HOL-CSP Version 2.0

Burkhart Wolff

October 11, 2012

Contents

0.1 Directed sets 2

1 Hoare/Roscoe's Denotational Semantics for CSP The Notion of Processes **3**
1.1 Pre-Requisite: Basic Traces and tick-Freeness 4
1.2 Basic Types, Traces, Failures and Divergences 7
1.3 The Process Type Invariant 7
1.4 The Abstraction to the process-Type 10
1.5 Some Consequences of the Process Characterization 14
1.6 Process Approximation is a Partial Ordering, a Cpo, and a Pcpo 16
1.7 Process Refinement is a Partial Ordering 21

2 The STOP Process Definition **26**

3 The Multi-Prefix Operator Definition **27**

4 Backpatch Isabelle 2009-1 **27**

5 The core of it . . . **27**
5.1 Well-foundedness of Mprefix 28
5.2 Projections in Prefix 28
5.3 Basic Properties 29
5.4 Proof of Continuity Rule 29
5.5 High-level Syntax 30

6 Deterministic Choice Operator Definition **31**

7 Nondeterministic Choice Operator Definition **33**

8 The Sequence Operator **34**

9 The Hiding Operator **36**

10 Toplevel Theory **40**
10.1 Refinement Proof Rules 40
10.2 The "Laws" of CSP 40

11 Infra-structure for Communication Primitives **62**

12 Operational Semantics **65**

13 Example: Refinement Example with Buffer over infinite Alphabet **65**

14 Defining the Copy-Buffer Example **65**

15 The Standard Proof **66**
15.1 Channels and Synchronization Sets 66
15.2 Definitions by Recursors 66
15.3 A Refinement Proof 67

16 An Alternative Approach: Using the fixrec-Package **67**
16.1 Channels and Synchronisation Sets 67
16.2 Process Definitions via fixrec-Package 67
16.3 Another Refinement Proof on fixrec-infrastructure 68

theory *Directed*
imports *HOLCF*
begin

0.1 Directed sets

default-sort *type*

definition *directed* :: *'a::po set* ⇒ *bool* **where**
directed S ⟷ (∃ *x*. *x* ∈ *S*) ∧ (∀ *x*∈*S*. ∀ *y*∈*S*. ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*)

lemma *directedI*:
assumes ∃ *z*. *z* ∈ *S*
assumes ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
shows *directed S*
⟨*proof*⟩

lemma *directedD1*: *directed S* ⟹ ∃ *z*. *z* ∈ *S*
⟨*proof*⟩

lemma *directedD2*: ⟦*directed S*; *x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
⟨*proof*⟩

lemma *directedE1*:
assumes *S*: *directed S*
obtains *z* **where** *z* ∈ *S*

⟨*proof*⟩

lemma *directedE2*:
assumes S: *directed* S
assumes x: $x \in S$ **and** y: $y \in S$
obtains z **where** $z \in S$ $x \sqsubseteq z$ $y \sqsubseteq z$
⟨*proof*⟩

lemma *directed-finiteI*:
assumes U: $\bigwedge U.\ [\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists\, z{\in}S.\ U <|\ z$
shows *directed* S
⟨*proof*⟩

lemma *directed-finiteD*:
assumes S: *directed* S
shows $[\![\mathit{finite}\ U;\ U \subseteq S]\!] \Longrightarrow \exists\, z{\in}S.\ U <|\ z$
⟨*proof*⟩

lemma *not-directed-empty* [*simp*]: ¬ *directed* {}
⟨*proof*⟩

lemma *directed-singleton*: *directed* $\{x\}$
⟨*proof*⟩

lemma *directed-bin*: $x \sqsubseteq y \Longrightarrow$ *directed* $\{x, y\}$
⟨*proof*⟩

lemma *directed-chain*: *chain* $S \Longrightarrow$ *directed* (*range* S)
⟨*proof*⟩

end

1 Hoare/Roscoe's Denotational Semantics for CSP The Notion of Processes

theory *Process*
imports *HOLCF Directed*
begin

⟨*ML*⟩

This is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe's Book [1],and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes", in Proceedings of the Pittsburgh semi-

nar on concurrency, Springer LNCS 197 (1985), 281-305. This work revealed minor, but omnipresent foundational errors in key concepts like the process invariant that were revealed by a first formalization in Isabelle/HOL, called HOL-CSP 1.0 [2].

In contrast to HOL-CSP 1.0, which came with an own fixpoint theory partly inspired by previous work of Franz Regensburger and developed by myself, it is the goal of this redesign of the HOL-CSP theory to reuse the HOLCF theory that emmerged from Franz'ens work. Thus, the footprint of this theory should be reduced drastically. Moreover, all proofs have been heavily revised or re-constructed to reflect the drastically improved state of the art of interactive theory development with Isabelle.

The following merely technical command has the purpose to undo a default setting of HOLCF.

default-sort *type*

1.1 Pre-Requisite: Basic Traces and tick-Freeness

The denotational semantics of CSP assumes a distinguishable special event, called `tick` and written ?, that is required to occur only in the end in order to signalize succesful termination of a process. (In the original text of Hoare, this treatment was more liberal and lead to foundational problems: the process invariant could not be established for the sequential composition operator of CSP; see [2] for details.)

datatype $'\alpha$ *event* = *ev* $'\alpha$ | *tick*

type-synonym $'\alpha$ *trace* = ($'\alpha$ *event*) *list*

We chose as standard ordering on traces the prefix ordxering.

instantiation *list* :: (*type*) *order*
begin

definition *le-list-def* : $s \leq t \longleftrightarrow (\exists\ r.\ s\ @\ r = t)$

definition *less-list-def*: $(s::'a\ list) < t \longleftrightarrow s \leq t \wedge s \neq t$

instance

⟨*proof*⟩

end

Some facts on the prefix ordering.

lemma *nil-le*[*simp*]: $[] \leq s$

⟨*proof*⟩

lemma *nil-le2*[*simp*]: $s \leq [] = (s = [])$
⟨*proof*⟩

lemma *nil-less*[*simp*]: $\neg\ t < []$
⟨*proof*⟩

lemma *nil-less2*[*simp*]: $[] < t\ @\ [a]$
⟨*proof*⟩

lemma *less-self*[*simp*]: $t < t@[a]$
⟨*proof*⟩

lemma *le-length-mono*: $s \leq t \Longrightarrow length\ s \leq length\ t$
⟨*proof*⟩

lemma *less-length-mono*: $s < t \Longrightarrow length\ s < length\ t$
⟨*proof*⟩

lemma *list-nonMt-append*:
$s \neq [] \Longrightarrow \exists\ a\ t.\ s = t\ @\ [a]$
⟨*proof*⟩

lemma *append-eq-first-pref-spec*[*rule-format*]:
$s\ @\ t = r\ @\ [x] \wedge t \neq [] \longrightarrow s \leq r$
⟨*proof*⟩

For the process invariant, it is a key element to reduce the notion of traces to traces that may only contain one tick event at the very end. This is captured by the definition of the predicate `front_tickFree` and its stronger version `tickFree`. Here is the theory of this concept.

definition *tickFree* :: $'\alpha$ *trace* ⇒ *bool*
where *tickFree s* = (*tick* ∉ *set s*)

definition *front-tickFree* :: $'\alpha$ *trace* ⇒ *bool*
where *front-tickFree s* = (*s* =[] ∨ *tickFree*(*tl*(*rev s*)))

lemma *tickFree-Nil* [*simp*]: *tickFree* []
⟨*proof*⟩

lemma *tickFree-Cons* [*simp*]: *tickFree* (*a* # *t*) = (*a* ≠ *tick* ∧ *tickFree t*)
⟨*proof*⟩

lemma *tickFree-tl* : [|*s* ~= [] ; *tickFree s*|] ==> *tickFree*(*tl s*)
⟨*proof*⟩

lemma *tickFree-append*[*simp*]: *tickFree*(*s*@*t*) = (*tickFree s* ∧ *tickFree t*)
⟨*proof*⟩

lemma *non-tickFree-tick* [*simp*]: ¬ *tickFree* [*tick*]
⟨*proof*⟩

lemma *non-tickFree-implies-nonMt*: ¬ *tickFree s* ⟹ *s* ≠ []
⟨*proof*⟩

lemma *tickFree-rev* : *tickFree*(*rev t*) = (*tickFree t*)
⟨*proof*⟩

lemma *front-tickFree-Nil*[*simp*]: *front-tickFree* []
⟨*proof*⟩

lemma *front-tickFree-single*[*simp*]:*front-tickFree* [*a*]
⟨*proof*⟩

lemma *tickFree-implies-front-tickFree*:
tickFree s ⟹ *front-tickFree s*
⟨*proof*⟩

lemma *front-tickFree-charn*:
front-tickFree s = (*s* = [] ∨ (∃ *a t*. *s* = *t* @ [*a*] ∧ *tickFree t*))
⟨*proof*⟩

lemma *front-tickFree-implies-tickFree*:
front-tickFree (*t* @ [*a*]) ⟹ *tickFree t*
⟨*proof*⟩

lemma *tickFree-implies-front-tickFree-single*:
tickFree t ⟹ *front-tickFree* (*t* @ [*a*])
⟨*proof*⟩

lemma *nonTickFree-n-frontTickFree*:
⟦¬ *tickFree s*; *front-tickFree s* ⟧ ⟹ ∃ *t*. *s* = *t* @ [*tick*]
⟨*proof*⟩

lemma *front-tickFree-dw-closed* :
front-tickFree (*s* @ *t*) ⟹ *front-tickFree s*
⟨*proof*⟩

lemma *front-tickFree-append*:
⟦ *tickFree s*; *front-tickFree t*⟧ ⟹ *front-tickFree* (*s* @ *t*)
⟨*proof*⟩

1.2 Basic Types, Traces, Failures and Divergences

type-synonym $'\alpha$ *refusal* = ($'\alpha$ *event*) *set*
type-synonym $'\alpha$ *failure* = $'\alpha$ *trace* × $'\alpha$ *refusal*
type-synonym $'\alpha$ *divergence* = $'\alpha$ *trace set*
type-synonym $'\alpha$ *process-pre* = $'\alpha$ *failure set* × $'\alpha$ *divergence*

definition *FAILURES* :: $'\alpha$ *process-pre* ⇒ ($'\alpha$ *failure set*)
where *FAILURES P* = *fst P*

definition *TRACES* :: $'\alpha$ *process-pre* ⇒ ($'\alpha$ *trace set*)
where *TRACES P* = {*tr*. ∃ *a*. *a* ∈ *FAILURES P* ∧ *tr* = *fst a*}

definition *DIVERGENCES* :: $'\alpha$ *process-pre* ⇒ $'\alpha$ *divergence*
where *DIVERGENCES P* = *snd P*

definition *REFUSALS* :: $'\alpha$ *process-pre* ⇒ ($'\alpha$ *refusal set*)
where *REFUSALS P* = {*ref*. ∃ *F*. *F* ∈ *FAILURES P* ∧ *F* = ([],*ref*)}

1.3 The Process Type Invariant

definition *is-process* :: $'\alpha$ *process-pre* ⇒ *bool* **where**
is-process P =
(([],{}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*,*X*) ∈ *FAILURES P* ⟶ *front-tickFree s*) ∧
(∀ *s t* . (*s*@*t*,{}) ∈ *FAILURES P* ⟶ (*s*,{}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*,*Y*) ∈ *FAILURES P* & *X* <= *Y* ⟶ (*s*,*X*) ∈ *FAILURES P*)
∧
(∀ *s X Y*. (*s*,*X*) ∈ *FAILURES P* ∧
(∀ *c*. *c* ∈ *Y* ⟶ ((*s*@[*c*],{})∉*FAILURES P*)) ⟶
(*s*,*X* ∪ *Y*)∈*FAILURES P*) ∧
(∀ *s X*. (*s*@[*tick*],{}) : *FAILURES P* ⟶ (*s*,*X*−{*tick*}) ∈ *FAILURES P*) ∧
(∀ *s t*. *s* ∈ *DIVERGENCES P* ∧ *tickFree s* ∧ *front-tickFree t*
⟶ *s*@*t* ∈ *DIVERGENCES P*) ∧
(∀ *s X*. *s* ∈ *DIVERGENCES P* ⟶ (*s*,*X*) ∈ *FAILURES P*) ∧
(∀ *s*. *s* @ [*tick*] : *DIVERGENCES P* ⟶ *s* ∈ *DIVERGENCES P*))

lemma *is-process-spec*:
is-process P =
(([],{}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*,*X*) ∈ *FAILURES P* ⟶ *front-tickFree s*) ∧
(∀ *s t* . (*s* @ *t*,{}) ∉ *FAILURES P* ∨ (*s*,{}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*,*Y*) ∉ *FAILURES P* ∨ ¬(*X*⊆*Y*) | (*s*,*X*) ∈ *FAILURES P*) ∧
(∀ *s X Y*.(*s*,*X*) ∈ *FAILURES P* ∧
(∀ *c*. *c* ∈ *Y* ⟶ ((*s*@[*c*],{}) ∉ *FAILURES P*)) ⟶(*s*,*X* ∪ *Y*) ∈ *FAILURES*
P) ∧
(∀ *s X*. (*s*@[*tick*],{}) ∈ *FAILURES P* ⟶ (*s*,*X* − {*tick*}) ∈ *FAILURES*
P) ∧

$(\forall\ s\ t.\ s \notin DIVERGENCES\ P \vee \neg tickFree\ s \vee \neg front\text{-}tickFree\ t$
$\vee\ s\ @\ t \in DIVERGENCES\ P) \wedge$
$(\forall\ s\ X.\ s \notin DIVERGENCES\ P \vee (s,X) \in\ FAILURES\ P) \wedge$
$(\forall\ s.\ s\ @\ [tick] \notin\ DIVERGENCES\ P \vee s \in\ DIVERGENCES\ P))$
⟨*proof*⟩

lemma *Process-eqI* :
assumes *A*: $FAILURES\ P = FAILURES\ Q$
assumes *B*: $DIVERGENCES\ P = DIVERGENCES\ Q$
shows $(P::'\alpha\ process\text{-}pre) = Q$
⟨*proof*⟩

lemma *process-eq-spec*:
$((P::'a\ process\text{-}pre) = Q) =$
$(FAILURES\ P = FAILURES\ Q \wedge DIVERGENCES\ P = DIVERGENCES\ Q)$
⟨*proof*⟩

lemma *process-surj-pair*:
$(FAILURES\ P, DIVERGENCES\ P) = P$
⟨*proof*⟩

lemma *Fa-eq-imp-Tr-eq*:
$FAILURES\ P = FAILURES\ Q \Longrightarrow TRACES\ P = TRACES\ Q$
⟨*proof*⟩

lemma *is-process1*:
$is\text{-}process\ P \Longrightarrow ([],\{\}) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process2*:
$is\text{-}process\ P \Longrightarrow \forall\ s\ X.\ (s,X) \in FAILURES\ P \longrightarrow front\text{-}tickFree\ s$
⟨*proof*⟩

lemma *is-process3*:
$is\text{-}process\ P \Longrightarrow \forall\ s\ t.\ (s\ @\ t,\{\}) \in FAILURES\ P \longrightarrow (s,\ \{\}) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process3-S-pref*:
$[\![is\text{-}process\ P;\ (t,\ \{\}) \in FAILURES\ P;\ s \leq t]\!] \Longrightarrow (s,\ \{\}) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process4*:
$is\text{-}process\ P \Longrightarrow \forall s\ X\ Y.\ (s,\ Y) \notin FAILURES\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in FAILURES\ P$

⟨*proof*⟩

lemma *is-process4-S*:
⟦*is-process P*; (*s*, *Y*) ∈ *FAILURES P*; *X* ⊆ *Y*⟧ ⟹ (*s*, *X*) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process4-S1*:
⟦*is-process P*; *x* ∈ *FAILURES P*; *X* ⊆ *snd x*⟧ ⟹ (*fst x*, *X*) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process5*:
is-process P ⟹
∀ *sa X Y*.
(*sa*, *X*) ∈ *FAILURES P* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*sa* @ [*c*], {}) ∉ *FAILURES P*)
⟶
(*sa*, *X* ∪ *Y*) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process5-S*:
⟦*is-process P*; (*sa*, *X*) ∈ *FAILURES P*;
∀ *c*. *c* ∈ *Y* ⟶ (*sa* @ [*c*], {}) ∉ *FAILURES P*⟧
⟹ (*sa*, *X* ∪ *Y*) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process5-S1*:
⟦*is-process P*; (*sa*, *X*) ∈ *FAILURES P*; (*sa*, *X* ∪ *Y*) ∉ *FAILURES P*⟧
⟹ ∃ *c*. *c* ∈ *Y* ∧ (*sa* @ [*c*], {}) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process6*:
is-process P ⟹
∀ *s X*. (*s*@[*tick*],{}) ∈ *FAILURES P* ⟶ (*s*,*X*−{*tick*}) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process6-S*:
⟦*is-process P* ;(*s*@[*tick*],{}) ∈ *FAILURES P*⟧ ⟹
(*s*,*X*−{*tick*}) ∈ *FAILURES P*
⟨*proof*⟩

lemma *is-process7*:
is-process P ⟹
∀ *s t*. *s* ∉ *DIVERGENCES P* ∨
¬ *tickFree s* ∨
¬ *front-tickFree t* ∨
s @ *t* ∈ *DIVERGENCES P*
⟨*proof*⟩

lemma *is-process7-S*:
⟦ *is-process P*;*s* : *DIVERGENCES P*;*tickFree s*;*front-tickFree t*⟧

$\Longrightarrow$ s @ $t \in$ *DIVERGENCES P*
⟨*proof*⟩

lemma *is-process8*:
is-process P $\Longrightarrow \forall$ $s\ X.\ s \notin$ *DIVERGENCES P* $\vee$ $(s,X) \in$ *FAILURES P*
⟨*proof*⟩

lemma *is-process8-S*:
⟦ *is-process P*; $s \in$ *DIVERGENCES P* ⟧ $\Longrightarrow$ (s,X) $\in$ *FAILURES P*
⟨*proof*⟩

lemma *is-process9*:
is-process P $\Longrightarrow \forall$ $s.\ s$@$[tick] \notin$ *DIVERGENCES P* $\vee$ $s \in$ *DIVERGENCES P*
⟨*proof*⟩

lemma *is-process9-S*:
⟦ *is-process P*;s@$[tick] \in$ *DIVERGENCES P* ⟧ $\Longrightarrow$ $s \in$ *DIVERGENCES P*
⟨*proof*⟩

lemma *Failures-implies-Traces*:
⟦*is-process P*; $(s,\ X) \in$ *FAILURES P*⟧ $\Longrightarrow$ $s \in$ *TRACES P*
⟨*proof*⟩

lemma *is-process5-sing*:
⟦ *is-process P* ; $(s,\{x\}) \notin$ *FAILURES P*;$(s,\{\}) \in$ *FAILURES P*⟧ $\Longrightarrow$
$(s$ @ $[x],\{\}) \in$ *FAILURES P*
⟨*proof*⟩

lemma *is-process5-singT*:
⟦ *is-process P* ; $(s,\{x\}) \notin$ *FAILURES P*;$(s,\{\}) \in$ *FAILURES P*⟧
$\Longrightarrow$ s @ $[x]$ $\in$ *TRACES P*
⟨*proof*⟩

lemma *front-trace-is-tickfree*:
⟦ *is-process P*; $(t$ @ $[tick],X) \in$ *FAILURES P*⟧ $\Longrightarrow$ *tickFree t*
⟨*proof*⟩

lemma *trace-with-Tick-implies-tickFree-front* :
⟦ *is-process P*; t @ $[tick] \in$ *TRACES P*⟧ $\Longrightarrow$ *tickFree t*
⟨*proof*⟩

1.4 The Abstraction to the process-Type

typedef (*Process*)

$'\alpha$ *process* = $\{p :: 'a$ *process-pre* . *is-process* $p\}$
⟨*proof*⟩

definition $F :: 'a$ *process* ⇒ ($'a$ *failure set*)
where $F\ P =$ *FAILURES* (*Rep-Process* P)

definition $T :: 'a$ *process* ⇒ ($'a$ *trace set*)
where $T\ P =$ *TRACES* (*Rep-Process* P)

definition $D :: 'a$ *process* ⇒ $'a$ *divergence*
where $D\ P =$ *DIVERGENCES* (*Rep-Process* P)

definition $R :: 'a$ *process* ⇒ ($'a$ *refusal set*)
where $R\ P =$ *REFUSALS* (*Rep-Process* P)

lemma *is-process-Rep* : *is-process* (*Rep-Process* P)
⟨*proof*⟩

lemma *Process-spec*: *Abs-Process*($(F\ P\ ,\ D\ P)$) = P
⟨*proof*⟩

theorem *Process-eq-spec*:
$(P = Q)$=$(F\ P = F\ Q \wedge D\ P = D\ Q)$
⟨*proof*⟩

theorem *is-processT*:
$([],\{\}) \in F\ P \wedge$
$(\forall\ s\ X.\ (s,X) \in F\ P \longrightarrow$ *front-tickFree* $s) \wedge$
$(\forall\ s\ t\ .(s@t,\{\}) \in F\ P \longrightarrow (s,\{\}) \in F\ P) \wedge$
$(\forall\ s\ X\ Y.(s,Y) \in F\ P \wedge (X \subseteq Y) \longrightarrow (s,X) \in F\ P) \wedge$
$(\forall\ s\ X\ Y.(s,X) \in F\ P \wedge (\forall\ c.\ c \in Y \longrightarrow ((s@[c],\{\}) \notin F\ P)) \longrightarrow (s,X \cup Y) \in F\ P) \wedge$
$(\forall\ s\ X.\ (s@[tick],\{\}) \in F\ P \longrightarrow (s,\ X - \{tick\}) \in F\ P) \wedge$
$(\forall\ s\ t.\ s \in D\ P \wedge$ *tickFree* $s \wedge$ *front-tickFree* $t \longrightarrow s\ @\ t \in D\ P) \wedge$
$(\forall\ s\ X.\ s \in D\ P \longrightarrow (s,X) \in F\ P) \wedge$
$(\forall\ s.\ s@[tick] \in D\ P \longrightarrow s \in D\ P)$
⟨*proof*⟩

theorem *process-charn*:
$([],\ \{\}) \in F\ P \wedge$
$(\forall s\ X.\ (s,\ X) \in F\ P \longrightarrow$ *front-tickFree* $s) \wedge$
$(\forall s\ t.\ (s\ @\ t,\ \{\}) \notin F\ P \vee (s,\ \{\}) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ Y) \notin F\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in F\ P) \wedge$
$(\forall s\ X\ Y.\ (s,\ X) \in F\ P \wedge (\forall c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin F\ P) \longrightarrow$
$(s,\ X \cup Y) \in F\ P) \wedge$

$(\forall\, s\ X.\ (s\ @\ [tick],\ \{\}) \in F\ P \longrightarrow (s,\ X - \{tick\}) \in F\ P) \wedge$
$(\forall\, s\ t.\ s \notin D\ P \vee \neg\ tickFree\ s \vee \neg\ front\text{-}tickFree\ t \vee s\ @\ t \in D\ P) \wedge$
$(\forall\, s\ X.\ s \notin D\ P \vee (s,\ X) \in F\ P) \wedge (\forall\, s.\ s\ @\ [tick] \notin D\ P \vee s \in D\ P)$
⟨*proof*⟩

split of `is_processT`:

lemma *is-processT1*: ([],{}) ∈ *F P*
⟨*proof*⟩

lemma *is-processT2*:
$\forall\, s\ X.\ (s,\ X) \in F\ P \longrightarrow front\text{-}tickFree\ s$
⟨*proof*⟩

lemma *is-processT2-TR* : $\forall\, s.\ s \in T\ P \longrightarrow front\text{-}tickFree\ s$
⟨*proof*⟩

lemma *is-proT2*:
$[\![(s,\ X) \in F\ P;\ s \neq []]\!] \Longrightarrow tick \notin set\ (tl\ (rev\ s))$
⟨*proof*⟩

lemma *is-processT3* :
$\forall\, s\ t.\ (s\ @\ t,\ \{\}) \in F\ P \longrightarrow (s,\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT3-S-pref* :
$[\![(t,\ \{\}) \in F\ P;\ s \leq t]\!] \Longrightarrow (s,\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT4* :
$\forall\, s\ X\ Y.\ (s,\ Y) \in F\ P \wedge X \subseteq Y \longrightarrow (s,\ X) \in F\ P$
⟨*proof*⟩

lemma *is-processT4-S1* :
$[\![x \in F\ P;\ X \subseteq snd\ x]\!] \Longrightarrow (fst\ x,\ X) \in F\ P$
⟨*proof*⟩

lemma *is-processT5*:
$\forall\, s\ X\ Y.(s,X) \in F\ P \wedge (\forall\, c.\ c{\in}Y \longrightarrow (s@[c],\{\}) \notin F\ P) \longrightarrow (s,X{\cup}Y){\in}F\ P$
⟨*proof*⟩

lemma *is-processT5-S1*:
$[\![(s,\ X) \in F\ P;\ (s,\ X \cup Y) \notin F\ P]\!] \Longrightarrow \exists\, c.\ c \in Y \wedge (s\ @\ [c],\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S2*:
$[\![(s,\ X) \in F\ P;\ (s\ @\ [c],\ \{\}) \notin F\ P]\!] \Longrightarrow (s,\ X \cup \{c\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S2a*:
$[\![(s,\ X) \in F\ P;\ (s,\ X \cup \{c\}) \notin F\ P]\!] \Longrightarrow (s\ @\ [c],\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S3*:
assumes *A*: $(s,\ \{\}) \in F\ P$
and *B*: $(s\ @\ [c],\ \{\}) \notin F\ P$
shows $(s,\ \{c\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S4*:
$[\![(s,\ \{\}) \in F\ P;\ (s,\ \{c\}) \notin F\ P]\!] \Longrightarrow (s\ @\ [c],\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S5*:
$[\![(s,\ X) \in F\ P;\ \forall\, c.\ c \in Y \longrightarrow (s,\ X \cup \{c\}) \notin F\ P]\!]$
$\Longrightarrow \forall\, c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT5-S6*:
$([],\ \{c\}) \notin F\ P \Longrightarrow ([c],\ \{\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT6*:
$\forall\, s\ X.\ (s\ @\ [tick],\ \{\}) \in F\ P \longrightarrow (s,\ X - \{tick\}) \in F\ P$
⟨*proof*⟩

lemma *is-processT7*:
$\forall\, s\ t.\ s \in D\ P \wedge tickFree\ s \wedge front\text{-}tickFree\ t \longrightarrow s\ @\ t \in D\ P$
⟨*proof*⟩

lemmas *is-processT7-S* =
is-processT7[*rule-format*,*OF conjI*[*THEN conjI*,
THEN conj-commute[*THEN iffD1*]]]

lemma *is-processT8*:
$\forall\, s\ X.\ s \in D\ P \longrightarrow (s,\ X) \in F\ P$
⟨*proof*⟩

lemmas *is-processT8-S* = *is-processT8*[*rule-format*]

lemma *is-processT8-Pair*: *fst s* $\in D\ P \Longrightarrow s \in F\ P$
⟨*proof*⟩

lemma *is-processT9*:
$\forall\, s.\ s\ @\ [tick] \in D\ P \longrightarrow s \in D\ P$
⟨*proof*⟩

lemma *is-processT9-S-swap*: $s \notin D\ P \Longrightarrow s\ @\ [tick] \notin D\ P$
⟨*proof*⟩

1.5 Some Consequences of the Process Characterization

lemma *no-Trace-implies-no-Failure*:
$s \notin T\ P \Longrightarrow (s,\ \{\}) \notin F\ P$
⟨*proof*⟩

lemmas *NT-NF* = *no-Trace-implies-no-Failure*

lemma *T-def-spec*:
$T\ P = \{tr.\ \exists\, a.\ a \in F\ P \wedge tr = fst\ a\}$
⟨*proof*⟩

lemma *F-T*:
$(s,\ X) \in F\ P \Longrightarrow s \in T\ P$
⟨*proof*⟩

lemma *F-T1*:
$a \in F\ P \Longrightarrow fst\ a \in T\ P$
⟨*proof*⟩

lemma *T-F*:
$s \in T\ P \Longrightarrow (s,\ \{\}) \in F\ P$
⟨*proof*⟩

lemmas *is-processT4-empty* [*elim*!]= *F-T* [*THEN T-F*]

lemma *NF-NT*:
$(s,\ \{\}) \notin F\ P \Longrightarrow s \notin T\ P$
⟨*proof*⟩

lemma *is-processT6-S1*:
$[\![\ tick \notin X; (s\ @\ [tick],\ \{\}) \in F\ P\ ]\!] \Longrightarrow (s::'a\ event\ list,\ X) \in F\ P$
⟨*proof*⟩

lemmas *is-processT3-ST* = *T-F* [*THEN is-processT3*[*rule-format*, *THEN F-T*]]

lemmas *is-processT3-ST-pref* = *T-F* [*THEN is-processT3-S-pref* [*THEN F-T*]]

lemmas *is-processT3-SR* = *F-T* [*THEN T-F* [*THEN is-processT3*[*rule-format*]]]

lemmas *D-T* = *is-processT8-S* [*THEN F-T*]

lemma *D-T-subset* : $D\ P \subseteq T\ P$ ⟨*proof*⟩

lemma *NF-ND* : $(s,\ X) \notin F\ P \Longrightarrow s \notin D\ P$
⟨*proof*⟩

lemmas *NT-ND* = *D-T-subset*[*THEN Set.contra-subsetD*]

lemma *T-F-spec* : $((t,\ \{\}) \in F\ P) = (t \in T\ P)$
⟨*proof*⟩

lemma *is-processT5-S7*:
$[\![t \in T\ P;\ (t,\ A) \notin F\ P]\!] \Longrightarrow \exists x.\ x \in A \wedge t\ @\ [x] \in T\ P$
⟨*proof*⟩

lemma *Nil-subset-T*: $\{[]\} \subseteq T\ P$
⟨*proof*⟩

lemma *Nil-elem-T*: $[] \in T\ P$
⟨*proof*⟩

lemmas *D-imp-front-tickFree* =
is-processT8-S[*THEN is-processT2*[*rule-format*]]

lemma *D-front-tickFree-subset* : $D\ P \subseteq$ *Collect front-tickFree*
⟨*proof*⟩

lemma *F-D-part*:
$F\ P = \{(s,\ x).\ s \in D\ P\} \cup \{(s,\ x).\ s \notin D\ P \wedge (s,\ x) \in F\ P\}$
⟨*proof*⟩

lemma *D-F* : $\{(s,\ x).\ s \in D\ P\} \subseteq F\ P$
⟨*proof*⟩

lemma *append-T-imp-tickFree*:

$[\![t \mathbin{@} s \in T\ P;\ s \neq []]\!] \Longrightarrow tickFree\ t$
⟨*proof*⟩

lemma *F-subset-imp-T-subset*:
$F\ P \subseteq F\ Q \Longrightarrow T\ P \subseteq T\ Q$
⟨*proof*⟩

lemmas *append-single-T-imp-tickFree* =
append-T-imp-tickFree[*of* - [*a*], *simplified*]

lemma *is-processT6-S2*:
$[\![tick \notin X;\ [tick] \in T\ P]\!] \Longrightarrow ([],\ X) \in F\ P$
⟨*proof*⟩

lemma *is-processT9-tick*:
$[\![[tick] \in D\ P;\ front\text{-}tickFree\ s]\!] \Longrightarrow s \in D\ P$
⟨*proof*⟩

lemma *T-nonTickFree-imp-decomp*:
$[\![t \in T\ P;\ \neg\ tickFree\ t]\!] \Longrightarrow \exists\, s.\ t = s \mathbin{@} [tick]$
⟨*proof*⟩

1.6 Process Approximation is a Partial Ordering, a Cpo, and a Pcpo

The Failure/Divergence Model of CSP Semantics provides two orderings: The *approximation ordering* (also called *process ordering*) will be used for giving semantics to recursion (fixpoints) over processes, the *refinement ordering* captures our intuition that a more concrete process is more deterministic and more defined than an abstract one.

We start with the key-concepts of the approximation ordering, namely the predicates *min_elems* and *Ra* (abbreviating *refusals after*). The former provides just a set of minimal elements from a given set of elements of typeclass *ord* ...

definition *min-elems* :: ('*s*::*ord*) *set* ⇒ '*s set*
where *min-elems X* = $\{s \in X.\ \forall\, t.\ t \in X \longrightarrow \neg\ (t < s)\}$

lemma *Nil-min-elems* : $[] \in A \Longrightarrow [] \in min\text{-}elems\ A$
⟨*proof*⟩

lemma *min-elems-le-self*[*simp*] : $(min\text{-}elems\ A) \subseteq A$
⟨*proof*⟩

lemmas *elem-min-elems* = *Set.set-mp*[*OF min-elems-le-self*]

lemma *min-elems-Collect-ftF-is-Nil* :
min-elems (*Collect front-tickFree*) = {[]}
⟨*proof*⟩

lemma *min-elems5* :
assumes *A*: (*x*::'*a list*) ∈ *A*
shows ∃ *s*≤*x*. *s* ∈ *min-elems A*
⟨*proof*⟩

lemma *min-elems4*:
A ≠ {} ⟹ ∃ *s*. (*s* :: '*a trace*) ∈ *min-elems A*
⟨*proof*⟩

lemma *min-elems-charn*:
t ∈ *A* ⟹ ∃ *t' r*. *t* = (*t'* @ *r*) ∧ *t'* ∈ *min-elems A*
⟨*proof*⟩

lemmas *min-elems-ex* = *min-elems-charn*

. . . while the second returns the set of possible refusal sets after a given trace s and a given process P:

definition *Ra* :: ['α *process*, 'α *trace*] ⇒ ('α *refusal set*)
where *Ra P s* = {*X*. (*s*, *X*) ∈ *F P*}

In the following, we link the process theory to the underlying fixpoint/domain theory of HOLCF by identifying the approximation ordering with HOLCF's pcpo's.

instantiation
process :: (*type*) *below*
begin

declares approximation ordering _ ⊑ _ also written _ << _.

definition *le-approx-def* : *P* ⊑ *Q* ≡ *D Q* ⊆ *D P* ∧
(∀ *s*. *s* ∉ *D P* ⟶ *Ra P s* = *Ra Q s*) ∧
min-elems (*D P*) ⊆ *T Q*

The approximation ordering captures the fact that more concrete processes should be more defined by ordering the divergence sets appropriately. For defined positions in a process, the failure sets must coincide pointwise; moreover, the minimal elements (wrt. prefix ordering on traces, i.e. lists) must be contained in the trace set of the more concrete process.

instance ⟨*proof*⟩

end

lemma *le-approx1*:
$P \sqsubseteq Q \Longrightarrow D\ Q \subseteq D\ P$
⟨*proof*⟩

lemma *le-approx2*:
$[\![\ P \sqsubseteq Q;\ s \notin D\ P]\!] \Longrightarrow (s,X) \in F\ Q = ((s,X) \in F\ P)$
⟨*proof*⟩

lemma *le-approx3*:
$P \sqsubseteq Q \Longrightarrow \mathit{min\text{-}elems}(D\ P) \subseteq T\ Q$
⟨*proof*⟩

lemma *le-approx2T*:
$[\![\ P \sqsubseteq Q;\ s \notin D\ P]\!] \Longrightarrow\ s \in T\ Q = (s \in T\ P)$
⟨*proof*⟩

lemma *le-approx-lemma-F* :
$P \sqsubseteq Q \Longrightarrow F\ Q \subseteq F\ P$
⟨*proof*⟩

lemmas *order-lemma* = *le-approx-lemma-F*

lemma *le-approx-lemma-T*:
$P \sqsubseteq Q \Longrightarrow T\ Q \subseteq T\ P$
⟨*proof*⟩

lemma *proc-ord2a* :
$[\![P \sqsubseteq Q;\ s \notin D\ P]\!] \Longrightarrow ((s,\ X) \in F\ P) = ((s,\ X) \in F\ Q)$
⟨*proof*⟩

instance
process :: (*type*) *po*
⟨*proof*⟩

At this point, we inherit quite a number of facts from the underlying HOLCF theory, which comprises a library of facts such as `chain`, `directed`(sets), upper bounds and least upper bounds, etc.

find-theorems *name*:*Porder is-lub*

Some facts from the theory of complete partial orders:

- `Porder.chainE` : $\mathit{chain}\ ?Y \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ (\mathit{Suc}\ ?i)$
- `Porder.chain_mono` : $[\![\mathit{chain}\ ?Y;\ ?i \leq ?j]\!] \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ ?j$

- `Directed.directed_chain` : $chain\ ?S \Longrightarrow directed\ (range\ ?S)$
- `Directed.directed_def` :
  $directed\ ?S = ((\exists x.\ x \in ?S) \wedge (\forall x{\in}?S.\ \forall y{\in}?S.\ \exists z{\in}?S.\ x \sqsubseteq z \wedge y \sqsubseteq z))$
- `Directed.directedD1` : $directed\ ?S \Longrightarrow \exists z.\ z \in ?S$
- `Directed.directedD2` :
  $[\![directed\ ?S;\ ?x \in ?S;\ ?y \in ?S]\!] \Longrightarrow \exists z{\in}?S.\ ?x \sqsubseteq z \wedge ?y \sqsubseteq z$
- `Directed.directedI` : $[\![\exists z.\ z \in ?S;\ \bigwedge x\ y.\ [\![x \in ?S;\ y \in ?S]\!] \Longrightarrow \exists z{\in}?S.\ x \sqsubseteq z \wedge y \sqsubseteq z]\!] \Longrightarrow directed\ ?S$
- `Porder.is_ubD` : $[\![?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?x \sqsubseteq ?u$
- `Porder.ub_rangeI` :
  $(\bigwedge i.\ ?S\ i \sqsubseteq ?x) \Longrightarrow range\ ?S <|\ ?x$
- `Porder.ub_imageD` : $[\![?f\ `\ ?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?f\ ?x \sqsubseteq ?u$
- `Porder.is_ub_upward` : $[\![?S <|\ ?x;\ ?x \sqsubseteq ?y]\!] \Longrightarrow ?S <|\ ?y$
- `Porder.is_lubD1` : $?S <<|\ ?x \Longrightarrow ?S <|\ ?x$
- `Porder.is_lubI` : $[\![?S <|\ ?x;\ \bigwedge u.\ ?S <|\ u \Longrightarrow ?x \sqsubseteq u]\!] \Longrightarrow ?S <<|\ ?x$
- `Porder.is_lub_maximal` : $[\![?S <|\ ?x;\ ?x \in ?S]\!] \Longrightarrow ?S <<|\ ?x$
- `Porder.is_lub_lub` : $?M <<|\ ?x \Longrightarrow ?M <<|\ lub\ ?M$
- `Porder.is_lub_range_shift`:
  $chain\ ?S \Longrightarrow range\ (\lambda i.\ ?S\ (i + ?j)) <<|\ ?x = range\ ?S <<|\ ?x$
- `Porder.is_lub_rangeD1`: $range\ ?S <<|\ ?x \Longrightarrow ?S\ ?i \sqsubseteq ?x$
- `Porder.lub_eqI`: $?M <<|\ ?l \Longrightarrow lub\ ?M = ?l$
- `Porder.is_lub_unique`:$[\![?S <<|\ ?x;\ ?S <<|\ ?y]\!] \Longrightarrow ?x = ?y$

definition *lim-proc* :: $('\alpha\ process)\ set \Rightarrow '\alpha\ process$
where $lim\text{-}proc\ (X) = Abs\text{-}Process\ (INTER\ X\ F,\ INTER\ X\ D)$

lemma *min-elems3*:
$[\![s\ @\ [c] \in D\ P;\ s\ @\ [c] \notin min\text{-}elems\ (D\ P)]\!] \Longrightarrow s \in D\ P$
⟨*proof*⟩

lemma *min-elems1* :

$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P]\!] \Longrightarrow s\ @\ [c] \in \mathit{min\text{-}elems}\ (D\ P)$
⟨*proof*⟩

lemma *min-elems2*:
$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow (s\ @\ [c], \{\}) \in F\ Q$
⟨*proof*⟩

lemma *min-elems6*:
$[\![s \notin D\ P;\ s\ @\ [c] \in D\ P;\ P \sqsubseteq S]\!] \Longrightarrow (s\ @\ [c], \{\}) \in F\ S$
⟨*proof*⟩

lemma *ND-F-dir2*:
$[\![s \notin D\ P;\ (s, \{\}) \in F\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow (s, \{\}) \in F\ Q$
⟨*proof*⟩

lemma *ND-F-dir2′*:
$[\![s \notin D\ P;\ s \in T\ P;\ P \sqsubseteq S;\ Q \sqsubseteq S]\!] \Longrightarrow s \in T\ Q$
⟨*proof*⟩

lemma *chain-lemma*:$[\![\mathit{chain}\ S]\!] \Longrightarrow S\ i \sqsubseteq S\ k \vee S\ k \sqsubseteq S\ i$
⟨*proof*⟩

lemma *is-process-REP-LUB*:
assumes *chain*: *chain S*
shows *is-process*(*INTER* (*range S*) *F*,*INTER* (*range S*) *D*)

⟨*proof*⟩

lemmas *Rep-Abs-LUB* = *Abs-Process-inverse*[*simplified Process-def*,
simplified, *OF is-process-REP-LUB*,
simplified]

lemma *F-LUB*: *chain S* $\Longrightarrow$ *F*(*lim-proc*(*range S*)) = *INTER* (*range S*) *F*
⟨*proof*⟩

lemma *D-LUB*: *chain S* $\Longrightarrow$ *D*(*lim-proc*(*range S*)) = *INTER* (*range S*) *D*
⟨*proof*⟩

lemma *T-LUB*: *chain S* $\Longrightarrow$ *T*(*lim-proc*(*range S*)) = *INTER* (*range S*) *T*
⟨*proof*⟩

schematic-lemma *D-LUB-2*: *chain S* $\Longrightarrow$ $t \in D(\textit{lim-proc}(\textit{range } S)) =$ *?X*
⟨*proof*⟩

schematic-lemma *T-LUB-2*: *chain S* $\Longrightarrow$ $(t \in T\ (\textit{lim-proc}\ (\textit{range } S))) =$ *?X*
⟨*proof*⟩

instance
process :: (*type*) *cpo*
⟨*proof*⟩

instance
process :: (*type*) *pcpo*
⟨*proof*⟩

1.7 Process Refinement is a Partial Ordering

The following type instantiation declares the refinement order $\_ \leq \_$ written `_ <= _`. It captures the intuition that more concrete processes should be more deterministic and more defined.

instantiation
process :: (*type*) *ord*
begin

definition *le-ref-def* : $P \leq Q \equiv D\ Q \subseteq D\ P \wedge F\ Q \subseteq F\ P$

definition *less-ref-def* : $(P::'a\ \textit{process}) < Q \equiv P \leq Q \wedge P \neq Q$

instance ⟨*proof*⟩

end

lemma *le-approx-implies-le-ref*: $(P::'\alpha\ \textit{process}) \sqsubseteq Q \Longrightarrow P \leq Q$
⟨*proof*⟩

lemma *le-ref1*: $P \leq Q \Longrightarrow D\ Q \subseteq D\ P$
⟨*proof*⟩

lemma *le-ref2*: $P \leq Q \Longrightarrow F\ Q \subseteq F\ P$
⟨*proof*⟩

lemma *le-ref2T* : $P \leq Q \Longrightarrow T\ Q \subseteq T\ P$
⟨*proof*⟩

instance *process* :: (*type*) *order*
⟨*proof*⟩

lemma *lim-proc-is-ub*:*chain S* ⟹ *range S* <| *lim-proc* (*range S*)
⟨*proof*⟩

lemma *lim-proc-is-lub1*:
chain S ⟹ ∀ *u* . (*range S* <| *u* ⟶ *D u* ⊆ *D* (*lim-proc* (*range S*)))
⟨*proof*⟩

lemma *lim-proc-is-lub2*:
chain S ⟹ ∀ *u* . *range S* <| *u* ⟶ (∀ *s*. *s* ∉ *D* (*lim-proc* (*range S*))
⟶ *Ra* (*lim-proc* (*range S*)) *s* = *Ra u s*)
⟨*proof*⟩

lemma *legacy-imp-conj*: (*P* −−> *Q* & *R′*) = ((*P* −−> *Q*) & (*P* −−> *R′*))
⟨*proof*⟩

lemma *legacy-all-conj-distr*: (! *x*. *p x* & *q x*) = ((! *x*. *p x*) & (! *x*. *q x*))
⟨*proof*⟩

lemma *legacy-INTER-def*: *INTER A B* == {*y*. ! *x*:*A*. *y* : *B x*}
⟨*proof*⟩

lemma *lim-proc-is-lub3*:
assumes *A*: *directed X*
shows ∀ *u*. *X* <| *u* ⟶ *min-elems*(*D*(*lim-proc X*)) ⊆ *T u*
⟨*proof*⟩

lemma *limproc-is-lub*: *chain S* ⟹ *range S* <<| *lim-proc* (*range S*)
⟨*proof*⟩

lemma *limproc-is-thelub*: *chain S* ⟹ *Lub S* = *lim-proc* (*range S*)
⟨*proof*⟩

end

theory *Bot*
imports *Process*
begin

definition *Bot* :: $'\alpha$ *process*
where *Bot* ≡ *Abs-Process* ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})

lemma *is-process-REP-Bot* :
is-process ({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})
⟨*proof*⟩

lemma *Rep-Abs-Bot* :*Rep-Process* (*Abs-Process* ({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})) =
({(*s*,*X*). *front-tickFree s*},{*d*. *front-tickFree d*})
⟨*proof*⟩

lemma *F-Bot*[*simp*]: *F Bot* = {(*s*,*X*). *front-tickFree s*}
⟨*proof*⟩

lemma *D-Bot*[*simp*]: *D Bot* = {*d*. *front-tickFree d*}
⟨*proof*⟩

lemma *T-Bot*[*simp*]: *T Bot* = {*s*. *front-tickFree s*}
⟨*proof*⟩

This is the key result: ⊥ — which we know to exist from the process instantiation — is equal Bot .

lemma *Bot-is-UU*: *Bot* = ⊥
⟨*proof*⟩

lemma *F-UU*[*simp*]: *F* ⊥ = {(*s*,*X*). *front-tickFree s*}
⟨*proof*⟩

lemma *D-UU*[*simp*]: *D* ⊥ = {*d*. *front-tickFree d*}
⟨*proof*⟩

lemma *T-UU*[*simp*]: *T* ⊥ = {*s*. *front-tickFree s*}
⟨*proof*⟩

end

theory *Skip*
imports *Process*

begin

definition *SKIP* :: *'a process*
where *SKIP* ≡ *Abs-Process* ({(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})

lemma *is-process-REP-Skip*:
is-process ({(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})
⟨*proof*⟩

lemma *is-process-REP-Skip2*:
is-process ({[]} × {*X*. *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}, {})
⟨*proof*⟩

lemmas *process-prover* = *Process-def Abs-Process-inverse*
FAILURES-def TRACES-def
DIVERGENCES-def is-process-REP-Skip

lemma *F-SKIP*:
F SKIP = {(*s*, *X*). *s* = [] ∧ *tick* ∉ *X*} ∪ {(*s*, *X*). *s* = [*tick*]}
⟨*proof*⟩

lemma *D-SKIP*: *D SKIP* = {}
⟨*proof*⟩

lemma *T-SKIP*: *T SKIP* ={[],[*tick*]}
⟨*proof*⟩

end

theory *Legacy*
imports *Process*
begin

lemmas *tF-Nil* = *tickFree-Nil*
lemmas *tF-Cons* = *tickFree-Cons*
lemmas *NtF-tick* = *non-tickFree-tick*
lemmas *tF-rev* = *tickFree-rev*
lemmas *ftF-Nil* = *front-tickFree-Nil*
lemmas *tF-imp-ftF* = *tickFree-implies-front-tickFree*
lemmas *ftF-imp-f-is-tF* = *front-tickFree-implies-tickFree*
lemmas *NtF-ftF-ex* = *nonTickFree-n-frontTickFree*

lemmas *Nconj-eq-disjN* = *HOL.nnf-simps*(*1*)
lemmas *Ndisj-eq-conjN* = *HOL.nnf-simps*(*2*)
lemmas *imp-disj* = *HOL.nnf-simps*(*3*)
lemmas *conj-imp* = *HOL.imp-conjL*
lemmas *Pair-fst-snd-eq* = *surjective-pairing*
lemmas *t-F-T* = *Failures-implies-Traces*
lemmas *f-F-is-tF* = *front-trace-is-tickfree*
lemmas *f-T-is-tF* = *trace-with-Tick-implies-tickFree-front*
lemmas *D-ftF-subset* = *D-front-tickFree-subset*
lemmas *append-T-tF* = *append-T-imp-tickFree*
lemmas *T-tF* = *append-single-T-imp-tickFree*
lemmas *T-tF1* = *append-single-T-imp-tickFree*
lemmas *T-NtF-ex* = *T-nonTickFree-imp-decomp*

definition *member* :: *'a list* ⇒ *'a* ⇒ *bool* **where**
mem-iff [*code-post*]: *member xs x* ⟷ *x* ∈ *set xs*

lemmas *is-process3-S* = *is-process3* [*rule-format*]
lemmas *is-process2-S* = *is-process2* [*THEN spec*, *THEN spec*, *THEN mp*]
lemmas *ProcessT-eqI* = *Process-eq-spec*[*THEN iffD2*,*OF conjI*]
lemmas *is-processT-spec* = *process-charn*
lemmas *is-processT2-TR-S* = *is-processT2-TR*[*rule-format*]
lemmas *is-processT2-S* = *is-processT2*[*rule-format*]
lemmas *is-processT3-S* = *is-processT3*[*rule-format*]
lemmas *is-processT4-S* = *is-processT4*[*rule-format*]
lemmas *is-processT5-S* = *is-processT5*[*rule-format*, *OF conjI*]
lemmas *is-processT6-S* = *is-processT6*[*rule-format*]
lemmas *is-processT9-S* = *is-processT9* [*rule-format*]
lemmas *subsetND* = *Set.contra-subsetD*
lemmas *D-ftF* = *D-imp-front-tickFree*
lemmas *ftF-imp-f-is-tF1* = *front-tickFree-implies-tickFree*

lemmas *less-eq-process-def* = *Process.le-ref-def*

lemma *Collect-eq-spec*:
{*x*. *P x*} = {*x*. *Q x*} = (∀ *x*. *P x* = *Q x*)
⟨*proof*⟩

lemmas *subset-spec* = *subset-iff*[*THEN iffD1*,*rule-format*]

lemmas *rec-ord-implies-ref-ord* = *le-approx-implies-le-ref*

lemmas *process-ref-ord-def = Process.le-ref-def*

lemmas *sq-eq-process = le-approx-def*
lemmas *process-ord-def = sq-eq-process*

lemmas *proc-ord1=le-approx1*
lemmas *proc-ord2=le-approx2*
lemmas *proc-ord3=le-approx3*
lemmas *proc-ord2T=le-approx2T*
lemmas *proc-ord-lemma-F=le-approx-lemma-F*
lemmas *proc-ord-lemma-T=le-approx-lemma-T*

lemmas *le-approx-implies-ref-ord = le-approx-implies-le-ref*
lemmas *ref-ord1 = le-ref1*
lemmas *ref-ord2 = le-ref2*
lemmas *ref-ord2T = le-ref2T*

end

2 The STOP Process Definition

theory *Stop*
imports *Process Legacy*
begin

definition *STOP* :: *'α process*
where *STOP* ≡ *Abs-Process* ({(*s*, *X*). *s* = []}, {})

lemma *is-process-REP-STOP*: *is-process* ({(*s*, *X*). *s* = []},{})
⟨*proof*⟩

lemma *Rep-Abs-STOP* : *Rep-Process* (*Abs-Process* ({(*s*, *X*). *s* = []},{})) = ({(*s*, *X*). *s* = []},{})
⟨*proof*⟩

lemma *F-STOP* : *F STOP* = {(*s*,*X*). *s* = []}
⟨*proof*⟩

lemma *D-STOP*: $D\ STOP = \{\}$
⟨*proof*⟩

lemma *T-STOP*: $T\ STOP = \{[]\}$
⟨*proof*⟩

end

3 The Multi-Prefix Operator Definition

theory *Mprefix*
imports *Process Legacy*
begin

4 Backpatch Isabelle 2009-1

definition
$contlub :: ('a::cpo \Rightarrow 'b::cpo) \Rightarrow bool$ — first cont. def **where**
$contlub\ f = (\forall\ Y.\ chain\ Y \longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i)))$

lemma *contlubE*:
$[\![contlub\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i))$
⟨*proof*⟩

lemma *monocontlub2cont*: $[\![monofun\ f;\ contlub\ f]\!] \Longrightarrow cont\ f$
⟨*proof*⟩

lemma *contlubI*:
$(\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup\ i.\ Y\ i) = (\bigsqcup\ i.\ f\ (Y\ i))) \Longrightarrow contlub\ f$
⟨*proof*⟩

lemma *cont2contlub*: $cont\ f \Longrightarrow contlub\ f$
⟨*proof*⟩

5 The core of it . . .

definition *Mprefix* :: ['a set,'a => 'a process] => 'a process **where**
Mprefix A P ≡ *Abs-Process*(
$\{(tr,ref).\ tr = [] \wedge ref\ Int\ (ev\ `\ A) = \{\}\} \cup$

$\{(tr,ref).\ tr \neq [] \wedge hd\ tr \in (ev\ `\ A) \wedge$
$(\exists\ a.\ ev\ a = (hd\ tr) \wedge (tl\ tr,ref) \in F(P\ a))\},$
$\{d.\ d \neq [] \wedge\ hd\ d\ \in (ev\ `\ A) \wedge$

$(\exists\ a.\ ev\ a = hd\ d \wedge tl\ d \in D(P\ a))\})$

syntax(*HOL*)
@*mprefix* :: [*pttrn*,'*a set*,'*a process*]=>'*a process* ((*3*[−]- : - −> -)[*0*,*0*,*64*]*64*)

syntax(*xsymbols*)
@*mprefix* :: [*pttrn*,'*a set*,'*a process*]=>'*a process* ((*3*□ - ∈ - → -)[*0*,*0*,*64*]*64*)

translations
□ $x \in A \to P$ == *CONST Mprefix A* (% x . P)

5.1 Well-foundedness of Mprefix

lemma *is-process-REP-Mp* :
is-process ({(*tr*,*ref*). *tr*=[] ∧ *ref* ∩ (*ev* ' *A*) = {}} ∪
{(*tr*,*ref*). *tr* ≠ [] ∧ *hd tr* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = (*hd tr*) ∧ (*tl tr*,*ref*) ∈ *F*(*P a*))},
{*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))})
(**is** *is-process*(*?f*, *?d*))
⟨*proof*⟩

lemma *Rep-Abs-Mp* :
assumes *H1* : f = {(*tr*, *ref*). *tr* = [] ∧ *ref* ∩ *ev* ' *A* = {}} ∪
{(*tr*, *ref*). *tr* ≠ [] ∧ *hd tr* ∈ *ev* ' *A*
∧ (∃ *a*. *ev a* = *hd tr* ∧ (*tl tr*, *ref*) ∈ *F* (*P a*))}
and *H2* : d = {*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))}
shows *Rep-Process* (*Abs-Process* (f,d)) = (f,d)
⟨*proof*⟩

5.2 Projections in Prefix

lemma *F-Mprefix* :
F(□ $x \in A \to P\ x$) = {(*tr*,*ref*). *tr*=[] ∧ *ref* ∩ (*ev* ' *A*) = {}} ∪
{(*tr*,*ref*). *tr* ≠ [] ∧ *hd tr* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = (*hd tr*) ∧ (*tl tr*,*ref*) ∈ *F*(*P a*))}
⟨*proof*⟩

lemma *D-Mprefix*:
D(□ $x \in A \to P\ x$) = {*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ' *A*) ∧
(∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ *D*(*P a*))}
⟨*proof*⟩

lemma *T-Mprefix*:
T(□ $x \in A \to P\ x$)={*s*. *s*=[] ∨ (∃ *a*. *a*∈*A* ∧ *s*≠[] ∧ *hd s* = *ev a* ∧ *tl s* ∈ *T*(*P a*))}

⟨*proof*⟩

5.3 Basic Properties

lemma *tick-T-Mprefix* [*simp*]: [*tick*] ∉ *T*(□ *x* ∈ *A* → *P x*)
⟨*proof*⟩

lemma *Nil-Nin-D-Mprefix* [*simp*]: [] ∉ *D*(□ *x* ∈ *A* → *P x*)
⟨*proof*⟩

5.4 Proof of Continuity Rule

lemma *mono-Mprefix1*:
∀ *a*. *P a* ⊑ *Q a* ⟹ *D* (*Mprefix A Q*) ⊆ *D* (*Mprefix A P*)
⟨*proof*⟩

lemma *mono-Mprefix2*:
∀ *x*. *P x* ⊑ *Q x* ⟹
∀ *s*. *s* ∉ *D* (*Mprefix A P*) ⟶ *Ra* (*Mprefix A P*) *s* = *Ra* (*Mprefix A Q*) *s*
⟨*proof*⟩

lemma *mono-Mprefix3* :
∀ *x*. *P x* ⊑ *Q x* ⟹ *min-elems* (*D* (*Mprefix A P*)) ⊆ *T* (*Mprefix A Q*)
⟨*proof*⟩

lemma *mono-Mprefix0*:
∀ *x*. *P x* ⊑ *Q x* ⟹ *Mprefix A P* ⊑ *Mprefix A Q*
⟨*proof*⟩

lemma *mono-Mprefix* : *monofun*(*Mprefix A*)
⟨*proof*⟩

lemma *contlub-Mprefix* : *contlub*(*Mprefix A*)
⟨*proof*⟩

lemma *cont-revert2cont-pointwise*:
⋀*x*. *cont* (*f x*) ⟹ *cont* (λ*x y*. *f y x*)
⟨*proof*⟩

lemma *Mprefix-cont* [*simp*]:
(⋀*x*. *cont* (*f x*)) ⟹ *cont* (λ*y*. □ *z* ∈ *A* → *f z y*)
⟨*proof*⟩

lemmas *proc-ord1D* = *proc-ord1* [*THEN subsetD*]

lemmas *proc-ord2b = proc-ord2a [THEN sym]*
lemmas *le-fun-def = Fun-Cpo.below-fun-def*
lemmas *cont-compose1 = Cont.cont-compose*
lemmas *mono-contlub-imp-cont = monocontlub2cont*

5.5 High-level Syntax

definition *read* :: [*'a=>'b,'a set, 'a => 'b process*] => *'b process*
where *read c A P ≡ Mprefix(c ‘ A) (P o (inv c))*
definition *write* :: [*'a=>'b, 'a, 'b process*] => *'b process*
where *write c a P ≡ Mprefix {c a} (λ x. P)*
definition *write0* :: [*'a, 'a process*] => *'a process*
where *write0 a P ≡ Mprefix {a} (λ x. P)*

syntax
-read :: [*id, pttrn, 'a process*] => *'a process*
((*3-‘?‘- /→ -) [0,0,28] 28*)
-readX :: [*id, pttrn, bool,'a process*] => *'a process*
((*3-‘?‘-‘|‘- /→ -) [0,0,28] 28*)
-readS :: [*id, pttrn, 'b set,'a process*] => *'a process*
((*3-‘?‘-‘:‘- /→ -) [0,0,28] 28*)

-write :: [*id, 'b, 'a process*] => *'a process*
((*3-‘!‘- /→ -) [0,0,28] 28*)
-writeS :: [*'a, 'a process*] => *'a process*
((*3- /→ -) [0,28] 28*)

translations
-read c p P == *CONST read c CONST UNIV (λp. P)*
-write c p P == *CONST write c p P*
-readX c p b P => *CONST read c {p. b} (λp. P)*
-writeS a P == *CONST write0 a P*

lemma *read-cont*[*simp*]:
(⋀*x. cont (f x)*) ⟹ *cont (λy. c‘?‘x → f x y)*
⟨*proof*⟩

lemma *write-cont*[*simp*]:
(⋀*x. cont (P::('b::cpo ⇒ 'a process))*)
⟹ *cont*(*λx. (c ‘!‘ a → P x)*)
⟨*proof*⟩

lemma *write0-cont*[*simp*]:
cont (P::('b::cpo ⇒ 'a process))
⟹ *cont*(*λx. (a → P x)*)

⟨*proof*⟩

end

6 Deterministic Choice Operator Definition

theory *Det*
imports *Process*
begin

definition
det :: [′α *process*,′α *process*] ⇒ ′α *process* (**infixl** [+] *18*)
where $P\ [+]\ Q \equiv$ *Abs-Process*($\{(s,X).\ s = [] \wedge (s,X) \in F\ P \cap F\ Q\}$
$\cup \{(s,X).\ s \neq [] \wedge (s,X) \in F\ P \cup F\ Q\}$
$\cup \{(s,X).\ s = [] \wedge s \in D\ P \cup D\ Q\}$
$\cup \{(s,X).\ s = [] \wedge tick \notin X \wedge [tick] \in T\ P \cup T\ Q\}$,
$D\ P \cup D\ Q$)

notation(*xsymbols*)
det (**infixl** □ *18*)

lemma *is-process-REP-D*:
is-process ($\{(s,\ X).\ s = [] \wedge (s,\ X) \in F\ P \cap F\ Q\} \cup$
$\{(s,\ X).\ s \neq [] \wedge (s,\ X) \in F\ P \cup F\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge s \in D\ P \cup D\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge tick \notin X \wedge [tick] \in T\ P \cup T\ Q\}$,
$D\ P \cup D\ Q$)
⟨*proof*⟩

lemma *Rep-Abs-D*:
Rep-Process
(*Abs-Process*
($\{(s,\ X).\ s = [] \wedge (s,\ X) \in F\ P \cap F\ Q\} \cup$
$\{(s,\ X).\ s \neq [] \wedge (s,\ X) \in F\ P \cup F\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge s \in D\ P \cup D\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge tick \notin X \wedge [tick] \in T\ P \cup T\ Q\}$,
$D\ P \cup D\ Q$)) =
($\{(s,\ X).\ s = [] \wedge (s,\ X) \in F\ P \cap F\ Q\} \cup$
$\{(s,\ X).\ s \neq [] \wedge (s,\ X) \in F\ P \cup F\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge s \in D\ P \cup D\ Q\} \cup$
$\{(s,\ X).\ s = [] \wedge tick \notin X \wedge [tick] \in T\ P \cup T\ Q\}$,
$D\ P \cup D\ Q$)

⟨*proof*⟩

lemma *F-det* :
$F(P \,\Box\, Q) = \{(s,X).\ s = [] \wedge (s,X) \in F\ P \cap F\ Q\}$
$\cup \{(s,X).\ s \neq [] \wedge (s,X) \in F\ P \cup F\ Q\}$
$\cup \{(s,X).\ s = [] \wedge s \in D\ P \cup D\ Q\}$
$\cup \{(s,X).\ s = [] \wedge tick \notin X \wedge [tick] \in T\ P \cup T\ Q\}$
⟨*proof*⟩

lemma *D-det*: $D(P \,\Box\, Q) = D\ P \cup D\ Q$
⟨*proof*⟩

lemma *T-det*: $T(P \,\Box\, Q) = T\ P \cup T\ Q$
⟨*proof*⟩

lemma *Det-commute*:$(P \,\Box\, Q) = (Q \,\Box\, P)$
⟨*proof*⟩

lemma *mono-D1*: $P \sqsubseteq Q \Longrightarrow D\ (Q \,\Box\, S) \subseteq D\ (P \,\Box\, S)$
⟨*proof*⟩

lemma *mono-D2*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin D\ (P \,\Box\, S) \longrightarrow Ra\ (P \,\Box\, S)\ s = Ra\ (Q \,\Box\, S)\ s)$
⟨*proof*⟩

lemma *mono-D3* : $P \sqsubseteq Q \Longrightarrow min\text{-}elems\ (D\ (P \,\Box\, S)) \subseteq T\ (Q \,\Box\, S)$
⟨*proof*⟩

lemma *mono-Det* : $P \sqsubseteq Q \Longrightarrow (P \,\Box\, S) \sqsubseteq (Q \,\Box\, S)$
⟨*proof*⟩

lemma *mono-Det-sym* : $P \sqsubseteq Q \Longrightarrow (S \,\Box\, P) \sqsubseteq (S \,\Box\, Q)$
⟨*proof*⟩

lemma *all-conj-distrib1*: $((\forall x.\ P\ x) \wedge (\forall x.\ Q\ x)) = (\forall x.\ P\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *all-conj-distrib2*: $((\forall x.\ P\ x) \wedge Q) = (\forall x.\ P\ x \wedge Q)$
⟨*proof*⟩

lemma *all-conj-distrib3*: $(P \wedge (\forall x.\ Q\ x)) = (\forall x.\ P \wedge Q\ x)$
⟨*proof*⟩

lemma *all-disj-distrib2*: $((\forall x.\ P\ x) \vee Q) = (\forall x.\ P\ x \vee Q)$

⟨*proof*⟩

lemma *all-disj-distrib3*: $(P \vee (\forall x.\ Q\ x)) = (\forall x.\ P \vee Q\ x)$
⟨*proof*⟩

lemma *cont-D* : *chain* $Y \Longrightarrow ((\bigsqcup\ i.\ Y\ i) \sqsubset S) = (\bigsqcup\ i.\ (Y\ i \sqsubset S))$
⟨*proof*⟩

lemma *cont-D'* :
assumes *chain*:*chain* Y
shows $((\bigsqcup\ i.\ Y\ i) \sqsubset S) = (\bigsqcup\ i.\ (Y\ i \sqsubset S))$
⟨*proof*⟩

lemma *det-cont*:
assumes f:*cont* f
and g:*cont* g
shows *cont* $(\lambda x.\ f\ x \sqsubset g\ x)$
⟨*proof*⟩

end

7 Nondeterministic Choice Operator Definition

theory *Ndet*
imports *Process Cont*
begin

definition
ndet :: [$'\alpha$ *process*,$'\alpha$ *process*] ⇒ $'\alpha$ *process* (**infixl** |−| *16*)
where P |−| $Q \equiv$ *Abs-Process*$(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$

notation(*xsymbols*)
ndet (**infixl** ⊓ *16*)

lemma *is-process-REP-ND*:
is-process $(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$
⟨*proof*⟩

lemma *Rep-Abs-ND*:
Rep-Process(*Abs-Process*$(F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)) = (F\ P \cup F\ Q\ ,\ D\ P \cup D\ Q)$
⟨*proof*⟩

lemma *F-ndet* : $F(P \sqcap Q) = F\ P \cup F\ Q$

⟨*proof*⟩

lemma *D-ndet* : $D(P \sqcap Q) = D\ P \cup D\ Q$
⟨*proof*⟩

lemma *T-ndet* : $T(P \sqcap Q) = T\ P \cup T\ Q$
⟨*proof*⟩

lemma *Ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
⟨*proof*⟩

lemma *mono-Ndet1*: $P \sqsubseteq Q \Longrightarrow D\ (Q \sqcap S) \subseteq D\ (P \sqcap S)$
⟨*proof*⟩

lemma *mono-Ndet2*: $P \sqsubseteq Q \Longrightarrow (\forall\ s.\ s \notin D\ (P \sqcap S) \longrightarrow Ra\ (P \sqcap S)\ s = Ra\ (Q \sqcap S)\ s)$
⟨*proof*⟩

lemma *mono-Ndet3*: $P \sqsubseteq Q \Longrightarrow$ *min-elems* $(D\ (P \sqcap S)) \subseteq T\ (Q \sqcap S)$
⟨*proof*⟩

lemma *mono-Ndet* : $P \sqsubseteq Q \Longrightarrow (P \sqcap S) \sqsubseteq (Q \sqcap S)$
⟨*proof*⟩

lemma *mono-Ndet-sym* : $P \sqsubseteq Q \Longrightarrow (S \sqcap P) \sqsubseteq (S \sqcap Q)$
⟨*proof*⟩

lemma *cont-Ndet1*:
assumes *chain*:*chain Y*
shows $((\bigsqcup\ i.\ Y\ i) \sqcap S) = (\bigsqcup\ i.\ (Y\ i \sqcap S))$
⟨*proof*⟩

lemma *ndet-cont*:
assumes *f*: *cont f*
and *g*: *cont g*
shows *cont* $(\lambda x.\ f\ x \sqcap g\ x)$
⟨*proof*⟩

end

8 The Sequence Operator

theory *Seq*

imports *Process*

begin

definition *seq* :: [*'a process*,*'a process*] => *'a process* (**infixl** *';'* *24*)
where *P ';' Q* ≡ *Abs-Process*
({(*t*, *X*). (*t*, *X* ∪ {*tick*}) ∈ *F P* ∧ *tickFree t*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ (*t2*, *X*) ∈ *F Q*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*},
{*t1* @ *t2* |*t1 t2*. *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{*t1* @ *t2* |*t1 t2*. *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*})

axioms
F-seq : *F*(*P ';' Q*) = {(*t*, *X*). (*t*, *X* ∪ {*tick*}) ∈ *F P* ∧ *tickFree t*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ (*t2*, *X*) ∈ *F Q*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{(*t*, *X*). ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*}

D-seq : *D*(*P ';' Q*) = {*t1* @ *t2* |*t1 t2*. *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{*t1* @ *t2* |*t1 t2*. *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*}

T-seq : *T*(*P ';' Q*) = {*t*. ∃ *X*. (*t*, *X* ∪ {*tick*}) ∈ *F P* ∧ *tickFree t*} ∪ (∗ *REALLY ???* ∗)
{*t*. ∃ *t1 t2*. *t* = *t1* @ *t2* ∧ *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *T Q*} ∪
{*t1* @ *t2* |*t1 t2*. *t1* ∈ *D P* ∧ *tickFree t1* ∧ *front-tickFree t2*} ∪
{*t1* @ *t2* |*t1 t2*. *t1* @ [*tick*] ∈ *T P* ∧ *t2* ∈ *D Q*}

seq-cont[*simp*]: ⟦*cont f*; *cont g*⟧ ⟹ *cont* (λ *x*. *f x ';' g x*)

lemma *is-processT1-SEQ*: ([],{}) : {(*t*,*X*). (*t*, *X Un* {*tick*}) : *F P* & *tickFree t*} *Un* {(*t*,*X*). *? t1 t2*. *t* = *t1* @ *t2* & *t1* @ [*tick*] : *T P* & (*t2*, *X*) : *F Q*} *Un* {(*t*,*X*). *? t1 t2*. *t* = *t1*@*t2* & *t1* : *D P* & *tickFree t1* & *front-tickFree t2*} *Un* {(*t*,*X*). *? t1 t2*. *t* = *t1*@*t2* & *t1*@[*tick*] : *T P* & *t2* : *D Q*}
⟨*proof*⟩

lemma *is-processT2-SEQ*: ! *s X*. (*s*, *X*) :{(*t*,*X*). (*t*, *X Un* {*tick*}) : *F P* & *tickFree*

$t\}$ *Un* $\{(t,X).$ *? t1 t2.* $t =$ *t1 @ t2 &* $t1 \in [tick] : T\ P\ \&\ (t2,\ X) : F\ Q\}$ *Un*
$\{(t,X).$ *? t1 t2.* $t =$ *t1@t2 & t1* $: D\ P$ *& tickFree t1 & front-tickFree t2*$\}$ *Un*
$\{(t,X).$ *? t1 t2.* $t =$ *t1@t2 &* $t1@[tick] : T\ P\ \&\ t2 : D\ Q\} --> front\text{-}tickFree\ s$
⟨*proof*⟩

lemma *F-D-SEQ-spec*: $F\ (P\ ;'\ Q) =$
$\{(t,\ X).\ (t,\ X \cup \{tick\}) \in F\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in [tick] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ x).\ t \in D\ (P\ ;'\ Q)\}$
⟨*proof*⟩

lemma *F-SEQ-spec*: $F\ (P\ ;'\ Q) =$
$\{(t,\ X).\ (t,\ X \cup \{tick\}) \in F\ P \wedge tickFree\ t\} \cup$
$\{(t,\ X).\ \exists t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in [tick] \in T\ P \wedge (t2,\ X) \in F\ Q\} \cup$
$\{(t,\ x).\ \exists t1\ t2.\ t = t1\ @\ t2 \wedge t1 \in D\ P \wedge tickFree\ t1 \wedge front\text{-}tickFree\ t2\}$
⟨*proof*⟩

end

9 The Hiding Operator

theory *Hide*
imports *Process*
begin

primrec *trace-hide* :: $['\alpha\ trace, ('\alpha\ event)\ set] => '\alpha\ trace$ **where**
trace-hide $[]\ A = []$
| *trace-hide* $(x\ \#\ s)\ A =$ (*if* $x \in A$
then trace-hide s A
else $x\ \#$ (*trace-hide s A*))

definition *IsChainOver* :: $[nat => '\alpha\ list, '\alpha\ list] => bool$
(**infixl** *IsChainOver 70*) **where**
f *IsChainOver* $t = (f\ 0 = t\ \wedge (\forall\ i.\ f\ i < f\ (Suc\ i)))$

definition *CongruentModuloHide* :: $[nat => '\alpha\ trace, '\alpha\ trace\ ,\ '\alpha\ set] => bool$
(- *Congruent - ModuloHide - 70*) **where**
f *Congruent t ModuloHide A* $\equiv$
$\forall\ i.$ *trace-hide* $(f\ i)\ (ev\ '\ A) =$ *trace-hide t* $(ev'A)$

definition
hiding :: [$'\alpha$ *process* ,$'\alpha$ *set*] => $'\alpha$ *process* (- \ - [*73*,*72*] *72*) **where**
$P \setminus A \equiv$ *Abs-Process*({(*s*,*X*). $\exists$ *t*. *s* = *trace-hide t* (*ev*‘*A*) $\wedge$ (*t*,*X* $\cup$ (*ev*‘*A*)) $\in$ *F P*} $\cup$
{(*s*,*X*). $\exists$ *t u*. *front-tickFree u* $\wedge$ *tickFree t* $\wedge$
s = *trace-hide t* (*ev*‘*A*) @ *u* $\wedge$
(*t* $\in$ *D P* $\vee$ ($\exists$ *f*. (*f IsChainOver t*) $\wedge$
(*f Congruent t ModuloHide A*) $\wedge$
($\forall$ *i*. *f i* $\in$ *T P*)))},
{*s*. $\exists$ *t u*. *front-tickFree u* $\wedge$
tickFree t $\wedge$ *s* = *trace-hide t* (*ev*‘*A*) @ *u* $\wedge$
(*t* $\in$ *D P* $\vee$ ($\exists$ *f*. (*f IsChainOver t*) $\wedge$
(*f Congruent t ModuloHide A*) $\wedge$
($\forall$ *i*. *f i* $\in$ *T P*)))})

axioms
F-hiding : *F*(*P* \ *A*) = {(*s*,*X*). $\exists$ *t*. *s* = *trace-hide t* (*ev*‘*A*) $\wedge$ (*t*,*X* $\cup$ (*ev*‘*A*)) $\in$ *F P*} $\cup$
{(*s*,*X*). $\exists$ *t u*. *front-tickFree u* $\wedge$ *tickFree t* $\wedge$
s = *trace-hide t* (*ev*‘*A*) @ *u* $\wedge$
(*t* $\in$ *D P* $\vee$ ($\exists$ *f*. (*f IsChainOver t*) $\wedge$
(*f Congruent t ModuloHide A*) $\wedge$
($\forall$ *i*. *f i* $\in$ *T P*))) }

D-hiding : *D*(*P* \ *A*) = {*s*. $\exists$ *t u*. *front-tickFree u* $\wedge$ *tickFree t* $\wedge$
s = *trace-hide t* (*ev*‘*A*) @ *u* $\wedge$
(*t* $\in$ *D P* $\vee$ ($\exists$ *f*. (*f IsChainOver t*) $\wedge$
(*f Congruent t ModuloHide A*) $\wedge$ ($\forall$ *i*. *f i* $\in$ *T P*)))}

T-hiding : *T*(*P* \ *A*) = {*s*. $\exists$ *t*. *s* = *trace-hide t* (*ev*‘*A*) $\wedge$ *t* $\in$ *T P*}

hiding-cont [*simp*]: $[\![$*cont f*; *finite A*$]\!] \Longrightarrow$ *cont* (λx. *f x* \ *A*)

lemmas *tr-hide-set-def* = *trace-hide-def*
lemmas *Hide-set-def* = *hiding-def*
lemmas *F-hide-set* = *F-hiding*
lemmas *D-hide-set* = *D-hiding*
lemmas *T-hide-set* = *T-hiding*
lemmas *hide-set-cont* = *hiding-cont*

end

theory *Sync*
imports *Process*
begin

fun *setinterleaving*::*'a trace* × (*'a event*) *set* × *'a trace* ⇒ (*'a trace*)*set*
where

si-empty1: *setinterleaving*([], *X*, []) = {[]}
| *si-empty2*: *setinterleaving*([], *X*, (*y* # *t*)) =
(*if* (*y* ∈ *X*)
then {}
else {*z*.∃ *u*. *z* = (*y* # *u*) ∧ *u* ∈ *setinterleaving* ([], *X*, *t*)})
| *si-empty3*: *setinterleaving*((*x* # *s*), *X*, []) =
(*if* (*x* ∈ *X*)
then {}
else {*z*.∃ *u*. *z* = (*x* # *u*) ∧ *u* ∈ *setinterleaving* (*s*, *X*, [])})
| *si-neq* : *setinterleaving*((*x* # *s*), *X*, (*y* # *t*)) =
(*if* (*x* ∈ *X*)
then if (*y* ∈ *X*)
then if (*x* = *y*)
then {*z*.∃ *u*. *z* = (*x*#*u*) ∧ *u* ∈ *setinterleaving*(*s*, *X*, *t*)}
else {}
else {*z*.∃ *u*. *z* = (*y*#*u*) ∧ *u* ∈ *setinterleaving* ((*x*#*s*), *X*, *t*)}
else if (*y* ∉ *X*)
then {*z*.∃ *u*. *z* = (*x* # *u*) ∧ *u* ∈ *setinterleaving* (*s*, *X*, (*y* # *t*))}
∪ {*z*.∃ *u*. *z* = (*y* # *u*) ∧ *u* ∈ *setinterleaving*((*x* # *s*), *X*, *t*)}
else {*z*.∃ *u*. *z* = (*x* # *u*) ∧ *u* ∈ *setinterleaving* (*s*, *X*, (*y* # *t*))})

lemma *sym1* [*simp*]: *setinterleaving*([], *X*, *t*) = *setinterleaving*(*t*, *X* ,[])
⟨*proof*⟩

lemma *sym2* [*simp*]:
∀ *s*. *setinterleaving* (*s*, *X*, *t*) = *setinterleaving* (*t*, *X*, *s*)
⟶ *setinterleaving* (*a* # *s*, *X*, *t*) = *setinterleaving* (*t*, *X*, *a* # *s*)

⟨*proof*⟩

lemma *sym* [*simp*] : *setinterleaving*(*s*, *X*, *t*)= *setinterleaving*(*t*, *X*, *s*)
⟨*proof*⟩

abbreviation *setinterleaves-syntax*
(- *setinterleaves* ′(()′(-, -′)(), -′) [60,0,0,0]70)
where
u setinterleaves ((*s*, *t*), *X*) == (*u* ∈ *setinterleaving*(*s*, *X*, *t*))

definition *sync* :: [′*a process*,′*a set*,′*a process*] => ′*a process*
((3- ⟦-⟧/ -) [14,0,15] 14)
where
P ⟦ *A* ⟧ *Q* ==
Abs-Process({(*s*,*R*).∃ *t u X Y*. (*t*,*X*) ∈ *F P* ∧ (*u*,*Y*) ∈ *F Q* ∧
(*s setinterleaves* ((*t*,*u*),(*ev*‘*A*) ∪ {*tick*})) ∧
R = (*X* ∪ *Y*) ∩ ((*ev*‘*A*) ∪ {*tick*}) ∪ *X* ∩ *Y*} ∪
{(*s*,*R*).∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧
(*r setinterleaves* ((*t*,*u*),(*ev*‘*A*) ∪ {*tick*})) ∧
(*t* ∈ *D P* ∧ *u* ∈ *T Q* ∨ *t* ∈ *D Q* ∧ *u* ∈ *T P*)},
{*s*. ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧
(*r setinterleaves* ((*t*,*u*),(*ev*‘*A*) ∪ {*tick*})) ∧
(*t* ∈ *D P* ∧ *u* ∈ *T Q* ∨ *t* ∈ *D Q* ∧ *u* ∈ *T P*)})

axioms
F-sync : *F*(*P* ⟦ *A* ⟧ *Q*) =
{(*s*,*R*).∃ *t u X Y*. (*t*,*X*) ∈ *F P* ∧
(*u*,*Y*) ∈ *F Q* ∧
s setinterleaves ((*t*,*u*),(*ev*‘*A*) ∪ {*tick*}) ∧
R = (*X* ∪ *Y*) ∩ ((*ev*‘*A*) ∪ {*tick*}) ∪ *X* ∩ *Y*} ∪
{(*s*,*R*).∃ *t u r v*. *front-tickFree v* ∧
(*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧
r setinterleaves ((*t*,*u*),(*ev*‘*A*)∪{*tick*}) ∧
(*t* ∈ *D P* ∧ *u* ∈ *T Q* ∨ *t* ∈ *D Q* ∧ *u* ∈ *T P*)}

D-sync : *D*(*P* ⟦ *A* ⟧ *Q*) =
{*s*. ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v*=[]) ∧
s = *r*@*v* ∧ *r setinterleaves* ((*t*,*u*),(*ev*‘*A*) ∪ {*tick*}) ∧
(*t* ∈ *D P* ∧ *u* ∈ *T Q* ∨ *t* ∈ *D Q* ∧ *u* ∈ *T P*)}

T-sync : *T*(*P* ⟦ *A* ⟧ *Q*) =
{*s*. ∀ *t u*. *t* ∈ *T P* ∧ *u* ∈ *T Q* ∧

s *setinterleaves* $((t,u),(ev`A) \cup \{tick\})\}$

abbreviation *Inter-syntax* $((\text{-}|||\text{-})\ [14,15]\ 14)$
where $P\ |||\ Q == (P\ [\![\ \{\}\ ]\!]\ Q)$

abbreviation *Par-syntax* $((\text{-}||\text{-})\ [14,15]\ 14)$
where $P\ ||\ Q == (P\ [\![\ UNIV\ ]\!]\ Q)$

lemma *sync-cont*[*simp*]:
$[\![\ cont\ f;\ cont\ g\ ]\!] \Longrightarrow cont\ (\%x.\ (f\ x)\ [\![A]\!]\ (g\ x))$
⟨*proof*⟩

end

10 Toplevel Theory

theory *CSP*
imports *Bot Skip Stop Mprefix Det Ndet Seq Hide Sync Legacy*
begin

10.1 Refinement Proof Rules

10.2 The "Laws" of CSP

axioms

hide-mprefix-distr : $[\![\ (B \cap A) = \{\}\ ]\!] \Longrightarrow$
$((Mprefix\ A\ P) \setminus B) = (Mprefix\ A\ (\%\ x.\ ((P\ x) \setminus B)))$
hide-prefix-distr1 : $a : B \Longrightarrow ((a \rightarrow P) \setminus B) = (P \setminus B)$
hide-prefix-distr2 : $a\ ^{\sim}: B \Longrightarrow ((a \rightarrow P) \setminus B) = (a \rightarrow (P \setminus B))$
hide-det : $((a \rightarrow P) \Box (b \rightarrow Q)) \setminus \{a\} =$
$((P \setminus \{a\}) \sqcap ((P \setminus \{a\}) \Box (b \rightarrow (Q \setminus \{a\}))))$

lemma *mprefix-singl*: $(Mprefix\ \{a\}\ P) = (a \rightarrow (P\ a))$
⟨*proof*⟩

lemma *mono-mprefix-ref*: $\forall x.\ P\ x \sqsubseteq Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq Mprefix\ A\ Q$
⟨*proof*⟩

lemma *mono-prefix-ref*: $P \sqsubseteq Q \Longrightarrow (a \rightarrow P) \sqsubseteq (a \rightarrow Q)$
⟨*proof*⟩

lemma *mono-ndet-ref*: $[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq (P' \sqcap S')$
⟨*proof*⟩

lemma *mono-det-ref*:
$[\![P \sqsubseteq P';\ S \sqsubseteq S']\!] \Longrightarrow (P \,\Box\, S) \sqsubseteq (P' \,\Box\, S')$
⟨*proof*⟩

lemma *mono-hide-set-refD*:
$P \sqsubseteq Q \Longrightarrow D\ (Q \setminus A) \subseteq D\ (P \setminus A)$
⟨*proof*⟩

lemma *mono-hide-set-refF*: $P \sqsubseteq Q \Longrightarrow F\ (Q \setminus A) \subseteq F\ (P \setminus A)$
⟨*proof*⟩

lemma *mono-hide-set-ref*: $P \sqsubseteq Q \Longrightarrow P \setminus A \sqsubseteq Q \setminus A$
⟨*proof*⟩

lemma *mono-HSI2a*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (P \setminus A)\ s \subseteq Ra\ (Q \setminus A)\ s$
⟨*proof*⟩

lemma *mono-HSI2b*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (Q \setminus A)\ s \subseteq Ra\ (P \setminus A)\ s$
⟨*proof*⟩

lemma *mono-HSI2*: $[\![P \leq Q;\ s \notin D\ (P \setminus A)]\!] \Longrightarrow Ra\ (P \setminus A)\ s = Ra\ (Q \setminus A)\ s$
⟨*proof*⟩

lemma *mono-HSI31*:
$[\![tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A);\ s \in T\ Q;$
$\forall s.\ tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A) \longrightarrow$
$(s,\ ev\ `\ A) \notin F\ Q]\!]$
$\Longrightarrow \exists a.\ a \in ev\ `\ A \wedge s\ @\ [a] \in T\ Q$
⟨*proof*⟩

lemma *help1*: [a,b]=[a]@[b]
⟨*proof*⟩

lemma *help2*: [a]<[a,b]
⟨*proof*⟩

lemma *mono-HSI32*:
$[\![tickFree\ t;\ t \in T\ Q;$
$\forall s.\ tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ s\ (ev\ `\ A) \longrightarrow$
$(s,\ ev\ `\ A) \notin F\ Q]\!]$
$\Longrightarrow \exists f.\ f\ IsChainOver\ t \wedge f\ Congruent\ t\ ModuloHide\ A \wedge (\forall i.\ f\ i \in T\ Q)$
⟨*proof*⟩

lemma *mono-HSI33*:
$\exists\, n\ s.\ length\ s = n \wedge s \leq t \wedge tr\text{-}hide\text{-}set\ s\ A = tr\text{-}hide\text{-}set\ t\ A$
⟨*proof*⟩

lemma *mono-HSI34*:
$\exists\, s.\ tr\text{-}hide\text{-}set\ s\ A = tr\text{-}hide\text{-}set\ t\ A \wedge$
$s \leq t \wedge (\forall\, s1 < s.\ tr\text{-}hide\text{-}set\ s1\ A \neq tr\text{-}hide\text{-}set\ t\ A)$
⟨*proof*⟩

lemma *mono-HSI35*:
$[\![s < t;\ tr\text{-}hide\text{-}set\ s\ A \neq tr\text{-}hide\text{-}set\ t\ A]\!]$
$\Longrightarrow tr\text{-}hide\text{-}set\ s\ A < tr\text{-}hide\text{-}set\ t\ A$
⟨*proof*⟩

lemma *mono-HSI36*:
$\forall\, ta.\ (\exists\, t\ u.\ front\text{-}tickFree\ u \wedge$
$tickFree\ t \wedge$
$ta = tr\text{-}hide\text{-}set\ t\ (ev\ `\ A)\ @\ u \wedge$
$(t \in D\ P \vee$
$(\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge (\forall\, i.\ f\ i \in T\ P)))) \longrightarrow$
$\neg\ ta < tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) \Longrightarrow$
$\exists\, t1.\ tr\text{-}hide\text{-}set\ t1\ (ev\ `\ A) = tr\text{-}hide\text{-}set\ t\ (ev\ `\ A) \wedge$
$t1 \leq t \wedge (t1 \notin D\ P \vee t1 \in min\text{-}elems\ (D\ P))$
⟨*proof*⟩

lemma *mono-HSI3*:
$P \leq Q \Longrightarrow min\text{-}elems\ (D\ (P \setminus A)) \subseteq T\ (Q \setminus A)$
⟨*proof*⟩

lemma *mono-HSI*: $P \leq Q \Longrightarrow P \setminus A \leq Q \setminus A$
⟨*proof*⟩

lemma *mono-HS-rec*: $mono\ (\lambda\ P.\ (P \setminus A))$
⟨*proof*⟩

lemma *mono-PaI-refD*:
$P \sqsubseteq Q \Longrightarrow D\ (Q\ [\![A]\!]\ S) \subseteq D\ (P\ [\![A]\!]\ S)$
⟨*proof*⟩

lemma *mono-PaI-refF*: $P \sqsubseteq Q \Longrightarrow F\ (Q\ [\![A]\!]\ S) \subseteq F\ (P\ [\![A]\!]\ S)$
⟨*proof*⟩

lemma *mono-PaI-ref-L*: $P \sqsubseteq Q \Longrightarrow (P\ [\![A]\!]\ S) \sqsubseteq (Q\ [\![A]\!]\ S)$
⟨*proof*⟩

lemma *mono-PaI-ref-R*: $P \sqsubseteq Q \Longrightarrow (S \ [\![A]\!] \ P) \sqsubseteq (S \ [\![A]\!] \ Q)$
⟨*proof*⟩

lemma *mono-PaI-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P \ [\![A]\!] \ Q) \sqsubseteq (P' \ [\![A]\!] \ Q')$
⟨*proof*⟩

lemma *mono-Inter-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P|||Q) \sqsubseteq (P'|||Q')$
⟨*proof*⟩

lemma *mono-Par-ref*: $[\![P \sqsubseteq P'; \ Q \sqsubseteq Q']\!] \Longrightarrow (P||Q) \sqsubseteq (P'||Qk')$
⟨*proof*⟩

lemma *least-process*: $\bot \leq (P::'a \ process)$
⟨*proof*⟩

lemma *subset-F-Bot*: $F \ Q \leq F \ \bot$
⟨*proof*⟩

lemma *subset-D-Bot*: $D \ Q \leq D \ \bot$
⟨*proof*⟩

lemma *eq-F-Bot*: $F \ P = F \ \bot = (F \ \bot \subseteq F \ P)$
⟨*proof*⟩

lemma *eq-D-Bot*: $D \ P = D \ \bot = (D \ \bot \subseteq D \ P)$
⟨*proof*⟩

lemma *ftF-D-Bot*: *front-tickFree* $t = (t \in D \ \bot)$
⟨*proof*⟩

lemma *ftF-T-Bot*: *front-tickFree* $t = (t \in T \ \bot)$
⟨*proof*⟩

lemma *D-Bot-eq-T-Bot*: $D \ \bot = T \ \bot$
⟨*proof*⟩

lemma *is-processT6-S3*: $[\![fst \ x = []; \ tick \notin snd \ x; \ [tick] \in T \ P]\!] \Longrightarrow x \in F \ P$
⟨*proof*⟩

lemma *div-lemma*: $([] \in D\ P) = (P = \bot)$
⟨*proof*⟩

lemma *det-commute* :$(P \ \Box\ Q) =(Q \ \Box\ P)$
⟨*proof*⟩

lemma *det-bot* [*simp*]: $(P \ \Box\ \bot) = \bot$
⟨*proof*⟩

lemma *det-bot'*[*simp*]: $(\bot \ \Box\ P) = \bot$
⟨*proof*⟩

lemma *det-id*[*simp*]: $(P \ \Box\ P) = P$
⟨*proof*⟩

lemma *det-assoc*: $((M \ \Box\ P) \ \Box\ Q) = (M \ \Box\ (P \ \Box\ Q))$
⟨*proof*⟩

lemma *det-STOP*[*simp*]: $(P \ \Box\ STOP) = P$
⟨*proof*⟩

lemma *det-STOP'*[*simp*]: $(STOP \ \Box\ P) = P$
⟨*proof*⟩

lemma *ndet-id*[*simp*]: $(P \sqcap P) = P$
⟨*proof*⟩

lemma *ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
⟨*proof*⟩

lemma *ndet-bot*[*simp*]: $(P \sqcap \bot) = \bot$
⟨*proof*⟩

lemma *ndet-bot'*[*simp*]: $(\bot \sqcap P) = \bot$
⟨*proof*⟩

lemma *non-det-assoc*: $((M \sqcap P) \sqcap Q) = (M \sqcap (P \sqcap Q))$
⟨*proof*⟩

lemma *det-distrib*: $(M \ \Box\ (P \sqcap Q)) = ((M \ \Box\ P) \sqcap (M \ \Box\ Q))$
⟨*proof*⟩

lemma *non-det-distrib*: $(M \sqcap (P \ \Box\ Q)) = ((M \sqcap P) \ \Box\ (M \sqcap Q))$

⟨*proof*⟩

lemma *pref-non-det*: $(a \rightarrow (P \sqcap Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
⟨*proof*⟩

lemma *Mprefix-STOP*: $(Mprefix\ \{\}\ P) = STOP$
⟨*proof*⟩

lemma *mprefix-Un-distrD*:
$D(Mprefix\ (A \cup B)\ P) = D\ (Mprefix\ A\ P \Box Mprefix\ B\ P)$
⟨*proof*⟩

lemma *mprefix-Un-distrF*:
$F(Mprefix\ (A \cup B)\ P) = F((Mprefix\ A\ P) \Box (Mprefix\ B\ P))$
⟨*proof*⟩

lemma *mprefix-Un-distr*: $(Mprefix\ (A \cup B)\ P) = ((Mprefix\ A\ P) \Box (Mprefix\ B\ P))$
⟨*proof*⟩

lemma *mnon-det-non-det*: $(Mprefix\ (A \cup \{a\})\ P) = ((Mprefix\ A\ P) \Box (a \rightarrow (P\ a)))$
⟨*proof*⟩

lemma *pref-det-non-det*: $((a \rightarrow P) \Box (a \rightarrow Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
⟨*proof*⟩

lemma *SEQ-SKIPD*: $D(P\ ;\ SKIP) = D\ P$
⟨*proof*⟩

lemma *SEQ-SKIPF*: $F(P\ ;\ SKIP) = F\ P$
⟨*proof*⟩

lemma *SEQ-SKIP*: $(P\ ;\ SKIP) = P$
⟨*proof*⟩

lemma *SKIP-SEQD*: $D(SKIP\ ;\ P) = D\ P$
⟨*proof*⟩

lemma *SKIP-SEQF*: $F(SKIP\ ;\ P) = F\ P$

⟨*proof*⟩

lemma *SKIP-SEQ*: $(SKIP\ ;'\ P) = P$
⟨*proof*⟩

lemma *ev-Neq-tick1*:$ev\ a = b \Longrightarrow b \tilde{=} tick$
⟨*proof*⟩

lemma *Nelem-image-ev*:
$[\![\forall\ c.\ c \in B \longrightarrow c \notin C;\ hd\ x \in ev\ `\ B]\!] \Longrightarrow hd\ x \notin ev\ `\ C$
⟨*proof*⟩

lemma *mprefix-seqD*:
$D((Mprefix\ A\ P)\ ;'\ Q) = D(Mprefix\ A\ (\lambda\ x.\ (P\ x)\ ;'\ Q))$
⟨*proof*⟩

lemma *mprefix-seqF1*:
$a \notin B \Longrightarrow (A \cup \{a\}) \cap B = A \cap B$
⟨*proof*⟩

lemma *mprefix-seqF*: $F((Mprefix\ A\ P)\ ;'\ Q) = F(Mprefix\ A\ (\lambda x.\ (P\ x)\ ;'\ Q))$
⟨*proof*⟩

lemma *mprefix-seq*:
$((Mprefix\ A\ P)\ ;'\ Q) = (Mprefix\ A\ (\lambda x.\ (P\ x)\ ;'\ Q))$
⟨*proof*⟩

lemma *pref-seq*: $((a \rightarrow P)\ ;'\ Q) = (a \rightarrow (P\ ;'\ Q))$
⟨*proof*⟩

lemma *STOP-SEQ*: $(STOP\ ;'\ P) = STOP$
⟨*proof*⟩

lemma *prefix-stop-seq*: $((a \rightarrow STOP)\ ;'\ P) = (a \rightarrow STOP)$
⟨*proof*⟩

lemma *prefix-skip-seq*: $((a \rightarrow SKIP)\ ;'\ P) = (a \rightarrow P)$
⟨*proof*⟩

lemma *Bot-SEQ*: $(\bot \; `;` \; P) = \bot$
⟨*proof*⟩

lemma *SEQ-Ndet-distrRD*: $D((P \sqcap Q) \; `;` \; S) = D((P \; `;` \; S) \sqcap (Q \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Ndet-distrRF*: $F((P \sqcap Q) \; `;` \; S) = F((P \; `;` \; S) \sqcap (Q \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Ndet-distrR*: $((P \sqcap Q) \; `;` \; S) = ((P \; `;` \; S) \sqcap (Q \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Ndet-distrLD*: $D(P \; `;` \; (Q \sqcap S)) = D((P \; `;` \; Q) \sqcap (P \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Ndet-distrLF*: $F(P \; `;` \; (Q \sqcap S)) = F((P \; `;` \; Q) \sqcap (P \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Ndet-distrL*: $(P \; `;` \; (Q \sqcap S)) = ((P \; `;` \; Q) \sqcap (P \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Det-distrRD*: $D((P \mathbin{\square} Q) \; `;` \; S) = D((P \; `;` \; S) \mathbin{\square} (Q \; `;` \; S))$
⟨*proof*⟩

lemma *SEQ-Det-distrRF*: $F((P \mathbin{\square} Q) \; `;` \; S) \subseteq F((P \; `;` \; S) \mathbin{\square} (Q \; `;` \; S))$
⟨*proof*⟩

find-theorems *front-tickFree*
lemma *par-Int-botD*: $D(P \; [\![\; A \;]\!] \; \bot) = D \; \bot$
⟨*proof*⟩

lemma *par-Int-botF*: $F(P \; [\![\; A \;]\!] \; \bot) = F \; \bot$
⟨*proof*⟩

lemma *par-Int-bot*[*simp*]: $(P \; [\![\; A \;]\!] \; \bot) = \bot$
⟨*proof*⟩

lemma *par-Int-bot1*[*simp*]: $(\bot \; [\![\; A \;]\!] \; P) = \bot$
⟨*proof*⟩

lemma *par-Int-skip-D*: $D(SKIP \llbracket A \rrbracket SKIP) = D\ SKIP$
⟨*proof*⟩

lemma *par-Int-skip-F*: $F(SKIP \llbracket A \rrbracket SKIP) = F\ SKIP$
⟨*proof*⟩

lemma *par-Int-skip*: $(SKIP \llbracket A \rrbracket SKIP) = SKIP$
⟨*proof*⟩

lemma *sync-commute*: $(P \llbracket A \rrbracket Q) = (Q \llbracket A \rrbracket P)$
⟨*proof*⟩

lemmas *Par-Int-commute* = *sync-commute*

lemma *Inter-commute*: $(P \mid\mid\mid Q) = (Q \mid\mid\mid P)$
⟨*proof*⟩

lemma *Inter-skip-D*: $D(P \mid\mid\mid SKIP) = D\ P$
⟨*proof*⟩

lemma *Inter-skip-F1*:
$[\![(t, X) \in F\ P; fst\ x\ setinterleaves\ ((t, [tick]), \{tick\})]\!]$
$\Longrightarrow x \in F\ P$
⟨*proof*⟩

lemma *Inter-skip-F*: $F(P \mid\mid\mid SKIP) = F\ P$
⟨*proof*⟩

lemma *Inter-skip1*: $(P \mid\mid\mid SKIP) = P$
⟨*proof*⟩

lemma *Inter-skip2*: $(SKIP \mid\mid\mid P) = P$
⟨*proof*⟩

lemma *skip-Neq-stop*: $SKIP \neq STOP$
⟨*proof*⟩

lemma *stop-Neq-skip*: $STOP \neq SKIP$
⟨*proof*⟩

lemma *Inter-stop-seq-stop-D*: $D(P \mid\mid\mid STOP) = D(P\ ;\ STOP)$
⟨*proof*⟩

lemma *setH1*:
$(X \cup Y) \cap A \cup X \cap Y \cup A = X \cap Y \cup A$
⟨*proof*⟩

lemma *Inter-stop-seq-stop-F*: $F(P \mid\mid\mid STOP) = F(P \text{ `;` } STOP)$
⟨*proof*⟩

lemma *Inter-stop-seq-stop*:
$(P \mid\mid\mid STOP) = (P \text{ `;` } STOP)$
⟨*proof*⟩

lemma *Inter-stop-seq-stop1*:
$(STOP \mid\mid\mid P) = (P \text{ `;` } STOP)$
⟨*proof*⟩

lemma *par-int-ndet-distribD*:
$D(M \llbracket A \rrbracket (P \sqcap Q)) = D((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
⟨*proof*⟩

lemma *par-int-ndet-distribF*:
$F(M \llbracket A \rrbracket (P \sqcap Q)) = F((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
⟨*proof*⟩

lemma *par-int-ndet-distrib*:
$(M \llbracket A \rrbracket (P \sqcap Q)) = ((M \llbracket A \rrbracket P) \sqcap (M \llbracket A \rrbracket Q))$
⟨*proof*⟩

lemma *par-int-ndet-distrib1*:
$((P \sqcap Q) \llbracket A \rrbracket M) = ((P \llbracket A \rrbracket M) \sqcap (Q \llbracket A \rrbracket M))$
⟨*proof*⟩

lemma *par-comm*: $(P \mid\mid Q) = (Q \mid\mid P)$
⟨*proof*⟩

lemma *par-ndet-distrib1*:
$(M \mid\mid (P \sqcap Q)) = ((M \mid\mid P) \sqcap (M \mid\mid Q))$

⟨*proof*⟩

lemma *par-ndet-distrib2*:
$((P \sqcap Q) \mid\mid M) = ((P \mid\mid M) \sqcap (Q \mid\mid M))$
⟨*proof*⟩

lemma *par-stopD*: $P \neq \bot \Longrightarrow D(P \mid\mid STOP) = D\ STOP$
⟨*proof*⟩

lemma *par-stopF*: $P \neq \bot \Longrightarrow F(P \mid\mid STOP) = F\ STOP$
⟨*proof*⟩

lemma *par-stop*: $P \neq \bot \Longrightarrow (P \mid\mid STOP) = STOP$
⟨*proof*⟩

lemma *par-assocD1*: $D((P \mid\mid Q) \mid\mid S) \subseteq D(P \mid\mid (Q \mid\mid S))$
⟨*proof*⟩

lemma *par-assocD2*: $D(P \mid\mid (Q \mid\mid S)) \subseteq D((P \mid\mid Q) \mid\mid S)$
⟨*proof*⟩

lemma *par-assocD*: $D((P \mid\mid Q) \mid\mid S) = D(P \mid\mid (Q \mid\mid S))$
⟨*proof*⟩

lemma *par-assocF1*: $F((P \mid\mid Q) \mid\mid S) \subseteq F(P \mid\mid (Q \mid\mid S))$
⟨*proof*⟩

lemma *par-assocF2*: $F(P \mid\mid (Q \mid\mid S)) \subseteq F((P \mid\mid Q) \mid\mid S)$
⟨*proof*⟩

lemma *par-assocF*: $F((P \mid\mid Q) \mid\mid S) = F(P \mid\mid (Q \mid\mid S))$
⟨*proof*⟩

lemma *par-assoc*: $((P \mid\mid Q) \mid\mid S) = (P \mid\mid (Q \mid\mid S))$
⟨*proof*⟩

lemma *hide-set-botD*: $D(\bot \setminus A) = D\ \bot$
⟨*proof*⟩

lemma *hide-set-botF*: $F(\bot \setminus A) = F\ \bot$
⟨*proof*⟩

lemma *hide-set-bot*[*simp*]: $(\bot \setminus A) = \bot$
⟨*proof*⟩

lemma *hide-set-STOPD*: $D(STOP \setminus A) = D\ STOP$
⟨*proof*⟩

lemma *hide-set-STOPF*: $F(STOP \setminus A) = F\ STOP$
⟨*proof*⟩

lemma *hide-set-STOP*: $(STOP \setminus A) = STOP$
⟨*proof*⟩

lemma *hide-set-SKIPD*: $D(SKIP \setminus A) = D\ SKIP$
⟨*proof*⟩

lemma *hide-set-SKIPF*: $F(SKIP \setminus A) = F\ SKIP$
⟨*proof*⟩

lemma *hide-set-SKIP*: $(SKIP \setminus A) = SKIP$
⟨*proof*⟩

lemma *hide-set-emptyD*: $D(P \setminus \{\}) = D\ P$
⟨*proof*⟩

lemma *hide-set-emptyF*: $F(P \setminus \{\}) = F\ P$
⟨*proof*⟩

lemma *hide-set-empty*: $(P \setminus \{\}) = P$
⟨*proof*⟩

lemma $D(P \setminus (A\ Un\ B)) <= D((P \setminus A) \setminus B)$
⟨*proof*⟩

lemma $D((P \setminus A) \setminus B) <= D(P \setminus (A\ Un\ B))$
⟨*proof*⟩

lemma *f-mono*:
$\forall i.\ f\ i < f\ (Suc\ i) \Longrightarrow i < j \longrightarrow f\ i < f\ j$
⟨*proof*⟩

lemma *f-0-less-f-i*:
$[\![f\ 0 = t; \forall i.\ f\ i < f\ (Suc\ i);\ 1 \leq i]\!] \Longrightarrow t < f\ i$
⟨*proof*⟩

lemma
$\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge (\forall i.\ f\ i \in T\ P \vee f\ i \in T\ Q) \Longrightarrow$

$\exists f.\ f\ IsChainOver\ t \wedge$
$f\ Congruent\ t\ ModuloHide\ A \wedge ((\forall i.\ f\ i \in T\ P) \vee (\forall i.\ f\ i \in T\ Q))$
⟨*proof*⟩

lemma $D((P \sqcap Q) \setminus A) = D((P \setminus A) \sqcap (Q \setminus A))$
⟨*proof*⟩

lemma *le-trans*:
$\bigwedge i.\ [\![i \leq j; j \leq k]\!] \Longrightarrow i \leq (k::nat)$
⟨*proof*⟩

lemma *length-f-i1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ length\ t + i \leq length\ (f\ i)$
⟨*proof*⟩

lemma *f-i-Neq-Nil1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ i \neq 0 \longrightarrow f\ i \neq []$
⟨*proof*⟩

lemma *tl-f-i-less-tl-f-Suc1*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow$
$\forall i.\ i \neq 0 \longrightarrow tl\ (f\ i) < tl\ (f\ (Suc\ i))$
⟨*proof*⟩

lemma *length-f-i2*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ length\ t + i < length\ (f\ (Suc\ i))$
⟨*proof*⟩

lemma *tl-f-Suc-i-Neq-Nil*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ i \neq 0 \longrightarrow tl\ (f\ (Suc\ i)) \neq []$
⟨*proof*⟩

lemma *tl-f-i-less-tl-f-Suc2*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow \forall i.\ f\ (Suc\ i) \neq []$
⟨*proof*⟩

lemma *tl-f-Suc-i-less-tl-f-Suc2*:
$f\ 0 = t \wedge (\forall i.\ f\ i < f\ (Suc\ i)) \Longrightarrow$
$(if\ i = 0\ then\ t\ else\ tl\ (f\ (Suc\ i))) < tl\ (f\ (Suc\ (Suc\ i)))$
⟨*proof*⟩

lemma *det-left-commute*: $(M \ \square\ P \ \square\ Q) = (P \ \square\ M \ \square\ Q)$

⟨*proof*⟩

lemma *det-left-id*: $(M \ \Box \ M \ \Box \ Q) = (M \ \Box \ Q)$
⟨*proof*⟩

lemma *non-det-left-commute*: $(M \sqcap P \sqcap Q) = (P \sqcap M \sqcap Q)$
⟨*proof*⟩

lemma *non-det-left-id*: $(M \sqcap M \sqcap Q) = (M \sqcap Q)$
⟨*proof*⟩

lemma *par-left-commute*: $(M \ || \ P \ || \ Q) = (P \ || \ M \ || \ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3D1*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow D \ (Mprefix \ A \ P \ [\![C \ ]\!] \ Mprefix \ B \ Q)$
$\subseteq D \ (\Box \ x \in \ B \rightarrow (Mprefix \ A \ P \ [\![C \ ]\!] \ Q \ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3D2*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow D \ (\Box \ x \in \ B \rightarrow (Mprefix \ A \ P \ [\![C \ ]\!] \ Q \ x))$
$\subseteq D \ (Mprefix \ A \ P \ [\![C \ ]\!] \ Mprefix \ B \ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3D*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow D \ (\Box \ x \in \ B \rightarrow (Mprefix \ A \ P \ [\![C \ ]\!] \ Q \ x))$
$\subseteq D \ (Mprefix \ A \ P \ [\![C]\!] \ Mprefix \ B \ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3F1*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F \ (Mprefix \ A \ P \ [\![C \ ]\!] \ Mprefix \ B \ Q)$
$\subseteq F \ (\Box \ x \in \ B \rightarrow (Mprefix \ A \ P \ [\![C \ ]\!] \ Q \ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3F2*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F \ (\Box \ x \in \ B \rightarrow (Mprefix \ A \ P \ [\![C \ ]\!] \ Q \ x))$
$\subseteq F \ (Mprefix \ A \ P \ [\![C \ ]\!] \ Mprefix \ B \ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3F*:

$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) =$
$F\ (\Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr3*:
$[\![B \cap C = \{\};\ A \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ x)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr2*:
$[\![A \cap C = \{\};\ B \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par2D1a*:
$[\![B \cap C = \{\};\ A \cap C = \{\};$
$\exists\, t\ u\ r\ v.$
$front\text{-}tickFree\ v \wedge$
$tickFree\ r \wedge$
$x = r\ @\ v \wedge$
$r\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ A\ P) \wedge u \in T\ (Mprefix\ B\ Q)]\!]$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par2D1b*:
$[\![B \cap C = \{\};\ A \cap C = \{\};$
$\exists\, t\ u\ r\ v.$
$front\text{-}tickFree\ v \wedge$
$tickFree\ r \wedge$
$x = r\ @\ v \wedge$
$r\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ B\ Q) \wedge u \in T\ (Mprefix\ A\ P)]\!]$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par2D1*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq D\ (\Box\ x \in\ A \rightarrow (P\ x\ [\![C\ ]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \rightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par2D2a*: [| B *Int* $C = \{\}$; A *Int* $C = \{\}$; x:$D(\ \Box\ x \in A \to ((P\ x) \llbracket\ C\ \rrbracket\ (Mprefix\ B\ Q)))$ |] ==> $x : D\ ((Mprefix\ A\ P) \llbracket\ C\ \rrbracket\ (Mprefix\ B\ Q))$
⟨*proof*⟩

lemma *mprefix-Par2D2b*:
$\llbracket B \cap C = \{\};\ A \cap C = \{\};\ x \in D\ (\Box\ y \in\ B \to (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Q\ y))\rrbracket$
$\Longrightarrow x \in D\ (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par2D2*:
$\llbracket B \cap C = \{\};\ A \cap C = \{\}\rrbracket$
$\Longrightarrow D\ (\Box\ x \in\ A \to (P\ x\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \to (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Q\ y))$
$\subseteq D\ (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par2D*:
$\llbracket B \cap C = \{\};\ A \cap C = \{\}\rrbracket$
$\Longrightarrow D\ (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q) =$
$D\ (\Box\ x \in\ A \to (P\ x\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \to (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Q\ y))$
⟨*proof*⟩

lemma *AuxEv1*:
$\forall\, c.\ c \in A \longrightarrow c \notin B \Longrightarrow \forall\, c.\ c \in ev\ `\ B \longrightarrow c \notin ev\ `\ A$
⟨*proof*⟩

lemma *mprefix-Par2F1*:
$\llbracket B \cap C = \{\};\ A \cap C = \{\}\rrbracket$
$\Longrightarrow F\ (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)$
$\subseteq F\ (\Box\ x \in\ A \to (P\ x\ \llbracket C\ \rrbracket\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in\ B \to (Mprefix\ A\ P\ \llbracket C\ \rrbracket\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par2F2a*:
$\llbracket B \cap C = \{\};\ A \cap C = \{\};\ fst\ x \neq [];\ hd\ (fst\ x) \in ev\ `\ A;$
$ev\ a = hd\ (fst\ x);\ (t,\ X) \in F\ (P\ a);\ (u,\ Y) \in F\ (Mprefix\ B\ Q);$
$tl\ (fst\ x)\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\});$
$snd\ x = (X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y\rrbracket$
$\Longrightarrow \exists\, t\ u\ Xa\ Ya.$
$(t = [] \wedge Xa \cap ev\ `\ A = \{\} \vee$
$t \neq [] \wedge$
$hd\ t \in ev\ `\ A \wedge (\exists\, a.\ ev\ a = hd\ t \wedge (tl\ t,\ Xa) \in F\ (P\ a))) \wedge$
$(u = [] \wedge Ya \cap ev\ `\ B = \{\} \vee$

$u \neq [] \wedge$
$hd\ u \in ev\ `\ B \wedge (\exists\, a.\ ev\ a = hd\ u \wedge (tl\ u,\ Ya) \in F\ (Q\ a))) \wedge$
$fst\ x\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$(X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y =$
$(Xa \cup Ya) \cap (ev\ `\ C \cup \{tick\}) \cup Xa \cap Ya$
⟨*proof*⟩

lemma *mprefix-Par2F2b*:
$(\exists\, t\ u\ Xa\ Ya.$
$P\ t\ Xa \wedge$
$Q\ u\ Ya \wedge$
$fst\ x\ setinterleaves\ ((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$(X \cup Y) \cap (ev\ `\ C \cup \{tick\}) \cup X \cap Y =$
$(Xa \cup Ya) \cap (ev\ `\ C \cup \{tick\}) \cup Xa \cap Ya) =$
$(\exists\, u\ t\ Ya\ Xa.$
$Q\ u\ Ya \wedge$
$P\ t\ Xa \wedge$
$fst\ x\ setinterleaves\ ((u,\ t),\ ev\ `\ C \cup \{tick\}) \wedge$
$(Ya \cup Xa) \cap (ev\ `\ C \cup \{tick\}) \cup Ya \cap Xa)$
⟨*proof*⟩

lemma *mprefix-Par2F2*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow F\ (\Box\ x \in A \rightarrow (P\ x\ [\![C]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in B \rightarrow (Mprefix\ A\ P\ [\![C]\!]\ Q\ y))$
$\subseteq F\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par2F*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q) =$
$F\ (\Box\ x \in A \rightarrow (P\ x\ [\![C]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in B \rightarrow (Mprefix\ A\ P\ [\![C]\!]\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par-Int2*:
$[\![B \cap C = \{\};\ A \cap C = \{\}]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q) =$
$(\Box\ x \in A \rightarrow (P\ x\ [\![C]\!]\ Mprefix\ B\ Q)\ \Box$
$\Box\ y \in B \rightarrow (Mprefix\ A\ P\ [\![C]\!]\ Q\ y))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1D1a*:
$[\![A \subseteq C;\ B \subseteq C;$
$\exists\, t\ u\ r\ v.$
front-tickFree $v \wedge$
tickFree $r \wedge$
$x = r\ @\ v \wedge$
r *setinterleaves* $((t,\ u),\ ev\ `\ C \cup \{tick\}) \wedge$
$t \in D\ (Mprefix\ A\ P) \wedge u \in T\ (Mprefix\ B\ Q)]\!]$
$\Longrightarrow x \in D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1D1*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
$\subseteq D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1D2*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
$\subseteq D\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1D*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow D\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q) =$
$D\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1F1*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow F\ (Mprefix\ A\ P\ [\![C]\!]\ Mprefix\ B\ Q)$
$\subseteq F\ (\Box\ x \in A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1F2*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow F(\Box\ x \in A \cap B \cap C \rightarrow (P\ x)\ [\![C]\!]\ (Q\ x))$
$\subseteq F((\Box\ x \in A \rightarrow P\ x)[\![\ C\ ]\!]\ ((\Box\ x \in B \rightarrow Q\ x)))$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1F*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow F\ (\Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x))$
$\subseteq F\ (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q)$
⟨*proof*⟩

lemma *mprefix-Par-Int-distr1*:
$[\![A \subseteq C;\ B \subseteq C]\!]$
$\Longrightarrow (Mprefix\ A\ P\ [\![C\ ]\!]\ Mprefix\ B\ Q) = \Box\ x \in\ A \cap B \cap C \rightarrow (P\ x\ [\![C\ ]\!]\ Q\ x)$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipD*:
$D\ (Mprefix\ A\ P\ [\![B\ ]\!]\ SKIP) = D\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipF1*:
$[\![X \cap A = \{\};\ tick \notin Y]\!]$
$\Longrightarrow ((X \cup Y) \cap (B \cup \{tick\}) \cup X \cap Y) \cap (A - B) = \{\}$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipF2*:
$X \cap (ev\ `\ A - ev\ `\ B) = \{\} \Longrightarrow$
$X =$
$(X - ev\ `\ A \cup (X - \{tick\})) \cap (ev\ `\ B \cup \{tick\}) \cup$
$(X - ev\ `\ A) \cap (X - \{tick\})$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipF3*:
$[\![x\ setinterleaves\ ((t,\ [tick]),\ ev\ `\ B \cup \{tick\});\ t \neq []]\!]$
$\Longrightarrow hd\ t \notin ev\ `\ B$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipF4*:
$[\![x \neq [];\ ev\ a = hd\ x;$
$tl\ x\ setinterleaves\ ((t,\ [tick]),\ ev\ `\ B \cup \{tick\});\ hd\ x \notin ev\ `\ B]\!]$
$\Longrightarrow x\ setinterleaves\ ((x,\ [tick]),\ ev\ `\ B \cup \{tick\})$
⟨*proof*⟩

lemma *mprefix-Par-Int-skipF*:
$F((Mprefix\ A\ P)\ [\![B]\!]\ SKIP) = F(Mprefix\ (A - B)(\%x.(P\ x)\ [\![\ B\ ]\!]\ SKIP))$
⟨*proof*⟩

lemma *mprefix-Par-Int-skip*:
$F\ (Mprefix\ A\ P\ [\![B]\!]\ SKIP) = F\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
⟨*proof*⟩

lemma *mprefix-Par-Int-skip1*:
$F\ (Mprefix\ A\ P\ [\![B]\!]\ SKIP) = F\ (\Box\ x \in\ A - B \rightarrow (P\ x\ [\![B\ ]\!]\ SKIP))$
⟨*proof*⟩

lemma *par-Int-skip-stop*: $(SKIP\ [\![A]\!]\ STOP) = STOP$

⟨*proof*⟩

lemma *par-Int-skip-stop1*: $(STOP\ [\![A]\!]\ SKIP) = STOP$
⟨*proof*⟩

lemma *Inter-skip-stop*: $(SKIP\ |||\ STOP) = STOP$
⟨*proof*⟩

lemma *Inter-stop-skip*: $(STOP\ |||\ SKIP) = STOP$
⟨*proof*⟩

lemma *prefix-Par-Int-skip1*:
$a \notin A \Longrightarrow (a \rightarrow P\ [\![A]\!]\ SKIP) = (a \rightarrow (P\ [\![A]\!]\ SKIP))$
⟨*proof*⟩

lemma *prefix-Par-Int-skip1a*:
$a \notin A \Longrightarrow (SKIP\ [\![A]\!]\ a \rightarrow P) = (a \rightarrow (SKIP\ [\![A]\!]\ P))$
⟨*proof*⟩

lemma *prefix-Par-Int-skip2*:
$a \in A \Longrightarrow (a \rightarrow P\ [\![A]\!]\ SKIP) = STOP$
⟨*proof*⟩

lemma *prefix-Par-Int-skip*:
$(a \rightarrow P\ [\![A]\!]\ SKIP) = (if\ a \in A\ then\ STOP\ else\ (a \rightarrow (P\ [\![A\ ]\!]\ SKIP)))$
⟨*proof*⟩

lemma *prefix-Par-Int-skip2a*:
$a \in A \Longrightarrow (SKIP\ [\![A]\!]\ a \rightarrow P) = STOP$
⟨*proof*⟩

lemma *prefix-par-Int1*:
$[\![a \in A;\ b \in A;\ a \neq b]\!] \Longrightarrow (a \rightarrow P\ [\![A]\!]\ b \rightarrow Q) = STOP$
⟨*proof*⟩

lemma *prefix-par-Int2*: $a{:}A ==> ((a \rightarrow P)\ [\![A]\!]\ (a \rightarrow Q)) = (a \rightarrow (P\ [\![\ A\ ]\!]\ Q))$
⟨*proof*⟩

lemma *prefix-par*: $((a \rightarrow P)\ ||\ (a \rightarrow Q)) = (a \rightarrow (P\ ||\ Q))$
⟨*proof*⟩

lemma *prefix-Par-Int3*:
$[\![a \in C;\ b \notin C]\!] \Longrightarrow (a \rightarrow P\ [\![C]\!]\ b \rightarrow Q) = (b \rightarrow (a \rightarrow P\ [\![C]\!]\ Q))$
⟨*proof*⟩

lemma *prefix-Par-Int4*:
$[\![a \notin C;\ b \in C]\!] \Longrightarrow (a \rightarrow P\ [\![C]\!]\ b \rightarrow Q) = (a \rightarrow (P\ [\![C]\!]\ b \rightarrow Q))$

⟨*proof*⟩

lemma *prefix-Inter*:
$((a \to P) \;|||\; (b \to Q)) = ((a \to (P \;|||\; (b \to Q))) \;\Box\; (b \to ((a \to P) \;|||\; Q)))$
⟨*proof*⟩

lemma *IF-SEQ*: $((\textit{if } b \textit{ then } P \textit{ else } Q)\ ;'\ S) = (\textit{if } b \textit{ then } P\ ;'\ S \textit{ else } Q\ ;'\ S)$
⟨*proof*⟩

lemma
$a \neq b \Longrightarrow$
$(((b \to SKIP)\ [\![\{b\}]\!]\ (a \to SKIP))|||b \to SKIP) \neq$
$((b \to SKIP)\ [\![\{b\}]\!]\ ((a \to SKIP)|||(b \to SKIP)))$
⟨*proof*⟩

lemma $[\![\textit{directed } X; \textit{finite } A]\!] \Longrightarrow (\bigsqcup P{\in}X.\ P \setminus A) = \textit{lub } X \setminus A$
⟨*proof*⟩

lemma *cont-imp-cont-lub*: $\textit{cont } f \Longrightarrow \textit{contlub } (f)$
⟨*proof*⟩

lemma *dir-image2*: $[\![\textit{directed } X; \textit{mono } f]\!] \Longrightarrow \textit{directed } (f\ `\ X)$
⟨*proof*⟩

lemma *mono-contlub-imp-cont*: $[\![\textit{mono } f; \textit{contlub } f]\!] \Longrightarrow \textit{cont } f$
⟨*proof*⟩

lemma *prefix-contlub*: $\textit{cont } f \Longrightarrow \textit{contlub } (\lambda x.\ a \to f\ x)$
⟨*proof*⟩

lemma *prefix-cont*: $\textit{cont } f \Longrightarrow \textit{cont } (\lambda x.\ a \to f\ x)$
⟨*proof*⟩

lemma *elemDIselemHD*: $t \in D\ P \Longrightarrow \textit{tr-hide-set } t\ (ev\ `\ A) \in D\ (P \setminus A)$
⟨*proof*⟩

lemma $D((P \setminus \{a\}) \setminus \{b\}) \subseteq D(P \setminus (\{a\} \cup \{b\}))$
⟨*proof*⟩

lemma $D(P \setminus (\{a\} \cup \{b\})) \subseteq D((P \setminus \{a\}) \setminus \{b\})$
⟨*proof*⟩

lemma *hide-set-UnF1*:
$D((P \setminus \{a\}) \setminus \{b\}) = D(P \setminus (\{a\} \cup \{b\})) \Longrightarrow$
$F((P \setminus \{a\}) \setminus \{b\}) \subseteq F(P \setminus (\{a\} \cup \{b\}))$
⟨*proof*⟩

lemma *hide-set-UnF*:
$D((P \setminus \{a\}) \setminus \{b\}) = D(P \setminus (\{a\} \cup \{b\})) \Longrightarrow$
$F(P \setminus (\{a\} \cup \{b\})) \subseteq F((P \setminus \{a\}) \setminus \{b\})$
⟨*proof*⟩

lemma *alphabet1*:
$P \setminus A = P \Longrightarrow \forall\, t.\ t \in T\ P - D\ P \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *alphabet2*:
$P \setminus A = P \Longrightarrow \forall\, t.\ t \in$ *min-elems* $(D\ P) \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *setinterl-Nil-tick*:
$[\![u = [] \vee u = [tick];\ r$ *setinterleaves* $((t,\ u),\ ev\ `\ A \cup \{tick\})]\!]$
$\Longrightarrow r = t \wedge (\forall\, x.\ x \in ev\ `\ A \longrightarrow \neg$ *member* $t\ x)$
⟨*proof*⟩

lemma *NelemAlphSpec*:
$P \setminus A = P \Longrightarrow$
$\forall\, t.\ t \in T\ P - D\ P \cup$ *min-elems* $(D\ P) \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *NelemAlphSpec1a*:
$(P\ [\![A]\!]\ SKIP) \sqsubseteq P \Longrightarrow \forall\, t.\ t \in T\ P - D\ P \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *NelemAlphSpec1b*:
$(P\ [\![A]\!]\ SKIP) \sqsubseteq P \Longrightarrow$
$\forall\, t.\ t \in$ *min-elems* $(D\ P) \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *NelemAlphSpec1*:
$(P\ [\![A]\!]\ SKIP) = P \Longrightarrow$
$\forall\, t.\ t \in T\ P - D\ P \cup$ *min-elems* $(D\ P) \longrightarrow$ *tr-hide-set* $t\ (ev\ `\ A) = t$
⟨*proof*⟩

lemma *adm-eq1*:

$[\![cont\ f;\ cont\ g]\!] \Longrightarrow adm\ (\lambda x.\ f\ x = g\ x)$
⟨*proof*⟩

lemma *NelemAlphRec*:
$[\![cont\ u;\ \bigwedge x.\ (x\ [\![A\ ]\!]\ SKIP) = x \Longrightarrow (u\ x\ [\![A\ ]\!]\ SKIP) = u\ x]\!]$
$\Longrightarrow (mu\ u\ [\![A\ ]\!]\ SKIP) = mu\ u$
⟨*proof*⟩

11 Infra-structure for Communication Primitives

lemma *read-read-sync*:
assumes *contained*: $(\bigwedge y.\ c\ y \in C)$
shows $((c\ `?`\ x \to P\ x)[\![C]\!](c\ `?`\ x \to Q\ x)) =$
$(c\ `?`\ x \to ((P\ x)[\![C]\!](Q\ x)))$
⟨*proof*⟩

lemmas *read-Par-Int-distr1 = read-read-sync*

lemma *read-read-nonsync-left*:
$[\![\bigwedge y.\ c\ y \notin C;\ \bigwedge y.\ d\ y \in C]\!] \Longrightarrow$
$((c\ `?`\ x \to (P\ x))\ [\![C]\!]\ (d\ `?`\ x \to (Q\ x))) =$
$(c\ `?`\ x \to ((P\ x)\ [\![C]\!]\ (d\ `?`\ x \to (Q\ x))))$
⟨*proof*⟩

lemmas *read-Par-Int-distr2 = read-read-nonsync-left*

lemma *read-read-nonsync-right*:
$[\![\bigwedge y.\ c\ y \notin C;\ \bigwedge y.\ d\ y \in C]\!] \Longrightarrow$
$((d\ `?`\ x \to (Q\ x))\ [\![C]\!]\ (c\ `?`\ x \to (P\ x))) =$
$(c\ `?`\ x \to ((d\ `?`\ x \to (Q\ x))\ \ [\![C]\!]\ (P\ x)\ ))$
⟨*proof*⟩

lemmas *read-Par-Int-distr3 = read-read-nonsync-right*

lemma *write-read-sync*:
assumes *contained*: $\bigwedge y.\ c\ y \in C$
assumes *is-construct*: *inj c*
shows $((c\ `!`\ a \to P)[\![C]\!](c\ `?`\ x \to Q\ x)) =$
$(c\ `!`\ a \to (P[\![C]\!](Q\ a)))$
⟨*proof*⟩

lemmas *write-ParInt-read = write-read-sync*

lemma *read-write-sync*:
assumes *contained*: $\bigwedge y.\ c\ y \in C$
assumes *is-construct*: *inj c*
shows $((c\ `?`\ x \to P\ x)[\![C]\!](c\ `!`\ a \to Q)) =$

$(c\ \text{`!`}\ a \to ((P\ a)\ [\![C]\!]\ Q))$
⟨*proof*⟩

lemmas *read-ParInt-write = read-write-sync*

lemma *write-read-nonsync-left*:
$[\![d\ a \notin C;\ \bigwedge y.\ c\ y \in C]\!] \Longrightarrow$
$((d\ \text{`!`}\ a \to P)[\![C]\!](c\ \text{`?`}\ x \to Q\ x)) =$
$(d\ \text{`!`}\ a \to (P[\![C]\!](c\ \text{`?`}\ x \to Q\ x)))$
⟨*proof*⟩

lemmas *write-ParInt-read2 = write-read-nonsync-left*

lemma *write0-read-nonsync-left* :
$[\![d \in C;\ \bigwedge y.\ c\ y \notin C]\!] \Longrightarrow$
$((d \to P)[\![C]\!](c\ \text{`?`}\ x \to Q\ x)) =$
$(c\ \text{`?`}\ x \to ((d \to P)[\![C]\!]Q\ x))$
⟨*proof*⟩

lemmas *prefix-ParInt-read2 = write0-read-nonsync-left*

lemma *read-write0-nonsync-left*:
$[\![d \in C;\ \bigwedge y.\ c\ y \notin C]\!] \Longrightarrow$
$((c\ \text{`?`}\ x \to Q\ x)[\![C]\!](d \to P)) =$
$(c\ \text{`?`}\ x \to (Q\ x[\![C]\!](d \to P)))$
⟨*proof*⟩

lemma *write0-write-nonsync-right*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow$
$((c \to Q)[\![C]\!](d\ \text{`!`}\ a \to P)) =$
$(d\ \text{`!`}\ a \to ((c \to Q)[\![C]\!]P))$
⟨*proof*⟩

lemmas *prefix-ParInt-write2 = write0-write-nonsync-right*

lemma *write-write0-nonsync-left*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow$
$((d\ \text{`!`}\ a \to P)[\![C]\!](c \to Q)) =$
$(d\ \text{`!`}\ a \to (P[\![C]\!](c \to Q)))$
⟨*proof*⟩

lemmas *write-ParInt-prefix2 = write-write0-nonsync-left*

lemma *write0-write0-sync* :
$c \in C \Longrightarrow ((c \to P)[\![C]\!](c \to Q)) = (c \to (P[\![C]\!]Q))$
⟨*proof*⟩

lemmas *sync-rules* =
read-read-sync read-read-nonsync-left read-read-nonsync-right
write-read-sync read-write-sync write-read-nonsync-left
write0-read-nonsync-left read-write0-nonsync-left
write0-write-nonsync-right write-write0-nonsync-left
write0-write0-sync

lemma *no-hide-read-1*:
$(\bigwedge y.\ c\ y \notin B) \Longrightarrow$
$((c\ \text{`?`}\ x \rightarrow (P\ x)) \setminus B) = (c\ \text{`?`}\ x \rightarrow ((P\ x) \setminus B))$
⟨*proof*⟩

lemmas *hide-read-distr1* = *no-hide-read-1*

lemma *no-hide-write*:
$(\bigwedge y.\ c\ y \notin B) \Longrightarrow ((c\ \text{`!`}\ a \rightarrow P) \setminus B) = (c\ \text{`!`}\ a \rightarrow (P \setminus B))$
⟨*proof*⟩

lemmas *hide-write-distr1* = *no-hide-write*

lemma *hide-write*:
$(c\ a) \in B \Longrightarrow ((c\ \text{`!`}\ a \rightarrow P) \setminus B) = (P \setminus B)$
⟨*proof*⟩

lemmas *hide-write-distr2* = *hide-write*

lemma *hide-write0*:
$c \in B \Longrightarrow ((c \rightarrow P) \setminus B) = (P \setminus B)$
⟨*proof*⟩

lemmas *hide-rules* = *no-hide-read-1 no-hide-write hide-write hide-write0*

lemma *mono-read-ref*:
$(\bigwedge x.\ P\ x \sqsubseteq Q\ x) \Longrightarrow\ (c\ \text{`?`}\ x \rightarrow (P\ x)) \sqsubseteq (c\ \text{`?`}\ x \rightarrow (Q\ x))$
⟨*proof*⟩

lemma *mono-write-ref*:
$(P \sqsubseteq Q) \Longrightarrow\ (c\ \text{`!`}\ x \rightarrow P) \sqsubseteq (c\ \text{`!`}\ x \rightarrow Q)$
⟨*proof*⟩

lemma *mono-write0-ref*:
$(P \sqsubseteq Q) \Longrightarrow\ (c \rightarrow P) \sqsubseteq (c \rightarrow Q)$
⟨*proof*⟩

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

lemmas *Det-commute* = *det-commute*
lemmas *non-det-id* = *ndet-id*
lemmas *Ndet-commute* = *ndet-commute*
lemmas *non-det-bot* = *ndet-bot*

12 Operational Semantics

datatype $'\alpha$ *sevent* = *sevent* $'\alpha$ *event* | τ

inductive *op-sem* :: [$'\alpha$ *process*, $'\alpha$ *sevent*, $'\alpha$ *process*] => *bool*
(- $\longrightarrow$ - - [0,0,60] 60)
where *refl* : $P \longrightarrow \tau\ P$
| *skip* : $SKIP \longrightarrow (sevent(tick))\ Bot$
| *mpref* : $y \in A \Longrightarrow (\Box\ x{\in}A \rightarrow P\ x\ ) \longrightarrow (sevent(ev\ y))\ (P\ y)$
| *refine* : $P \sqsubseteq Q \Longrightarrow Q \longrightarrow a\ Q' \Longrightarrow P \longrightarrow a\ Q'$

end

13 Example: Refinement Example with Buffer over infinite Alphabet

theory *CopyBuffer*
imports *../src/CSP*
begin

14 Defining the Copy-Buffer Example

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

definition *SYN* :: ($'a$ *channel*) *set*
where *SYN* $\equiv$ (*range mid*) $\cup$ {*ack*}

definition *COPY* :: (*'a channel*) *process*
where *COPY* ≡ (*μ COPY*. *left‘?‘x* → *right‘!‘x* → *COPY*)

definition *SEND* :: (*'a channel*) *process*
where *SEND* ≡ (*μ SEND*. *left‘?‘x* → *mid‘!‘x* → *ack* → *SEND*)

definition *REC* :: (*'a channel*) *process*
where *REC* ≡ (*μ REC*. *mid‘?‘x* → *right‘!‘x* → *ack* → *REC*)

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

15 The Standard Proof

15.1 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 [*simp*]: *left x* ∉ *SYN*
⟨*proof*⟩

lemma [*simp*]: *right x* ∉ *SYN*
⟨*proof*⟩

lemma [*simp*]: *ack* ∈ *SYN*
⟨*proof*⟩

lemma [*simp*]: *mid x* ∈ *SYN*
⟨*proof*⟩

lemma [*simp*]: *inj mid*
⟨*proof*⟩

15.2 Definitions by Recursors

Second part: Derive recursive process equations, which are easier to handle in proofs. This part IS actually automated if we could reuse the fixrec-syntax below.

lemma *COPY-rec*:
(*COPY*::*'a channel process*) = (*left‘?‘x* → *right‘!‘x* → *COPY*)
⟨*proof*⟩

lemma *SEND-rec*:
SEND = (*left‘?‘x* → *mid‘!‘x* → *ack* → *SEND*)
⟨*proof*⟩

lemma *REC-rec*:
REC = (*mid*‘?‘*x* → *right*‘!‘*x* → *ack* → *REC*)
⟨*proof*⟩

15.3 A Refinement Proof

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma *impl-refines-spec* : (*COPY*::*'a channel process*) ⊑ *SYSTEM*
⟨*proof*⟩

lemma *spec-refines-impl* :
assumes *fin*: *finite* (*SYN*::(*'a channel*)*set*)
shows *SYSTEM* ⊑ (*COPY*::*'a channel process*)
⟨*proof*⟩

Note that this was actually proven for the Process ordering, not the refinement ordering. But the former implies the latter. And due o anti-symmetrie, 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*)
⟨*proof*⟩

16 An Alternative Approach: Using the fixrec-Package

16.1 Channels and Synchronisation Sets

As before.

16.2 Process Definitions via fixrec-Package

fixrec
COPY' :: (*'a channel*) *process*
and
SEND' :: (*'a channel*) *process*
and
REC' :: (*'a channel*) *process*
where
COPY'-rec[*simp del*]: *COPY'* = (*left*‘?‘*x* → *right*‘!‘*x* → *COPY'*)
| *SEND'*-rec[*simp del*]: *SEND'* = (*left*‘?‘*x* → *mid*‘!‘*x* → *ack* → *SEND'*)
| *REC'*-rec[*simp del*] : *REC'* = (*mid*‘?‘*x* → *right*‘!‘*x* → *ack* → *REC'*)

find-theorems *name*:*COPY*

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

16.3 Another Refinement Proof on fixrec-infrastructure

Third part: No comes the proof by fixpoint induction. Not too bad in automation considering what is inferred, but wouldn't scale for large examples.

lemma *impl-refines-spec'* : $(COPY'::'a\ channel\ process) \sqsubseteq SYSTEM'$
⟨*proof*⟩

lemma *spec-refines-impl'* :
assumes *fin*: *finite* $(SYN::('a\ channel)set)$
shows $SYSTEM' \sqsubseteq (COPY'::'a\ channel\ process)$
⟨*proof*⟩

lemma *spec-equal-impl'* :
assumes *fin*: *finite* $(SYN::('a\ channel)set)$
shows $SYSTEM' = (COPY'::'a\ channel\ process)$
⟨*proof*⟩

end

References

[1] A. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, 1998.

[2] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.