllMapThreads* ⟦*threadSyncSet*⟧ *AllThreadHandler* {}) \ *alphaThread*)›

definition *SpecL* :: ‹*channels process*›
where ‹*SpecL* ≡ ((*SpecThreadsL* ⟦*specSyncSet*⟧ *MapSpec*) \ *internalSpec*)›

Fixrec [*RS*, *HOLCF*] *Spec-with-crash* :: ‹*channels process*›
where *Spec-with-crash-rec* : ‹*Spec-with-crash* = ((*AllMapThreads* ||| (*crash* → *Skip*))
Θ *a* ∈ {*crash*}. *Spec-with-crash*)›

definition *SpecThreads* :: ‹*channels process*›
where ‹*SpecThreads* ≡ (((*Spec-with-crash* ⟦*threadSyncSet*⟧ *AllThreadHandler* {}) \ *alphaThread*)
||| *Persist*)›

definition *Spec* :: ‹*channels process*›
where ‹*Spec* ≡ ((*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*) \ *internalSpec*)›

Note that the event *persist* is in *internalSpec*, and since *Persist* = *persist* → *Persist* can happen at any time, the hiding operator will thus produce a ⊥ for this trace. Thus, there is a divergence in this model (which was already present in the original version by Edmonds, Derrick, et al). This limits the current model to trace - and failure analysis. An alternative would have been to exclude *persist* from the internals, but this has other consequences.

Formally: an infinite run of hidden *persist* events makes *Spec* divergent from the start, i.e., equal to ⊥:

lemma *persist-Persist*: ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ *Persist*›
by (*induct n*) (*simp*, *subst Persist-rec*, *simp add*: *T-write0*)

lemma *persist-MapOps*: ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ (*MapOps* (*m*, *m*, *c*, *c*))›
proof (*induct n arbitrary*: *m c*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **show** *?case*

by (*subst MapOps-rec*, *unfold T-Det*, *rule UnI2*) (*simp add*: *T-write0 Suc*)
qed

lemma *persist-SpecThreads*: ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ *SpecThreads*›
unfolding *SpecThreads-def*
by (*rule T-SyncI*[*OF Nil-elem-T persist-Persist*], *rule emptyLeftSelf*) *auto*

lemma *persist-SpecU*: ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*)›
proof (*unfold MapSpec-def MapSpec0-def*, *rule T-SyncI*)
show ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ *SpecThreads*› **by** (*fact persist-SpecThreads*)
show ‹*replicate n* (*ev persist*) ∈ $\mathcal{T}$ (*MapOps* (*emptyMap*, *emptyMap*, *0*, *0*))› **by** (*fact persist-MapOps*)
show ‹*replicate n* (*ev persist*) *setinterleaves*
((*replicate n* (*ev persist*), *replicate n* (*ev persist*)), *range tick* ∪ *ev* ‘ *specSyncSet*)›
by (*rule replicate-setinterleaves*) (*simp add*: *specSyncSet-def*)
qed

lemma *Spec-is-BOT*: ‹*Spec* = ⊥›
proof −
let *?f* = ‹λ*i*. *replicate i* (*ev persist*)›
have *run* : ‹*isInfHiddenRun ?f* (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*) *internalSpec*›
by (*simp add*: *replicate-strict-mono persist-SpecU internalSpec-def*)
have *start* : ‹[] ∈ *range ?f*› **by** (*metis range-eqI replicate-0*)
have *hide* : ‹[] = *trace-hide* [] (*ev* ‘ *internalSpec*) @ []› **by** *simp*
have ‹[] ∈ $\mathcal{D}$ *Spec*›
unfolding *Spec-def D-Hiding mem-Collect-eq*
using *run start hide ftF-Nil tF-Nil* **by** *blast*
thus ‹*Spec* = ⊥› **by** (*simp add*: *BOT-iff-Nil-D*)
qed

definition *alphaEndEvents* :: ‹*channels set*›
where ‹*alphaEndEvents* ≡ {*x*. ∃ *t v*. *x* = *resRead* (*t*,*v*)}
∪ {*x*. ∃ *t*. *x* = *resUpdate t*}
∪ {*x*. ∃ *t*. *x* = *resUpdateFull t*}›

definition *SpecE* :: ‹*channels process*›
where ‹*SpecE* ≡ *Spec* \ *alphaEndEvents*›

end

44 The General Dali Model Instantiated

The original, finite model of the article is the instance with three nodes and pointers, two keys, values and threads, and three epochs:

locale *Dali-orig* = *Dali 2 1 2 1 1 2*
— the instance as a context for tests: inside, *Spec*, *Dali*, *NodeID*, ... denote the processes and data of this instance

The instance is consistent:

interpretation *orig* : *Dali-orig* **by** *standard simp*

context *Dali-orig* **begin**

lemma ‹*NodeID* = {*N0*, *N1*, *N2*}› **and** ‹*ThreadIDs* = {*T0*, *T1*}›
unfolding *set-eq-iff NodeID-iff ThreadIDs-iff*
by (*auto simp*: *numeral-2-eq-2 le-Suc-eq*)

44.1 Component Traces

lemma *ThreadIDs-orig*: ‹*ThreadIDs* = {*T0*, *T1*}›
unfolding *set-eq-iff ThreadIDs-iff* **by** (*auto simp*: *le-Suc-eq*)

lemma *NodeID-orig*: ‹*NodeID* = {*N0*, *N1*, *N2*}›
unfolding *set-eq-iff NodeID-iff* **by** (*auto simp*: *numeral-2-eq-2 le-Suc-eq*)

lemma *maxNodes-orig*: ‹*maxNodes* = *3*› **by** (*simp add*: *maxNodes-def NodeID-orig*)

lemma *MapThread-update-trace*:
‹[*ev* (*invUpdate* (*T0*,*K0*,*V0*)), *ev* (*doUpdate* (*T0*,*K0*,*V0*)), *ev* (*resUpdate T0*)] ∈ $\mathcal{T}$ (*MapThread T0*)›
apply (*subst MapThread-rec*, *rule T-DetI2*, *rule T-readI*[**where** *x* = ‹(*T0*,*K0*,*V0*)›])
apply (*simp-all add*: *inj-on-def*)
by (*rule T-DetI2*, *rule T-writeI*, *rule T-writeI*, *rule Nil-elem-T*)

lemma *MapThread-read-trace*:
‹[*ev* (*invRead* (*T1*,*K0*)), *ev* (*doRead* (*T1*,*K0*,*V0*)), *ev* (*resRead* (*T1*,*V0*))] ∈ $\mathcal{T}$ (*MapThread T1*)›
apply (*subst MapThread-rec*, *rule T-DetI1*, *rule T-readI*[**where** *x* = ‹(*T1*,*K0*)›])
apply (*simp-all add*: *inj-on-def*)
apply (*rule T-readI*)
apply (*simp-all add*: *inj-on-def*)
by (*rule T-writeI*, *rule Nil-elem-T*)

lemma *AllMapThreads-trace1*:
‹[*ev* (*getUsedThreads* {}),

```
    ev (invUpdate (T0,K0,V0)),
    ev (doUpdate (T0,K0,V0)),
    ev (resUpdate T0)]
   ∈ 𝒯 AllMapThreads›
  apply (subst AllMapThreads-rec, rule T-readI[where x = ‹{}›])
   apply (simp-all add: inj-on-def ThreadIDs-orig[simplified])
  by (rule T-SyncI-left[OF MapThread-update-trace]) auto

lemma AllMapThreads-trace2:
  ‹[ev (getUsedThreads {T0}),
    ev (invRead (T1,K0)),
    ev (doRead (T1,K0,V0)),
    ev (resRead (T1,V0))]
   ∈ 𝒯 AllMapThreads›
  apply (subst AllMapThreads-rec, rule T-readI[where x = ‹{T0}›])
   apply (simp-all add: inj-on-def ThreadIDs-orig[simplified] insert-Diff-if)
  using MapThread-read-trace by simp

lemma Spec-with-crash-trace:
  ‹[ev (getUsedThreads {}), ev (invUpdate (T0,K0,V0)), ev (doUpdate
(T0,K0,V0)), ev (resUpdate T0)]
   @ ev crash #
    [ev (getUsedThreads {T0}), ev (invRead (T1,K0)), ev (doRead
(T1,K0,V0)), ev (resRead (T1,V0))]
   ∈ 𝒯 Spec-with-crash›
proof (subst Spec-with-crash-rec, rule T-ThrowI2)
   show ‹[ev (getUsedThreads {}), ev (invUpdate (T0,K0,V0)), ev
(doUpdate (T0,K0,V0)), ev (resUpdate T0)]
        @ [ev crash] ∈ 𝒯 (AllMapThreads ||| (crash → Skip))›
   by (rule T-SyncI[OF AllMapThreads-trace1 T-write0I[OF Nil-elem-T]])
auto
  show ‹[ev (getUsedThreads {T0}), ev (invRead (T1,K0)), ev (doRead
(T1,K0,V0)), ev (resRead (T1,V0))]
        ∈ 𝒯 Spec-with-crash›
   by (subst Spec-with-crash-rec, rule T-ThrowI1, rule T-SyncI-left[OF
AllMapThreads-trace2]) auto
qed auto

lemma AllThreadHandler-trace:
  ‹[ev (getUsedThreads {}),
    ev (invUpdate (T0,K0,V0)),
    ev (getUsedThreads {T0}),
    ev (invRead (T1,K0))]
   ∈ 𝒯 (AllThreadHandler {})›
  apply (subst AllThreadHandler-rec, rule T-DetI2, rule T-writeI)
  apply (subst AllThreadHandler-rec, rule T-DetI1, rule T-DetI2)
 apply (rule T-readI, simp add: inj-on-def, simp, simp only: prod.case)
  apply (subst AllThreadHandler-rec, rule T-DetI2, rule T-writeI)
  apply (subst AllThreadHandler-rec, rule T-DetI1, rule T-DetI1)
```

by (*rule T-readI*, *simp add*: *inj-on-def*, *simp*, *rule Nil-elem-T*)

lemma *MapSpec-trace*:
‹[*ev* (*doUpdate* (*T0*,*K0*,*V0*)), *ev persist*, *ev crash*, *ev* (*doRead* (*T1*,*K0*,*V0*))]
∈ $\mathcal{T}$ *MapSpec*›
proof −
have *c0*: ‹(*0*::*int*) < *maxNodes*› **by** (*simp add*: *maxNodes-orig*)
have *m1*: ‹*mapUpdate emptyMap K0 V0* ≠ *emptyMap*›
by (*auto simp*: *mapUpdate-def emptyMap-def fun-eq-iff*)
show *?thesis*
unfolding *MapSpec-def MapSpec0-def*
apply (*subst MapOps-rec*, *rule T-DetI1*, *rule T-DetI1*, *rule T-DetI2*)
apply (*simp only*: *if-P*[*OF c0*])
apply (*rule T-readI*, *simp add*: *inj-on-def*, *simp*, *simp only*: *prod.case*)
apply (*subst MapOps-rec*, *rule T-DetI2*, *rule T-write0I*)
apply (*subst MapOps-rec*, *rule T-DetI1*, *rule T-DetI2*, *rule T-write0I*)
apply (*rule T-GlobalDetI*[**where** *x* = ‹*0* + *1*›], *simp*)
apply (*subst MapOps-rec*, *rule T-DetI1*, *rule T-DetI1*, *rule T-DetI1*)
apply (*simp only*: *if-not-P*[*OF m1*])
by (*rule T-readI*, *simp add*: *inj-on-def*,
simp add: *mapLookupNull-def mapUpdate-def*, *rule Nil-elem-T*)
qed

lemma *crash-Spec-with-crash*: ‹[*ev crash*] ∈ $\mathcal{T}$ *Spec-with-crash*›
by (*subst Spec-with-crash-rec*, *subst AllMapThreads-rec*)
(*simp add*: *read-Sync-write0-indep Throw-Det Throw-write0 T-Det T-write0*)

45 Assertions

Following *assertions.csp* in the original text.

lemma *dali-trace-refinement*: ‹*Spec* $\sqsubseteq_T$ *Dali*› **oops**

lemma *daliL-trace-refinement*: ‹*SpecL* $\sqsubseteq_T$ *DaliL*› **oops**

definition *traces-hiding* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ '*a set* ⇒ ('*a*, '*r*) *trace*$_{ptick}$ *set*›
where ‹*traces-hiding Proc A* ≡ {*trace-hide t* (*ev* ' *A*) |*t*. *t* ∈ $\mathcal{T}$ *Proc*}›

lemma *traces-hidingI*: ‹*t* ∈ $\mathcal{T}$ *Proc* ⟹ *s* = *trace-hide t* (*ev* ' *A*) ⟹ *s* ∈ *traces-hiding Proc A*›
unfolding *traces-hiding-def* **by** *blast*

lemma *spec-has-crash-trace*:
‹[*ev crash*] ∈ *traces-hiding* (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*) *internalSpec*›
proof (*rule traces-hidingI*)
have ‹[*ev crash*] ∈ $\mathcal{T}$ *SpecThreads*›

proof (*unfold SpecThreads-def*, *rule T-SyncI-left*)
have *inner*: ‹[*ev crash*] ∈ $\mathcal{T}$ (*Spec-with-crash* ⟦*threadSyncSet*⟧ *AllThreadHandler* {})›
by (*rule T-SyncI-left*[*OF crash-Spec-with-crash*]) (*auto simp*: *threadSyncSet-def*)
have ‹*trace-hide* [*ev crash*] (*ev* ‘ *alphaThread*) = [*ev crash*]› **by** (*auto simp*: *alphaThread-def*)
with *mem-T-imp-mem-T-Hiding*[*OF inner*, *of alphaThread*]
show ‹[*ev crash*] ∈ $\mathcal{T}$ ((*Spec-with-crash* ⟦*threadSyncSet*⟧ *AllThreadHandler* {}) \ *alphaThread*)›
by *metis*
qed *auto*
moreover have ‹[*ev crash*] ∈ $\mathcal{T}$ *MapSpec*›
unfolding *MapSpec-def MapSpec0-def*
by (*subst MapOps-rec*, *rule T-DetI1*, *rule T-DetI2*, *rule T-write0I*, *rule Nil-elem-T*)
ultimately show ‹[*ev crash*] ∈ $\mathcal{T}$ (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*)›
by (*rule T-SyncI*) (*simp add*: *specSyncSet-def*)
qed (*auto simp*: *internalSpec-def*)

lemma *spec-trace-crash-read-value*:
‹*map ev* [*invUpdate* (*T0*,*K0*,*V0*), *resUpdate T0*, *crash*, *invRead* (*T1*,*K0*), *resRead* (*T1*,*V0*)]
∈ *traces-hiding* (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*) *internalSpec*›
proof (*rule traces-hidingI*)
let *?tIn* = ‹[*ev* (*getUsedThreads* {}), *ev* (*invUpdate* (*T0*,*K0*,*V0*)), *ev* (*doUpdate* (*T0*,*K0*,*V0*)),
ev (*resUpdate T0*), *ev crash*, *ev* (*getUsedThreads* {*T0*}), *ev* (*invRead* (*T1*,*K0*)),
ev (*doRead* (*T1*,*K0*,*V0*)), *ev* (*resRead* (*T1*,*V0*))]›
let *?tX* = ‹[*ev* (*invUpdate* (*T0*,*K0*,*V0*)), *ev* (*doUpdate* (*T0*,*K0*,*V0*)), *ev* (*resUpdate T0*), *ev crash*,
ev (*invRead* (*T1*,*K0*)), *ev* (*doRead* (*T1*,*K0*,*V0*)), *ev* (*resRead* (*T1*,*V0*))]›
let *?tST* = ‹[*ev* (*invUpdate* (*T0*,*K0*,*V0*)), *ev* (*doUpdate* (*T0*,*K0*,*V0*)), *ev* (*resUpdate T0*), *ev persist*,
ev crash, *ev* (*invRead* (*T1*,*K0*)), *ev* (*doRead* (*T1*,*K0*,*V0*)), *ev* (*resRead* (*T1*,*V0*))]›
have *inner*: ‹*?tIn* ∈ $\mathcal{T}$ (*Spec-with-crash* ⟦*threadSyncSet*⟧ *AllThreadHandler* {})›
by (*rule T-SyncI*[*OF Spec-with-crash-trace AllThreadHandler-trace*]) (*auto simp*: *threadSyncSet-def*)
have *hide*: ‹*trace-hide ?tIn* (*ev* ‘ *alphaThread*) = *?tX*› **by** (*auto simp*: *alphaThread-def*)
have *X*: ‹*?tX* ∈ $\mathcal{T}$ ((*Spec-with-crash* ⟦*threadSyncSet*⟧ *AllThreadHandler* {}) \ *alphaThread*)›
using *mem-T-imp-mem-T-Hiding*[*OF inner*, *of alphaThread*] *hide*
by *metis*

have *persist*: ‹[*ev persist*] ∈ $\mathcal{T}$ *Persist*›
by (*subst Persist-rec*, *rule T-write0I*, *rule Nil-elem-T*)
have *ST*: ‹*?tST* ∈ $\mathcal{T}$ *SpecThreads*›
unfolding *SpecThreads-def* **by** (*rule T-SyncI*[*OF X persist*]) *auto*
show ‹*?tST* ∈ $\mathcal{T}$ (*SpecThreads* ⟦*specSyncSet*⟧ *MapSpec*)›
by (*rule T-SyncI*[*OF ST MapSpec-trace*]) (*auto simp*: *specSyncSet-def*)
show ‹*map ev* [*invUpdate* (*T0*,*K0*,*V0*), *resUpdate T0*, *crash*, *invRead* (*T1*,*K0*), *resRead* (*T1*,*V0*)]
= *trace-hide ?tST* (*ev* ‘ *internalSpec*)›
by (*auto simp*: *internalSpec-def*)
qed

lemma *spec-trace-crash-read-null*:
‹*map*(*ev*)[
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*)
] ∈ *Traces Spec*›
oops

lemma *dali-has-crash-trace*: ‹[*ev crash*] ∈ *Traces Dali*› **oops**

lemma *dali-simple-update-trace*: ‹*map*(*ev*)[*invUpdate* (*T0*,*K0*,*V0*), *resUpdate T0*] ∈ *Traces Dali*› **oops**

lemma *dali-trace-crash-read-null*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*)
] ∈ *Traces Dali*›
oops

lemma *dali-trace-crash-read-value*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invRead (*T1*,*K0*),
resRead (*T1*,*V0*)
] ∈ *Traces Dali*›
oops

lemma *spec-trace-complex-1*:

‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
resUpdate T1,
invRead (*T1*,*K1*),
resRead (*T1*,*V1*),
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*)
] ∈ *Traces Spec*›
oops

lemma *spec-trace-complex-2*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
resUpdate T1,
invRead (*T1*,*K1*),
resRead (*T1*,*V1*),
invRead (*T1*,*K0*),
resRead (*T1*,*V0*)
] ∈ *Traces Spec*›
oops

lemma *dali-trace-complex-1*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
resUpdate T1,
invRead (*T1*,*K1*),
resRead (*T1*,*V1*),
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*)
] ∈ *Traces Dali*›
oops

lemma *dali-trace-complex-2*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),

resUpdate T0,
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
resUpdate T1,
invRead (*T1*,*K1*),
resRead (*T1*,*V1*),
invRead (*T1*,*K0*),
resRead (*T1*,*V0*)
] ∈ *Traces Dali*›
oops

lemma *spec-trace-multi-crash-1*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
crash,
invUpdate (*T2*,*K1*,*V1*),
resUpdate T2,
invRead (*T2*,*K1*),
resRead (*T2*,*V1*),
invRead (*T2*,*K0*),
resRead (*T2*,*TNull*)
] ∈ *Traces Spec*›
oops

lemma *spec-trace-multi-crash-2*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
crash,
invRead (*T2*,*K1*),
resRead (*T2*,*V0*),
invRead (*T2*,*K0*),
resRead (*T2*,*V0*)
] ∈ *Traces Spec*›
oops

lemma *dali-trace-complex-1*:

‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
resUpdate T1,
invRead (*T1*,*K1*),
resRead (*T1*,*V1*),
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*)
] ∈ *Traces Dali*›
oops

lemma *dali-trace-multi-crash-epoch*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K1*,*V1*),
crash,
invRead (*T2*,*K1*),
resRead (*T2*,*V0*),
invRead (*T2*,*K0*),
resRead (*T2*,*V0*)
] ∈ *Traces Dali*›
oops

lemma *spec-trace-multi-crash-3*:
‹*map*(*ev*)[
invUpdate (*T0*,*K0*,*V0*),
resUpdate T0,
crash,
crash,
invRead (*T1*,*K0*),
resRead (*T1*,*TNull*),
invUpdate (*T1*,*K1*,*V0*),
resUpdate T1,
invUpdate (*T1*,*K0*,*V0*),
resUpdate T1,
crash,
invRead (*T2*,*K1*),
resRead (*T2*,*V0*),
invRead (*T2*,*K0*),
resRead (*T2*,*TNull*)

```
  ] ∈ Traces Spec›
  oops


lemma dali-trace-multi-crash-3:
  ‹map(ev) [
      invUpdate (T0,K0,V0),
      resUpdate T0,
      crash,
      crash,
      invRead (T1,K0),
      resRead (T1,TNull),
      invUpdate (T1,K1,V0),
      resUpdate T1,
      invUpdate (T1,K0,V0),
      resUpdate T1,
      crash,
      invRead (T2,K1),
      resRead (T2,V0),
      invRead (T2,K0),
      resRead (T2,TNull)
  ] ∈ Traces Dali›
  oops


lemma spec-trace-concurrent-1:
  ‹map(ev) [
      invUpdate (T0,K0,V0),
      invUpdate (T1,K1,V1),
      resUpdate T0,
      resUpdate T1,
      crash,
      invRead (T2,K1),
      resRead (T2,V1),
      invRead (T2,K0),
      resRead (T2,V0)
  ] ∈ Traces Spec›
  oops


lemma spec-trace-concurrent-2:
  ‹map(ev) [
      invUpdate (T0,K0,V0),
      invUpdate (T1,K1,V1),
      resUpdate T0,
      resUpdate T1,
      crash,
      invRead (T2,K1),
      resRead (T2,TNull),
```

invRead (*T2*,*K0*),
resRead (*T2*, *V0*)
] ∈ *Traces Spec*›
oops

lemma *dali-trace-concurrent-1*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*, *V0*),
invUpdate (*T1*,*K1*, *V1*),
resUpdate T0,
resUpdate T1,
crash,
invRead (*T2*,*K1*),
resRead (*T2*, *V1*),
invRead (*T2*,*K0*),
resRead (*T2*, *V0*)
] ∈ *Traces Dali*›
oops

lemma *dali-trace-concurrent-2*:
‹*map*(*ev*) [
invUpdate (*T0*,*K0*, *V0*),
invUpdate (*T1*,*K1*, *V1*),
resUpdate T0,
resUpdate T1,
crash,
invRead (*T2*,*K1*),
resRead (*T2*, *TNull*),
invRead (*T2*,*K0*),
resRead (*T2*, *V0*)
] ∈ *Traces Dali*›
oops

end

46 Check: Dali is Accepted in Restriction Spaces

With *Fixrec* [*RS*, *HOLCF*], a failing constructiveness proof falls back to HOLCF silently. The following assertion guards against this: all recursive definitions of Dali must be restriction fixpoints υ (this relies in particular on *ThrowR-constructive* for the crash

handling).

ML ‹
local
  val ctxt = @{*context*}
  val rec-defs = *Facts.dest-static false* [] (*Proof-Context.facts-of ctxt*)
|> *map fst*
   |> *filter* (*fn n => String.isPrefix Dali-Combined. n andalso String.isSuffix -rec-def n*)
 fun is-rs n = *exists-Const* (*fn* (*c*, -) => *c* = ***const-name*** ‹*restriction-fix*›)
                 (*Thm.prop-of* (*hd* (*Proof-Context.get-thms ctxt n*)))
  val non-rs = *filter-out is-rs rec-defs*
in
  val - = @{*assert*} (*not* (*null rec-defs*))
  val - = *null non-rs orelse error* (*Dali*: *not accepted in RS*: ^ *commas non-rs*)
end
›

end