HOL-CSP Version 2.0

Benoit Ballenghien Safouan Taha Burkhart Wolff
Lina Ye

May 26, 2024

Contents

Chapter 1

Context

1.1 Preface

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 "Theory and Practice of Concurrency" [12] and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes" [2].
The original version of this formalization, called HOL-CSP 1.0 [13], revealed minor, but omnipresent foundational errors in key concepts like the process invariant. A correction was proposed slightly before the apparition of Roscoe's book (all three authors were in e-mail contact at that time).

In recent years, a team of authors undertook the task to port HOL-CSP 1.0 to modern Isabelle/HOL and structured proof techniques. This results in the present version are called HOL-CSP 2.0.
The effort is motivated by the following assets of CSP:

- the theory is interesting in itself, and reworking its formal structure might help to make it more widely accessible, given that it is a particularly advanced example of the shallow embedding technique using the denotational semantics of a language,
- it is interesting to *compare* the ancient, imperative, ML-heavy proof style to the more recent declarative one in Isabelle/Isar; this comparison (not presented here) gives a source of empirical evidence that such proofs are more stable wrt. the constant changes in the Isabelle itself,
- the *semantic* presentation of CSP lends itself to a semantically clean and well-understood *combination* of specification languages, which represents a major step to our longterm goal of heterogenuous, yet seman-

tically clean system specifications consisting of different formalisms describing components or system aspects separately,

- the resulting HOL-CSP environment could one day be used as a tool that certifies traces of other CSP model-checkers such as FDR4 or PAT.

In contrast to HOL-CSP 1.0, which came with an own fixpoint theory partly inspired by previous work of Alberto Camilieri under HOL4 [3], it is the goal of the redesign of HOL-CSP 2.0 to reuse the HOLCF theory that emmerged from Franz Regensburgers work and was substantially extended by Brian Huffman. Thus, the footprint of the HOL-CSP 2.0 theory should be reduced drastically. Moreover, all proofs have been heavily revised or reconstructed to reflect the drastically improved state of the art of interactive theory development with Isabelle.

1.2 Introduction

DRAWN FROM THE PAPER [13]

In his invited lecture at FME'96, C.A.R. Hoare presented his view on the status quo of formal methods in industry. With respect to formal proof methods, he ruled that they "are now sufficiently advanced that a [...] formal methodologist could occasionally detect [...] obscure latent errors before they occur in practice" and asked for their publication as a possible "milestone in the acceptance of formal methods" in industry.

In this paper, we report of a larger verification effort as part of the UniForM project [7]. It revealed an obscure latent error that was not detected within a decade. It cannot be said that the object of interest is a "large software system" whose failure may "cost millions", but it is a well-known subject in the center of academic interest considered foundational for several formal methods tools: the theory of the failure- divergence model of CSP ([5], [2]). And indeed we hope that this work may further encourage the use of formal proof methods at least in the academic community working on formal methods.

Implementations of proof support for a formal method can roughly be divided into two categories. In direct tools like FDR [1], the logical rules of a method (possibly integrated into complex proof techniques) are hard-wired into the code of their implementation. Such tools tend to be difficult to modify and to formally reason about, but can possess enviable automatic proof power in specific problem domains and comfortable user interfaces.

The other category can be labelled as logical embeddings. Formal methods such as CSP or Z can be logically embedded into an LCF-style tactical theorem prover such as HOL [4] or Isabelle[8]. Coming with an open system

design going back to Milner, these provers allow for user-programmed extensions in a logically sound way. Their strength is flexibility, generality and expressiveness that makes them to symbolic programming environments.

In this paper we present a tool of the latter category (as a step towards a future combination with the former). After a brief introduction into the failure divergence semantics in the traditional CSP-literature, we will discuss the revealed problems and present a correction. Although the error is not "mathematically deep", it stings since its correction affects many definitions. It is shown that the corrected CSP still fulfils the desired algebraic laws. The addition of fixpoint-theory and specialised tactics extends the embedding in Isabelle/HOL to a formally proven consistent proof environment for CSP. Its use is demonstrated in a final example.

1.3 An Outline of Failure-Divergence Semantics

A very first approach to give denotational semantics to CSP is to view it as a kind of a regular expression. This way, it can be understood as an automata and the denotations are just the language of the automaton; this way, synchronization and concurrency can be basically understood as the construction of a product automaton with potential interleaving. The semantics becomes compositional, and internal communication between subcomponents of a component can be modeled by the concealment operator.

Hoares work [5] was strongly inspired by this initial idea. However, it became quickly clear that the simplistic automata vision is not a satisfying paradigm for all aspects of concurrency. Particularly regarding the nature of communication, where one "sends" actively information and the other "receives" it, the bi-directional product construction seems to be misleading. Furthermore, it is an obvious difference if a group of processes remains in a passive deadlock because all possible communications contradict each other, or if a group of processes is too busy with internal chatter and never reaches the point where this component is again ready for communication.

Hoare solved these apparent problems by presenting a multi-layer approach, in which the denotational models were refined more and more allowing to distinguish the above critical situations. An ingenious concept in the overall scheme is to distinguish *non-deterministic* choice from *deterministic* one [1] in order to solve the sender/receiver problem.

Hoare proposed 3 denotational semantics for CSP:

- the *trace* model, which is basically the above naive automata model not allowing to distinguish non-deterministic choice from deterministic

[1] which in itself produces problems with recursion which had to be overcome by some restrictions on its use.

one, neither to distinguish deadlock from infinite internal chatter,

- the *failure* model is able to distinguish non-deterministic choice from deterministic one by different maximum refusal sets, which is however cannot differentiate deadlock from infinite internal chatter,
- the *failure-divergence* model overcomes additionally the unresolved problem of failure model.

In the sequel, we explain these two problems in more detail, giving some motivation for the daunting complexity of the latter model. It is this complexity which finally raises general interest in a formal verification.

1.3.1 Non-Determinism

Let a and b be any two events in some set of events Σ. The two processes

$$(a \rightarrow \mathit{Stop}) \,\Box\, b \rightarrow \mathit{Stop} \tag{1}$$

and

$$(a \rightarrow \mathit{Stop}) \sqcap b \rightarrow \mathit{Stop} \tag{2}$$

cannot be distinguished under the trace semantics, in which both processes are capable of performing the same sequences of events, i.e. both have the same set of traces $\{\{\}, \{a\}, \{b\}\}$. This is because both processes can either engage in a and then *Stop*, or engage in b and then *Stop*. We would, however, like to distinguish between *a deterministic* choice of a or b (1) and *a non-deterministic* choice of a or b (2).

This can be done by considering the events that a process can refuse to engage in when these events are offered by the environment; it cannot refuse either, so we say its maximal refusal set is the set containing all elements of Σ other than a and b, written $\Sigma - \{a, b\}$, i.e. it can refuse all elements in Σ other than a or b. In the case of the non-deterministic process (2), however, we wish to express that if the environment offers the event a say, the process non-deterministically chooses either to engage in a, to refuse it and engage in b (likewise for b). We say therefore, that process (2) has two maximal refusal sets, $\Sigma - \{a\}$ and $\Sigma - \{b\}$, because it can refuse to engage in either a or b, but not both. The notion of refusal sets is in this way used to distinguish non-determinism from determinism in choices.

1.3.2 Infinite Chatter

Consider the infinite process $\mu\ x.\ a \rightarrow x$ which performs an infinite stream of a's. If one now conceals the event a in this process by writing

$$(\mu\ x.\ a \rightarrow x) \setminus a \tag{3}$$

it no longer becomes possible to distinguish the behaviour of this process from that of the deadlock process Stop. We would like to be able to make such a distinction, since the former process has clearly not stopped but is engaging in an unbounded sequence of internal actions invisible to the environment. We say the process has diverged, and introduce the notion of a divergence set to denote all sequences events that can cause a process to diverge. Hence, the process Stop is assigned the divergence set {}, since it can not diverge, whereas the process (3) above diverges on any sequence of events since the process begins to diverge immediately, i.e. its divergence set is Σ^* , where Σ^* denotes the set of all sequences with elements in Σ. Divergence is undesirable and so it is essential to be able to express it to ensure that it is avoided.

1.3.3 The Global Architecture of HOL-CSP 2.0

The global architecture of HOL-CSP 2.0 has been substantially simplified compared to HOL-CSP 1.0: the fixpoint reasoning is now entirely based on HOLCF (which meant that the continuity proofs for CSP operators had basically been re-done).

The theory `Process` establishes the basic common notions for events, traces, ticks and tickfree-ness, the type definitions for failures and divergences as well as the global constraints on them (called the "axioms" in Hoare's Book.) captured in a predicate `is_process`. On this basis, the set of failures and divergences satisfying `is_process` is turned into the type `'a process` via a type-definition (making `is_process` as the central data invariant). In the sequel, it is shown that `'a process` belongs to the type-class *cpo* stemming from *HOLCF* which makes the concepts of complete partial order, continuity, fixpoint-induction and general recursion available to all expressions of type `'a process`.

The theory `Process` also establishes the two partial orderings $P \leq P'$ for refinements and $P \sqsubseteq P'$ for the approximation on processes used to give semantics to recursion. The latter is well-known to be logically weaker than the former. Note that, unfortunately, the use of these two symbols in HOL-CSP 2.0, where the latter is already used in the *HOLCF*-theory, is just the other way round as in the literature.

Each CSP operator is described in an own theory which comprises:

- The denotational core definition in terms of a pair of Failures and Divergences

- The establishment of `is_process` for the Failures and Divergences in the range of the given operator (thus, the invariance of `is_process` for this operator). In this new version, the proof is required immediatly after the definition of the operator since we use lift_definition instead of definition
- The proof of the projections `T` and `F` and `D` for this operator
- The proof of continuity of the operator (which is always possible except for Hiding if applied to infinite hide-sets).

Finally, the theory CSP contains the "Laws" of CSP, i.e. the derived rules allowing abstractly to reason over CSP processes. The overall dependency graph is shown in Figure 1.1.

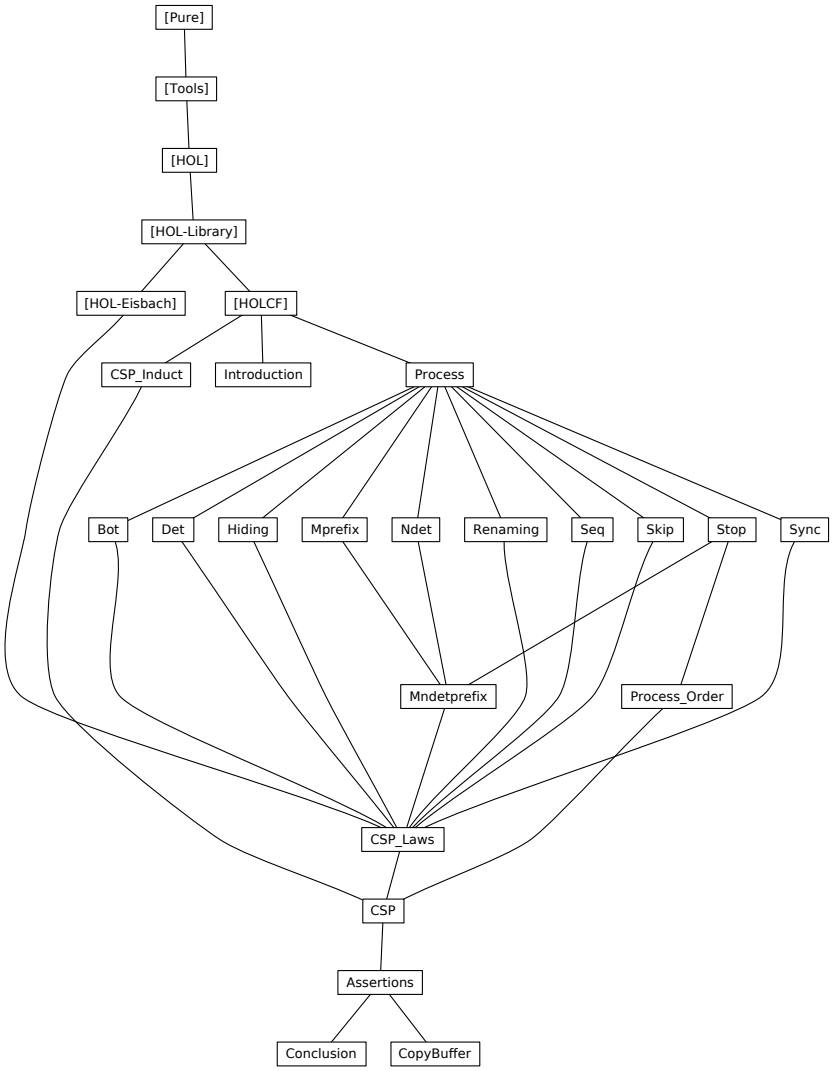

Figure 1.1: The HOL-CSP 2.0 Theory Graph

Chapter 2

The Notion of Processes

As mentioned earlier, we base the theory of CSP on HOLCF, a Isabelle/HOL library providing a theory of continuous functions, fixpoint induction and recursion.

theory *Process*
  imports *HOLCF*
begin

Since HOLCF sets the default type class to *cpo*, while our Process theory establishes links between standard types and *pcpo* types. Consequently, we reset the default type class to the default in HOL.

default-sort *type*

2.1 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 of traces in order to signalize successful 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 [13] for details.)

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

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

We chose as standard ordering on traces the prefix ordering.

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

definition *le-list-def* : (*s*::$'a$ *list*) $\leq$ *t* $\longleftrightarrow$ ($\exists$ *r*. *s* @ *r* = *t*)

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

lemma A : $((x::'a\ list) < y) = (x \leq y \wedge \neg\ y \leq x)$
⟨*proof*⟩

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 *less-cons*: $s < t \Longrightarrow a\ \#\ s < a\ \#\ t$
⟨*proof*⟩

lemma *less-append*: $s < t \Longrightarrow a\ @\ s < a\ @\ t$
⟨*proof*⟩

lemma *less-tail*: $s \neq [] \Longrightarrow s < t \Longrightarrow tl\ s < tl\ t$
⟨*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*⟩

lemma *prefixes-fin*: *let prefixes* = $\{t.\ \exists\, t2.\ x = t\ @\ t2\}$ *in finite prefixes* $\wedge$ *card prefixes* = *length x* + *1*
⟨*proof*⟩

lemma *sublists-fin*: *finite* $\{t.\ \exists\, t1\ t2.\ x = t1\ @\ t\ @\ t2\}$
⟨*proof*⟩

lemma *suffixes-fin*: *finite* $\{t.\ \exists\, t1.\ x = t1\ @\ t\}$
⟨*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* : *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* $(t \mathbin{@} [a]) \Longrightarrow$ *tickFree* t
⟨*proof*⟩

lemma *tickFree-implies-front-tickFree-single*: *tickFree* $t \Longrightarrow$ *front-tickFree* $(t \mathbin{@} [a])$
⟨*proof*⟩

lemma *nonTickFree-n-frontTickFree*: $[\![\neg$ *tickFree* s; *front-tickFree* $s\ ]\!] \Longrightarrow \exists\, t.\ s = t \mathbin{@} [tick]$
⟨*proof*⟩

lemma *front-tickFree-dw-closed* : *front-tickFree* $(s \mathbin{@} t) \Longrightarrow$ *front-tickFree* s
⟨*proof*⟩

lemma *front-tickFree-append*: $[\![$ *tickFree* s; *front-tickFree* $t]\!] \Longrightarrow$ *front-tickFree* $(s \mathbin{@} t)$
⟨*proof*⟩

lemma *front-tickFree-mono*: *front-tickFree* $(t \mathbin{@} r) \wedge r \neq [] \longrightarrow$ *tickFree* $t \wedge$ *front-tickFree* r
⟨*proof*⟩

lemma *tickFree-butlast*: ‹*tickFree* $s \longleftrightarrow$ *tickFree* $(butlast\ s) \wedge (s \neq [] \longrightarrow last\ s \neq tick)$›
⟨*proof*⟩

lemma *front-tickFree-butlast*: ‹*front-tickFree* $s \longleftrightarrow$ *tickFree* $(butlast\ s)$›
⟨*proof*⟩

2.2 Basic Types, Traces, Failures and Divergences

type-synonym $'\alpha$ *refusal* = $('\alpha$ *event*$)$ *set*
type-synonym $'\alpha$ *failure* = $'\alpha$ *trace* $\times$ $'\alpha$ *refusal*
type-synonym $'\alpha$ *divergence* = $'\alpha$ *trace set*
type-synonym $'\alpha$ *process*$_0$ = $'\alpha$ *failure set* $\times$ $'\alpha$ *divergence*

definition *FAILURES* :: $'\alpha$ *process*$_0 \Rightarrow ('\alpha$ *failure set*$)$
where *FAILURES* P = *fst* P

definition *TRACES* :: $'\alpha$ *process*$_0 \Rightarrow ('\alpha$ *trace set*$)$
where *TRACES* $P = \{tr.\ \exists\ a.\ a \in FAILURES\ P \wedge tr = fst\ a\}$

definition *DIVERGENCES* :: $'\alpha$ *process*$_0 \Rightarrow '\alpha$ *divergence*
where *DIVERGENCES* P = *snd* P

definition *REFUSALS* :: $'\alpha$ *process*$_0 \Rightarrow ('\alpha$ *refusal set*$)$
where *REFUSALS* $P = \{ref.\ \exists\ F.\ F \in FAILURES\ P \wedge F = ([],ref)\}$

2.3 The Process Type Invariant

definition *is-process* :: $'\alpha$ *process*$_0$ $\Rightarrow$ *bool* **where**
is-process P =
(([],{}) $\in$ *FAILURES P* $\wedge$
($\forall$ *s X*. (*s*,*X*) $\in$ *FAILURES P* $\longrightarrow$ *front-tickFree s*) $\wedge$
($\forall$ *s t* . (*s*@*t*,{}) $\in$ *FAILURES P* $\longrightarrow$ (*s*,{}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*Y*) $\in$ *FAILURES P* & *X* <= *Y* $\longrightarrow$ (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*X*) $\in$ *FAILURES P* $\wedge$
($\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ ((*s*@[*c*],{})$\notin$*FAILURES P*)) $\longrightarrow$
(*s*,*X* $\cup$ *Y*)$\in$*FAILURES P*) $\wedge$
($\forall$ *s X*. (*s*@[*tick*],{}) : *FAILURES P* $\longrightarrow$ (*s*,*X*$-${*tick*}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s t*. *s* $\in$ *DIVERGENCES P* $\wedge$ *tickFree s* $\wedge$ *front-tickFree t*
$\longrightarrow$ *s*@*t* $\in$ *DIVERGENCES P*) $\wedge$
($\forall$ *s X*. *s* $\in$ *DIVERGENCES P* $\longrightarrow$ (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s*. *s* @ [*tick*] : *DIVERGENCES P* $\longrightarrow$ *s* $\in$ *DIVERGENCES P*))

lemma *is-process-spec*:
is-process P =
(([],{}) $\in$ *FAILURES P* $\wedge$
($\forall$ *s X*. (*s*,*X*) $\in$ *FAILURES P* $\longrightarrow$ *front-tickFree s*) $\wedge$
($\forall$ *s t* . (*s* @ *t*,{}) $\notin$ *FAILURES P* $\vee$ (*s*,{}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*. (*s*,*Y*) $\notin$ *FAILURES P* $\vee$ $\neg$(*X*$\subseteq$*Y*) | (*s*,*X*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X Y*.(*s*,*X*) $\in$ *FAILURES P* $\wedge$
($\forall$ *c*. *c* $\in$ *Y* $\longrightarrow$ ((*s*@[*c*],{}) $\notin$ *FAILURES P*)) $\longrightarrow$(*s*,*X* $\cup$ *Y*) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s X*. (*s*@[*tick*],{}) $\in$ *FAILURES P* $\longrightarrow$ (*s*,*X* $-$ {*tick*}) $\in$ *FAILURES P*) $\wedge$
($\forall$ *s t*. *s* $\notin$ *DIVERGENCES P* $\vee$ $\neg$*tickFree s* $\vee$ $\neg$*front-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*$_0$) = *Q*
⟨*proof*⟩

lemma *process-eq-spec*:
((*P*::*'a process*$_0$) = *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*: $[\![is\text{-}process\ P;\ (s,\ Y) \in FAILURES\ P;\ X \subseteq Y]\!] \Longrightarrow (s,\ X) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process4-S1*: $[\![is\text{-}process\ P;\ x \in FAILURES\ P;\ X \subseteq snd\ x]\!] \Longrightarrow (fst\ x,\ X) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process5*:
$is\text{-}process\ P \Longrightarrow$
$\quad \forall\ sa\ X\ Y.\ (sa,\ X) \in FAILURES\ P \wedge (\forall\ c.\ c \in Y \longrightarrow (sa\ @\ [c],\ \{\}) \notin FAILURES\ P) \longrightarrow (sa,\ X \cup Y) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process5-S*:
$[\![is\text{-}process\ P;\ (sa,\ X) \in FAILURES\ P;\ \forall\ c.\ c \in Y \longrightarrow (sa\ @\ [c],\ \{\}) \notin FAILURES\ P]\!] \Longrightarrow (sa,\ X \cup Y) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process5-S1*:
$[\![is\text{-}process\ P;\ (sa,\ X) \in FAILURES\ P;\ (sa,\ X \cup Y) \notin FAILURES\ P]\!] \Longrightarrow \exists\ c.\ c \in Y \wedge (sa\ @\ [c],\ \{\}) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process6*: $is\text{-}process\ P \Longrightarrow\ \forall\ s\ X.\ (s@[tick],\{\}) \in FAILURES\ P \longrightarrow$

$(s,X-\{tick\}) \in FAILURES\ P$
⟨*proof*⟩

lemma *is-process6-S*: $[\![is\text{-}process\ P\ ;(s@[tick],\{\}) \in FAILURES\ P]\!] \Longrightarrow (s,X-\{tick\})$ $\in FAILURES\ P$
⟨*proof*⟩

lemma *is-process7*:
$is\text{-}process\ P \Longrightarrow \forall\ s\ t.\ s \notin DIVERGENCES\ P \vee \neg\ tickFree\ s \vee \neg\ front\text{-}tickFree\ t$ $\vee\ s\ @\ t \in DIVERGENCES\ P$
⟨*proof*⟩

lemma *is-process7-S*:
$[\![\ is\text{-}process\ P;s : DIVERGENCES\ P;tickFree\ s;front\text{-}tickFree\ t]\!]$
$\Longrightarrow\ s\ @\ t \in DIVERGENCES\ P$
⟨*proof*⟩

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

lemma *is-process8-S*: $[\![\ is\text{-}process\ P;\ s \in DIVERGENCES\ P\ ]\!] \Longrightarrow (s,X)\ \ \in$ $FAILURES\ P$
⟨*proof*⟩

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

lemma *is-process9-S*: $[\![\ is\text{-}process\ P;s@[tick] \in DIVERGENCES\ P\ ]\!] \Longrightarrow s \in DI$- $VERGENCES\ P$
⟨*proof*⟩

lemma *Failures-implies-Traces*: $[\![is\text{-}process\ P;\ (s,\ X) \in FAILURES\ P]\!] \Longrightarrow s \in$ $TRACES\ P$
⟨*proof*⟩

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

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

lemma *front-trace-is-tickfree*:

assumes *A*: *is-process P* **and** *B*: $(t \mathbin{@} [tick], X) \in FAILURES\ P$
shows *tickFree t*
⟨*proof*⟩

lemma *trace-with-Tick-implies-tickFree-front* : $[\![is\text{-}process\ P;\ t \mathbin{@} [tick] \in TRACES\ P]\!] \Longrightarrow tickFree\ t$
⟨*proof*⟩

2.4 The Abstraction to the process-Type

typedef
$'\alpha\ process = \{p :: '\alpha\ process_0\ .\ is\text{-}process\ p\}$
⟨*proof*⟩

setup-lifting *type-definition-process*

This is where we differ from previous versions: we lift definitions using Isabelle's machinery instead of doing it by hand.

lift-definition *Failures* :: $'\alpha\ process \Rightarrow ('\alpha\ failure\ set)$ $(\mathcal{F})$
is $\lambda P.\ FAILURES\ P$ ⟨*proof*⟩

lift-definition *Traces* :: $'\alpha\ process \Rightarrow\ ('\alpha\ trace\ set)$ $(\mathcal{T})$
is $\lambda P.\ TRACES\ P$ ⟨*proof*⟩

lift-definition *Divergences* :: $'\alpha\ process \Rightarrow '\alpha\ divergence$ $(\mathcal{D})$
is $\lambda P.\ DIVERGENCES\ P$ ⟨*proof*⟩

lift-definition *Refusals* :: $'\alpha\ process \Rightarrow ('\alpha\ refusal\ set)$ $(\mathcal{R})$
is $\lambda P.\ REFUSALS\ P$ ⟨*proof*⟩

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

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

lemma *Process-eq-spec*: $(P = Q) = (\mathcal{F}\ P = \mathcal{F}\ Q \wedge \mathcal{D}\ P = \mathcal{D}\ Q)$
⟨*proof*⟩

lemma *Process-eq-spec-optimized*: $(P = Q) = (\mathcal{D}\ P = \mathcal{D}\ Q \wedge (\mathcal{D}\ P = \mathcal{D}\ Q \longrightarrow \mathcal{F}\ P = \mathcal{F}\ Q))$
⟨*proof*⟩

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

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

⟨*proof*⟩

split of `is_processT`:

lemma *is-processT1*: $([],\ \{\}) \in \mathcal{F}\ P$
⟨*proof*⟩

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

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

lemma *is-proT2*:
assumes A : $(s,\ X) \in \mathcal{F}\ P$ **and** B : $s \neq []$
shows $\mathit{tick} \notin \mathit{set}\ (\mathit{tl}\ (\mathit{rev}\ s))$
⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5 Some Consequences of the Process Characterization

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

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

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

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

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

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

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

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

lemma *is-processT6-S1*: $tick \notin X \Longrightarrow (s\ @\ [tick],\ \{\}) \in \mathcal{F}\ P \Longrightarrow (s,\ X) \in \mathcal{F}\ P$
⟨*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* : $\mathcal{D}\ P \subseteq \mathcal{T}\ P$ ⟨*proof*⟩

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

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

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

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

lemma *is-processT5-S7'*:
$(t,\ X) \in \mathcal{F}\ P \Longrightarrow (t,\ X \cup A) \notin \mathcal{F}\ P \Longrightarrow \exists x.\ x \in A \wedge x \notin X \wedge t\ @\ [x] \in \mathcal{T}\ P$
⟨*proof*⟩

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

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

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

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

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

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

lemma *append-T-imp-tickFree*: $t \mathbin{@} s \in \mathcal{T}\ P \Longrightarrow s \neq [] \Longrightarrow$ *tickFree* t
⟨*proof*⟩

corollary *append-single-T-imp-tickFree* : $t \mathbin{@} [a] \in \mathcal{T}\ P \Longrightarrow$ *tickFree* t
⟨*proof*⟩

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

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

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

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

2.6 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-elems* A
⟨*proof*⟩

lemma *min-elems-le-self*[*simp*] : (*min-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*: $(x::'a\ list) \in A$
shows $\exists\, s{\leq}x.\ s \in min\text{-}elems\ A$
⟨*proof*⟩

lemma *min-elems4*: $A \neq \{\} \Longrightarrow \exists\, s.\ (s :: 'a\ trace) \in min\text{-}elems\ A$
⟨*proof*⟩

lemma *min-elems-charn*: $t \in A \Longrightarrow \exists\ t'\ r.\ t = (t' \mathbin{@} r) \wedge t' \in min\text{-}elems\ A$
⟨*proof*⟩

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

lemma *min-elems-no*: $(x::'a\ list) \in min\text{-}elems\ A \Longrightarrow t \in A \Longrightarrow t \leq x \Longrightarrow x = t$
⟨*proof*⟩

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

definition *Ra* :: $['\alpha\ process, '\alpha\ trace] \Rightarrow ('\alpha\ refusal\ set)\ (\mathcal{R}_a)$
where $\mathcal{R}_a\ P\ s = \{X.\ (s, X) \in \mathcal{F}\ P\}$

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 $\_ \sqsubseteq \_$ also written `_ << _`.

definition *le-approx-def* : $P \sqsubseteq Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P \wedge$
$(\forall\, s.\ s \notin \mathcal{D}\ P \longrightarrow \mathcal{R}_a\ P\ s = \mathcal{R}_a\ Q\ s) \wedge$
$min\text{-}elems\ (\mathcal{D}\ P) \subseteq \mathcal{T}\ Q$

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*: $P{\sqsubseteq}Q \Longrightarrow \mathcal{D}\ Q \subseteq \mathcal{D}\ P$
⟨*proof*⟩

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

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

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

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

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

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

lemma *proc-ord2a* : $P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow ((s,\ X) \in \mathcal{F}\ P) = ((s,\ X) \in \mathcal{F}\ Q)$
⟨*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.

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` : $[\![\mathit{chain}\ ?Y;\ ?i \leq ?j]\!] \Longrightarrow ?Y\ ?i \sqsubseteq ?Y\ ?j$
- `Porder.is_ubD` : $[\![?S <|\ ?u;\ ?x \in ?S]\!] \Longrightarrow ?x \sqsubseteq ?u$
- `Porder.ub_rangeI` :
  $(\bigwedge i.\ ?S\ i \sqsubseteq ?x) \Longrightarrow \mathit{range}\ ?S <|\ ?x$
- `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* :: ('α *process*) *set* $\Rightarrow$ 'α *process*
where *lim-proc* $(X) =$ *Abs-process* $(\bigcap\ (\mathcal{F}\ `\ X), \bigcap\ (\mathcal{D}\ `\ X))$

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

lemma *min-elems1*: $s \notin \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in \mathcal{D}\ P \Longrightarrow s\ @\ [c] \in$ *min-elems* $(\mathcal{D}\ P)$
⟨*proof*⟩

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

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

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

lemma *ND-F-dir2'*: $s \notin \mathcal{D}\ P \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow P \sqsubseteq S \Longrightarrow Q \sqsubseteq S \Longrightarrow s \in \mathcal{T}\ Q$
⟨*proof*⟩

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

lemma *is-process-REP-LUB*:
assumes *chain*: *chain S*
shows *is-process* $(\bigcap\ (\mathcal{F}\ `$ *range S*$), \bigcap\ (\mathcal{D}\ `$ *range S*$))$

⟨*proof*⟩

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

simplified]

lemma *F-LUB*: *chain S* $\Longrightarrow \mathcal{F}(\textit{lim-proc}(\textit{range } S)) = \bigcap (\mathcal{F} \text{ ` } \textit{range } S)$
⟨*proof*⟩

lemma *D-LUB*: *chain S* $\Longrightarrow \mathcal{D}(\textit{lim-proc}(\textit{range } S)) = \bigcap (\mathcal{D} \text{ ` } \textit{range } S)$
⟨*proof*⟩

lemma *T-LUB*: *chain S* $\Longrightarrow \mathcal{T}\ (\textit{lim-proc}(\textit{range } S)) = \bigcap (\mathcal{T} \text{ ` } \textit{range } S)$
⟨*proof*⟩

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

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

2.7 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* : $P \leq Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P \wedge \mathcal{F}\ Q \subseteq \mathcal{F}\ P$

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

instance ⟨*proof*⟩

end

Note that this just another syntax to our standard process refinement order defined in the theory Process.

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

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

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

lemma *le-ref2T* : $P \leq Q \Longrightarrow \mathcal{T}\ Q \subseteq \mathcal{T}\ P$
⟨*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* ⟶ $\mathcal{D}$ *u* ⊆ $\mathcal{D}$ (*lim-proc* (*range S*)))
⟨*proof*⟩

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

lemma *lim-proc-is-lub3a*: *front-tickFree s* ⟹ *s* ∉ $\mathcal{D}$ *P* ⟶ (∀ *t*. *t* ∈ $\mathcal{D}$ *P* ⟶ ¬ *t* < *s* @ [*c*])
⟨*proof*⟩

lemma *lim-proc-is-lub3b*:
assumes *1* : ∀ *x*. *x* ∈ *X* ⟶ (∀ *xa*. *xa* ∈ $\mathcal{D}$ *x* ∧ (∀ *t*. *t* ∈ $\mathcal{D}$ *x* ⟶ ¬ *t* < *xa*) ⟶ *xa* ∈ $\mathcal{T}$ *u*)
and *2* : *xa* ∈ *X*
and *3* : ∀ *xa*. *xa* ∈ *X* ⟶ *x* ∈ $\mathcal{D}$ *xa*
and *4* : ∀ *t*. (∀ *x*. *x* ∈ *X* ⟶ *t* ∈ $\mathcal{D}$ *x*) ⟶ ¬ *t* < *x*
shows *x* ∈ $\mathcal{T}$ *u*
⟨*proof*⟩

lemma *lim-proc-is-lub3c*:
assumes ∗:*chain S*
and ∗∗:*X* = *range S* — protection for range - otherwise auto unfolds and gets lost
shows ∀ *u*. *X* <| *u* ⟶ *min-elems*($\mathcal{D}$ (*lim-proc X*)) ⊆ $\mathcal{T}$ *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*⟩

instance
process :: (*type*) *cpo*

⟨*proof*⟩

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

2.8 Process Refinement is Admissible

lemma *le-adm*[*simp*]: *cont* (*u*::(*'a*::*cpo*) ⇒ *'b process*) ⟹ *monofun v* ⟹ *adm*(λ*x*. *u x* ≤ *v x*)
⟨*proof*⟩

lemmas *le-adm-cont*[*simp*] = *le-adm*[*OF - cont2mono*]

2.9 The Conditional Statement is Continuous

The conditional operator of CSP is obtained by a direct shallow embedding. Here we prove it continuous

lemma *if-then-else-cont*[*simp*]:
assumes ∗:(⋀*x*. *P x* ⟹ *cont* ((*f*::*'c* ⇒ (*'a*::*cpo*) ⇒ *'b process*) *x*))
and ∗∗:(⋀*x*. ¬ *P x* ⟹ *cont* (*g x*))
shows ⋀*x*. *cont*(λ*y*. *if P x then f x y else g x y*)
⟨*proof*⟩

end

Chapter 3

The CSP Operators

3.1 The Undefined Process

theory *Bot*
imports *Process*
begin

lift-definition *BOT* :: ‹'α *process*›
  is ‹({(*s*,*X*). *front-tickFree s*}, {*d*. *front-tickFree d*})›
  ⟨*proof*⟩

lemma *F-BOT*: $\mathcal{F}$ *BOT* = {(*s*,*X*). *front-tickFree s*}
  ⟨*proof*⟩

lemma *D-BOT*: $\mathcal{D}$ *BOT* = {*d*. *front-tickFree d*}
  ⟨*proof*⟩

lemma *T-BOT*: $\mathcal{T}$ *BOT* = {*s*. *front-tickFree s*}
  ⟨*proof*⟩

This is the key result: $\bot$ — which we know to exist from the process instantiation — is equal *BOT* .

lemma *BOT-is-UU*[*simp*]: *BOT* = $\bot$
⟨*proof*⟩

lemma *F-UU*: $\mathcal{F}$ $\bot$ = {(*s*,*X*). *front-tickFree s*}
  ⟨*proof*⟩

lemma *D-UU*: $\mathcal{D}$ $\bot$ = {*d*. *front-tickFree d*}
  ⟨*proof*⟩

lemma *T-UU*: $\mathcal{T}$ $\bot$ = {*s*. *front-tickFree s*}
  ⟨*proof*⟩

lemma *BOT-iff-D*: ‹$P = \bot \longleftrightarrow [] \in \mathcal{D}\ P$›
⟨*proof*⟩

end

3.2 The SKIP Process

theory *Skip*
imports *Process*
begin

lift-definition *SKIP* :: ‹$'\alpha$ *process*›
is ‹$(\{(s, X).\ s = [] \wedge tick \notin X\} \cup \{(s, X).\ s = [tick]\}, \{\})$›
⟨*proof*⟩

lemma *F-SKIP*:
$\mathcal{F}\ SKIP = \{(s, X).\ s = [] \wedge tick \notin X\} \cup \{(s, X).\ s = [tick]\}$
⟨*proof*⟩

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

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

end

3.3 The STOP Process

theory *Stop*
imports *Process*
begin

lift-definition *STOP* :: ‹$'\alpha$ *process*›
is ‹$(\{(s, X).\ s = []\}, \{\})$›
⟨*proof*⟩

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

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

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

lemma *STOP-iff-T*: $‹P = STOP \longleftrightarrow \mathcal{T}\ P = \{[]\}›$
⟨*proof*⟩

end

3.4 Deterministic Choice Operator Definition

theory *Det*
imports *Process*
begin

3.4.1 The Det Operator Definition

lift-definition
Det :: $['\alpha\ process, '\alpha\ process] \Rightarrow '\alpha\ process$ (**infixl** $[+]$ 79)
is $\lambda P\ Q.\ (\ \{(s,X).\ s = [] \wedge (s,X) \in \mathcal{F}\ P \cap \mathcal{F}\ Q\}$
$\cup \{(s,X).\ s \neq [] \wedge (s,X) \in \mathcal{F}\ P \cup \mathcal{F}\ Q\}$
$\cup \{(s,X).\ s = [] \wedge s \in \mathcal{D}\ P \cup \mathcal{D}\ Q\}$
$\cup \{(s,X).\ s = [] \wedge tick \notin X \wedge [tick] \in \mathcal{T}\ P \cup \mathcal{T}\ Q\},$
$\mathcal{D}\ P \cup \mathcal{D}\ Q)$
⟨*proof*⟩

notation
Det (**infixl** $\Box$ 79)

term $‹(A \Box B) \Box D' = C›$

3.4.2 The Projections

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

lemma *D-Det*: $\mathcal{D}\ (P \Box Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$
⟨*proof*⟩

lemma *T-Det*: $\mathcal{T}\ (P \Box Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$
⟨*proof*⟩

3.4.3 Basic Laws

The following theorem of Commutativity helps to simplify the subsequent continuity proof by symmetry breaking. It is therefore already developed here:

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

3.4.4 The Continuity-Rule

lemma *mono-Det1*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q \ \Box \ S) \subseteq \mathcal{D}\ (P \ \Box \ S)$
⟨*proof*⟩

lemma *mono-Det2*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (P \ \Box \ S) \longrightarrow Ra\ (P \ \Box \ S)\ s = Ra\ (Q \ \Box \ S)\ s)$
⟨*proof*⟩

lemma *mono-Det3*: $P \sqsubseteq Q \Longrightarrow \textit{min-elems}\ (\mathcal{D}\ (P \ \Box \ S)) \subseteq \mathcal{T}\ (Q \ \Box \ S)$
⟨*proof*⟩

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

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

lemma *cont-Det0*:
assumes C : *chain* Y
shows $(\textit{lim-proc}\ (\textit{range}\ Y) \ \Box \ S) = \textit{lim-proc}\ (\textit{range}\ (\lambda i.\ Y\ i \ \Box \ S))$
⟨*proof*⟩

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

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

lemma *Det-cont*[*simp*]:
assumes f:*cont* f

and *g*:*cont g*
shows *cont* ($\lambda x.\ f\ x\ \Box\ g\ x$)
⟨*proof*⟩

end

3.5 Nondeterministic Choice Operator Definition

theory *Ndet*
imports *Process*
begin

3.5.1 The Ndet Operator Definition

lift-definition
Ndet :: ['α *process*,'α *process*] ⇒ 'α *process* (**infixl** |−| *80*)
is $\lambda P\ Q.\ (\mathcal{F}\ P \cup \mathcal{F}\ Q\ ,\ \mathcal{D}\ P \cup \mathcal{D}\ Q)$
⟨*proof*⟩

notation
Ndet (**infixl** ⊓ *80*)

3.5.2 The Projections

lemma *F-Ndet* : $\mathcal{F}\ (P \sqcap Q) = \mathcal{F}\ P \cup \mathcal{F}\ Q$
⟨*proof*⟩

lemma *D-Ndet* : $\mathcal{D}\ (P \sqcap Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$
⟨*proof*⟩

lemma *T-Ndet* : $\mathcal{T}\ (P \sqcap Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$
⟨*proof*⟩

3.5.3 Basic Laws

The commutativity of the operator helps to simplify the subsequent continuity proof and is therefore developed here:

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

3.5.4 The Continuity Rule

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

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

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

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

lemma *mono-Ndet-sym*[*simp*] : $P \sqsubseteq Q \Longrightarrow (S \sqcap P) \sqsubseteq (S \sqcap Q)$
⟨*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*[*simp*]:
assumes *f*: *cont f*
and *g*: *cont g*
shows *cont* $(\lambda x.\ f\ x \sqcap g\ x)$
⟨*proof*⟩

end

3.6 The Sequence Operator

theory *Seq*
imports *Process*
begin

3.6.1 Definition

abbreviation *div-seq P Q* $\equiv \{t1 \mathbin{@} t2 \mid t1\ t2.\ t1 \in \mathcal{D}\ P \wedge$ *tickFree t1* $\wedge$ *front-tickFree t2*$\}$
$\cup \{t1 \mathbin{@} t2 \mid t1\ t2.\ t1 \mathbin{@} [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\}$

lift-definition *Seq* :: ['*a process*,'*a process*] ⇒ '*a process* (**infixl** **;** 74)
is $\lambda P\ Q.\ (\{(t, X).\ (t, X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$
$\{(t, X).\ \exists t1\ t2.\ t = t1 \mathbin{@} t2 \wedge t1 \mathbin{@} [tick] \in \mathcal{T}\ P \wedge (t2, X) \in \mathcal{F}\ Q\} \cup$
$\{(t, X).\ t \in div\text{-}seq\ P\ Q\},$
$div\text{-}seq\ P\ Q)$
⟨*proof*⟩

3.6.2 The Projections

lemma *F-Seq*: $‹\mathcal{F}\ (P\ ;\ Q) = \{(t, X).\ (t, X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$
$\{(t1 \mathbin{@} t2, X) \mid t1\ t2\ X.\ t1 \mathbin{@} [tick] \in \mathcal{T}\ P \wedge (t2, X) \in \mathcal{F}\ Q\} \cup$

$\{(t1,\ X)\ |t1\ X.\ t1 \in \mathcal{D}\ P\}\rangle$
⟨*proof*⟩

lemma *D-Seq*: $\langle\mathcal{D}\ (P\ ;\ Q) = \mathcal{D}\ P \cup \{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{D}\ Q\}\rangle$
⟨*proof*⟩

lemma *T-Seq*: $\langle\mathcal{T}\ (P\ ;\ Q) = \{t.\ \exists X.\ (t,\ X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup \{t1\ @\ t2\ |t1\ t2.\ t1\ @\ [tick] \in \mathcal{T}\ P \wedge t2 \in \mathcal{T}\ Q\} \cup \mathcal{D}\ P\rangle$
⟨*proof*⟩

3.6.3 Continuity Rule

lemma *mono-Seq-D11*:
$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q\ ;\ S) \subseteq \mathcal{D}\ (P\ ;\ S)$
⟨*proof*⟩

lemma *mono-Seq-D12*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (P\ ;\ S) \longrightarrow Ra\ (P\ ;\ S)\ s = Ra\ (Q\ ;\ S)\ s)$
⟨*proof*⟩

lemma *minSeqInclu*:
$min\text{-}elems(\mathcal{D}\ (P\ ;\ S))$
$\subseteq min\text{-}elems(\mathcal{D}\ P) \cup \{t1@t2|t1\ t2.\ t1@[tick] \in \mathcal{T}\ P \wedge t1 \notin \mathcal{D}\ P \wedge t2 \in min\text{-}elems(\mathcal{D}\ S)\}$
⟨*proof*⟩

lemma *mono-Seq-D13* :
assumes *ordered*: $P \sqsubseteq Q$
shows $min\text{-}elems\ (\mathcal{D}\ (P\ ;\ S)) \subseteq \mathcal{T}\ (Q\ ;\ S)$
⟨*proof*⟩

lemma *mono-Seq*[*simp*] : $P \sqsubseteq Q \Longrightarrow (P\ ;\ S) \sqsubseteq (Q\ ;\ S)$
⟨*proof*⟩

lemma *mono-Seq-D21*:
$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (S\ ;\ Q) \subseteq \mathcal{D}\ (S\ ;\ P)$
⟨*proof*⟩

lemma *mono-Seq-D22*:
assumes *ordered*: $P \sqsubseteq Q$
shows $(\forall\ s.\ s \notin \mathcal{D}\ (S\ ;\ P) \longrightarrow Ra\ (S\ ;\ P)\ s = Ra\ (S\ ;\ Q)\ s)$
⟨*proof*⟩

lemma *mono-Seq-D23* :
assumes *ordered*: $P \sqsubseteq Q$
shows *min-elems* $(\mathcal{D}\ (S\ ;\ P)) \subseteq \mathcal{T}\ (S\ ;\ Q)$
⟨*proof*⟩

lemma *mono-Seq-sym*[*simp*] : $P \sqsubseteq Q \Longrightarrow (S\ ;\ P) \sqsubseteq (S\ ;\ Q)$
⟨*proof*⟩

lemma *chain-Seq1*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i\ ;\ S)$
⟨*proof*⟩

lemma *chain-Seq2*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ S\ ;\ Y\ i)$
⟨*proof*⟩

lemma *limproc-Seq-D1*: *chain* $Y \Longrightarrow \mathcal{D}\ (\textit{lim-proc}\ (\textit{range}\ Y)\ ;\ S) = \mathcal{D}\ (\textit{lim-proc}\ (\textit{range}\ (\lambda i.\ Y\ i\ ;\ S)))$
⟨*proof*⟩

lemma *limproc-Seq-F1*: *chain* $Y \Longrightarrow \mathcal{F}\ (\textit{lim-proc}\ (\textit{range}\ Y)\ ;\ S) = \mathcal{F}\ (\textit{lim-proc}\ (\textit{range}\ (\lambda i.\ Y\ i\ ;\ S)))$
⟨*proof*⟩

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

lemma *limproc-Seq-D2*: *chain* $Y \Longrightarrow \mathcal{D}\ (S\ ;\ \textit{lim-proc}\ (\textit{range}\ Y)) = \mathcal{D}\ (\textit{lim-proc}\ (\textit{range}\ (\lambda i.\ S\ ;\ Y\ i\ )))$
⟨*proof*⟩

lemma *limproc-Seq-F2*:
chain $Y \Longrightarrow \mathcal{F}\ (S\ ;\ \textit{lim-proc}\ (\textit{range}\ Y)) = \mathcal{F}\ (\textit{lim-proc}\ (\textit{range}\ (\lambda i.\ S\ ;\ Y\ i\ )))$
⟨*proof*⟩

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

lemma *Seq-cont*[*simp*]:
assumes f:*cont* f
and g:*cont* g
shows *cont* $(\lambda x.\ f\ x\ ;\ g\ x)$
⟨*proof*⟩

end

3.7 The Renaming Operator

theory *Renaming*
imports *Process*
begin

3.7.1 Some preliminaries

definition *EvExt* **where** ‹*EvExt f x ≡ case-event (ev o f) tick x*›

term ‹*f −‘ B*›

definition *finitary* :: ‹(*'a ⇒ 'b*) *⇒ bool*›
where ‹*finitary f ≡ ∀ x. finite*(*f −‘ {x}*)›

We start with some simple results.

lemma ‹*f −‘ {} = {}*› ⟨*proof*⟩

lemma ‹*X ⊆ Y ⟹ f −‘ X ⊆ f −‘ Y*› ⟨*proof*⟩

lemma ‹*f −‘*(*X ∪ Y*) *= f −‘ X ∪ f −‘ Y*› ⟨*proof*⟩

lemma *EvExt-id*: ‹*EvExt id = id*›
⟨*proof*⟩

lemma *EvExt-eq-tick*: ‹*EvExt f a = tick ⟷ a = tick*›
⟨*proof*⟩

lemma *tick-eq-EvExt*: ‹*tick = EvExt f a ⟷ a = tick*›
⟨*proof*⟩

lemma *EvExt-ev1*: ‹*EvExt f b = ev a ⟷* (∃ *c. b = ev c ∧ EvExt f* (*ev c*) *= ev a*)›
⟨*proof*⟩

lemma *EvExt-tF*: ‹*tickFree* (*map* (*EvExt f*) *s*) *⟷ tickFree s*›
⟨*proof*⟩

lemma *inj-EvExt*: ‹*inj EvExt*›
⟨*proof*⟩

lemma *EvExt-ftF*: ‹*front-tickFree* (*map* (*EvExt f*) *s*) *⟷ front-tickFree s*›
⟨*proof*⟩

lemma *map-EvExt-tick*: ‹[*tick*] *= map* (*EvExt f*) *t ⟷ t = *[*tick*]›

⟨*proof*⟩

lemma *anteced-EvExt-diff-tick*: ‹*EvExt f −‘ (X − {tick}) = EvExt f −‘ X − {tick}›
⟨*proof*⟩

lemma *ev-elem-anteced1*: ‹*ev a ∈ EvExt f −‘ A ⟷ ev (f a) ∈ A*›
and *tick-elem-anteced1*: ‹*tick ∈ EvExt f −‘ A ⟷ tick ∈ A*›
⟨*proof*⟩

lemma *hd-map-EvExt*: ‹*t ≠ [] ⟹ hd t = ev a ⟹ hd (map (EvExt f) t) = ev (f a)*›
and *tl-map-EvExt*: ‹*t ≠ [] ⟹ tl (map (EvExt f) t) = map (EvExt f) (tl t)*›
⟨*proof*⟩

3.7.2 The Renaming Operator Definition

lift-definition *Renaming* :: ‹['*a process*, '*a* ⇒ '*b*] ⇒ '*b process*›
is ‹*λP f. ({(s, R). ∃ s1. (s1, (EvExt f) −‘ R) ∈ ℱ P ∧ s = map (EvExt f) s1} ∪*
{(s ,R). ∃ s1 s2. tickFree s1 ∧ front-tickFree s2 ∧ s = (map (EvExt f) s1) @ s2 ∧ s1 ∈ 𝒟 P},
{t. ∃ s1 s2. tickFree s1 ∧ front-tickFree s2 ∧ t = (map (EvExt f) s1) @ s2 ∧ s1 ∈ 𝒟 P})›
⟨*proof*⟩

Some syntaxic sugar

syntax
-Renaming :: ‹'*a process* ⇒ *updbinds* ⇒ '*a process*› (‹-⟦-⟧› [*100*, *100*])
translations
-Renaming P updates ⇌ *CONST Renaming P* (*-Update* (*CONST id*) *updates*)

Now we can write $P⟦a := b⟧$. But like in *HOL.Fun*, we can write this kind of expression with as many updates we want: $P⟦a := b,\ c := d,\ e := f,\ g := h⟧$. By construction we also inherit all the results about *fun-upd*, for example *?z ≠ ?x ⟹ (?f(?x := ?y)) ?z = ?f ?z*

?f(?x := ?y, ?x := ?z) = ?f(?x := ?z)

?a ≠ ?c ⟹ ?m(?a := ?b, ?c := ?d) = ?m(?c := ?d, ?a := ?b).

lemma ‹*P⟦x := y, x := z⟧ = P⟦x := z⟧*› ⟨*proof*⟩

lemma ‹*a ≠ c ⟹ P⟦a := b, c := d⟧ = P⟦c := d, a := b⟧*› ⟨*proof*⟩

3.7.3 The Projections

lemma *F-Renaming*: ‹*ℱ (Renaming P f) =*

$\{(s, R).\ \exists\, s1.\ (s1, EvExt\ f\ -`\ R) \in \mathcal{F}\ P \wedge s = map\ (EvExt\ f)\ s1\} \cup$
$\{(s, R).\ \exists\, s1\ s2.\ tickFree\ s1 \wedge front\text{-}tickFree\ s2 \wedge s = map\ (EvExt\ f)\ s1\ @\ s2 \wedge s1 \in \mathcal{D}\ P\}\rangle$
⟨*proof*⟩

lemma *D-Renaming*: ‹$\mathcal{D}\ (Renaming\ P\ f) = \{t.\ \exists\, s1\ s2.\ tickFree\ s1 \wedge front\text{-}tickFree\ s2 \wedge t = map\ (EvExt\ f)\ s1\ @\ s2 \wedge s1 \in \mathcal{D}\ P\}$›
⟨*proof*⟩

lemma *T-Renaming*: ‹$\mathcal{T}\ (Renaming\ P\ f) =$
$\{t.\ \exists\ t1.\ t1 \in \mathcal{T}\ P \wedge t = map\ (EvExt\ f)\ t1\} \cup$
$\{t.\exists\ t1\ t2.\ tickFree\ t1 \wedge front\text{-}tickFree\ t2 \wedge t = map\ (EvExt\ f)\ t1\ @\ t2 \wedge t1 \in \mathcal{D}\ P\}\rangle$
⟨*proof*⟩

3.7.4 Continuity Rule

Monotonicity of *Renaming*.

lemma *mono-Renaming*[*simp*] : ‹$P \sqsubseteq Q \Longrightarrow (Renaming\ P\ f) \sqsubseteq (Renaming\ Q\ f)\rangle$
⟨*proof*⟩

lemma ‹$\{ev\ c\ |c.\ f\ c = a\} = ev`\{c\ .\ f\ c = a\}\rangle$ ⟨*proof*⟩

lemma *vimage-EvExt-subset-vimage*: ‹$EvExt\ f\ -`\ (ev\ `\ A) \subseteq insert\ tick\ (ev\ `\ (f\ -`\ A))\rangle$
and *vimage-subset-vimage-EvExt*: ‹$(ev\ `\ (f\ -`\ A)) \subseteq (EvExt\ f\ -`\ (ev\ `\ A)) - \{tick\}\rangle$
⟨*proof*⟩

Some useful results about *finitary*, and preliminaries lemmas for continuity.

lemma *inj-imp-finitary*[*simp*] : ‹$inj\ f \Longrightarrow finitary\ f\rangle$ ⟨*proof*⟩

lemma *finitary-comp-iff* : ‹$finitary\ g \Longrightarrow finitary\ (g\ o\ f) \longleftrightarrow finitary\ f\rangle$
⟨*proof*⟩

lemma *finitary-updated-iff*[*simp*] : ‹$finitary\ (f\ (a := b)) \longleftrightarrow finitary\ f\rangle$
⟨*proof*⟩

By declaring this simp, we automatically obtain this kind of result.

lemma ‹$finitary\ f \longleftrightarrow finitary\ (f\ (a := b,\ c := d,\ e := g,\ h := i))\rangle$ ⟨*proof*⟩

lemma *Cont-RenH2*: ‹*finitary* (*EvExt f*) ⟷ *finitary f*›
⟨*proof*⟩

lemma *Cont-RenH3*:
‹{*s1* @ *t1* |*s1 t1*. (∃ *b*. *s1* = [*b*] & *f b* = *a*) ∧ *list* = *map f t1*} =
(⋃ *b* ∈ *f* −‘{*a*}. (λ*t*. *b* # *t*) ‘ {*t*. *list* = *map f t*})›
⟨*proof*⟩

lemma *Cont-RenH4*: ‹*finitary f* ⟷ (∀ *s*. *finite* {*t*. *s* = *map f t*})›
⟨*proof*⟩

lemma *Cont-RenH5*: ‹*finitary f* ⟹ *finite* (⋃ *t* ∈ {*t*. *t* ≤ *s*}. {*s*. *t*=*map* (*EvExt f*) *s*})›
⟨*proof*⟩

lemma *Cont-RenH6*: ‹{*t*. ∃ *t2*. *tickFree t* ∧ *front-tickFree t2* ∧ *x* = *map* (*EvExt f*) *t* @ *t2*}
⊆ {*y*. ∃ *xa*. *xa* ∈ {*t*. *t* <= *x*} ∧ *y* ∈ {*s*. *xa* = *map* (*EvExt f*) *s*}}›
⟨*proof*⟩

lemma *Cont-RenH7*: ‹*finite* {*t*. ∃ *t2*. *tickFree t* ∧ *front-tickFree t2* ∧ *s* = *map* (*EvExt f*) *t* @ *t2*}›
if ‹*finitary f*›
⟨*proof*⟩

lemma *chain-Renaming*: ‹*chain Y* ⟹ *chain* (λ*i*. *Renaming* (*Y i*) *f*)›
⟨*proof*⟩

A key lemma for the continuity.

lemma *Inter-nonempty-finite-chained-sets*:
‹∀ *i*. *S i* ≠ {} ⟹ *finite* (*S 0*) ⟹ ∀ *i*. *S* (*Suc i*) ⊆ *S i* ⟹ (⋂ *i*. *S i*) ≠ {}›
⟨*proof*⟩

Finally, continuity !

lemma *Cont-Renaming-prem*:

‹(⨆ *i*. *Renaming* (*Y i*) *f*) = (*Renaming* (⨆ *i*. *Y i*) *f*)› **if** *finitary*: ‹*finitary f*›
and *chain*: ‹*chain Y*›
⟨*proof*⟩

lemma *Renaming-cont*[*simp*] : ‹*cont P* ⟹ *finitary f* ⟹ *cont* (λ*x*. (*Renaming* (*P x*) *f*))›
⟨*proof*⟩

lemma *RenamingF-cont* : ‹*cont P* ⟹ *cont* (λ*x*. (*P x*)⟦*a* := *b*⟧)›

⟨*proof*⟩

lemma ‹*cont P* ⟹ *cont* (λ*x*. (*P x*)⟦*a* := *b*, *c* := *d*, *e* := *f*, *g* := *a*⟧)›

⟨*proof*⟩

3.7.5 Some nice properties

lemma *EvExt-comp*: ‹*EvExt* (*g* ∘ *f*) = *EvExt g* ∘ *EvExt f*›
⟨*proof*⟩

lemma *Renaming-comp* : ‹*Renaming P* (*g o f*) = *Renaming* (*Renaming P f*) *g*›
⟨*proof*⟩

lemma *Renaming-id*: ‹*Renaming P id* = *P*›
⟨*proof*⟩

lemma *Renaming-inv*: ‹*Renaming* (*Renaming P f*) (*inv f*) = *P*› **if** ‹*inj f*›
⟨*proof*⟩

end

Chapter 4

Concurrent CSP Operators

4.1 The Hiding Operator

theory *Hiding*
imports *Process*
begin

4.1.1 Preliminaries : primitives and lemmas

abbreviation *trace-hide t A* ≡ *filter* (λ*x*. *x* ∉ *A*) *t*

lemma *Hiding-tickFree* : *tickFree s* ⟷ *tickFree* (*trace-hide s* (*ev*‘*A*))
⟨*proof*⟩

lemma *Hiding-fronttickFree* : *front-tickFree s* ⟹ *front-tickFree* (*trace-hide s* (*ev*‘*A*))
⟨*proof*⟩

lemma *trace-hide-union*[*simp*] : *trace-hide t* (*ev* ‘ (*A* ∪ *B*)) = *trace-hide* (*trace-hide t* (*ev* ‘ *A*)) (*ev* ‘ *B*)
⟨*proof*⟩

abbreviation *isInfHiddenRun f P A* ≡ *strict-mono f* ∧ (∀ *i*. *f i* ∈ *T P*) ∧
(∀ *i*. *trace-hide* (*f i*) (*ev* ‘ *A*) = *trace-hide* (*f* (*0*::*nat*)) (*ev* ‘ *A*))

lemma *isInfHiddenRun-1*: *isInfHiddenRun f P A* ⟷ *strict-mono f* ∧ (∀ *i*. *f i* ∈ *T P*) ∧
(∀ *i*. ∃ *t*. *f i* = *f 0* @ *t* ∧ *set t* ⊆ (*ev* ‘ *A*))
(**is** *?A* ⟷ *?B*)
⟨*proof*⟩

abbreviation *div-hide P A* ≡ {*s*. ∃ *t u*. *front-tickFree u* ∧
tickFree t ∧ *s* = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧
(*t* ∈ *D P* ∨ (∃ *f*. *isInfHiddenRun f P A* ∧ *t* ∈

range f))}

lemma *inf-hidden*:
assumes *as1*:$\forall\, t.$ *trace-hide t (ev ‘ A) = trace-hide s (ev ‘ A)* $\longrightarrow$ *(t, ev ‘ A)* $\notin$ $\mathcal{F}$ *P*
and *as2*:$s \in \mathcal{T}$ *P*
shows $\exists\, f.$ *isInfHiddenRun f P A* $\wedge$ $s \in$ *range f*
⟨*proof*⟩

lemma *trace-hide-append*: *s @ t = trace-hide ta (ev ‘ A)* $\Longrightarrow$ $\exists$ *ss tt. ta = ss@tt* $\wedge$
s = trace-hide ss (ev ‘ A) $\wedge$
t = trace-hide tt (ev ‘ A)
⟨*proof*⟩

4.1.2 The Hiding Operator Definition

lift-definition *Hiding* :: [$'\alpha$ *process* ,$'\alpha$ *set*] => $'\alpha$ *process* (‹- \ -› [57,56] 57)
is
‹λ*P A.* ({(*s*,*X*). $\exists$ *t. s = trace-hide t (ev‘A)* $\wedge$ (*t*,*X* $\cup$ (*ev‘A*)) $\in$ $\mathcal{F}$ *P*} $\cup$
{(*s*,*X*). *s* $\in$ *div-hide P A*}, *div-hide P A*)›
⟨*proof*⟩

4.1.3 Projections

lemma *F-Hiding*: ‹$\mathcal{F}$ *(P \ A) =* {(*s*,*X*). $\exists$ *t. s = trace-hide t (ev‘A)* $\wedge$ (*t*,*X* $\cup$ (*ev‘A*)) $\in$ $\mathcal{F}$ *P*} $\cup$
{(*s*,*X*). *s* $\in$ *div-hide P A*}›
⟨*proof*⟩

lemma *D-Hiding*: ‹$\mathcal{D}$ *(P \ A) = div-hide P A*›
⟨*proof*⟩

lemma *T-Hiding*: ‹$\mathcal{T}$ *(P \ A) =* {*s*. $\exists$ *t. s = trace-hide t (ev‘A)* $\wedge$ *(t, ev‘A)* $\in$ $\mathcal{F}$ *P*} $\cup$ *div-hide P A*›
⟨*proof*⟩

4.1.4 Continuity Rule

lemma *mono-Hiding*[*simp*]: ‹(*P*::$'a$ *process*) $\sqsubseteq$ *Q* $\Longrightarrow$ *(P \ A)* $\sqsubseteq$ *(Q \ A)*›
⟨*proof*⟩

lemma *cont-Hiding1* :‹*chain Y* $\Longrightarrow$ *chain* (λ *i. (Y i \ A)*)›
⟨*proof*⟩

lemma *KoenigLemma*:
assumes $*$:*infinite* (*Tr*::$'a$ *list set*) **and** $**$:$\forall$ *i. finite*{*t*. $\exists\, t' \in$ *Tr. t = take i t'*}
shows $\exists$ (*f*::*nat* $\Rightarrow$ $'a$ *list*). *strict-mono f* $\wedge$ *range f* $\subseteq$ {*t*. $\exists\, t' \in$ *Tr. t* $\leq$ *t'*}

⟨*proof*⟩

lemma *div-Hiding-lub* :
‹*finite* (*A*::'*a set*) ⟹ *chain Y* ⟹ $\mathcal{D}$ (⨆ *i*. (*Y i* \ *A*)) ⊆ $\mathcal{D}$ ((⨆ *i*. *Y i*) \ *A*)›
⟨*proof*⟩

lemma *cont-Hiding2* : ‹*finite A* ⟹ *chain Y* ⟹ ((⨆ *i*. *Y i*) \ *A*) = (⨆ *i*. (*Y i* \ *A*))›
⟨*proof*⟩

lemma *cont-Hiding-base*[*simp*] : ‹*finite A* ⟹ *cont* (λ*x*. *x* \ *A*)›
⟨*proof*⟩

lemma *Hiding-cont*[*simp*]: ‹*finite A* ⟹ *cont f* ⟹ *cont* (λ*x*. *f x* \ *A*)›
⟨*proof*⟩

end

4.2 The Synchronizing Operator

theory *Sync*
imports *Process HOL−Library.Infinite-Set*
begin

4.2.1 Basic Concepts

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 $\{z.\exists\ u.\ z = (x\ \#\ u) \wedge u \in$ *setinterleaving* $(s,\ X,\ (y\ \#\ t))\}$)

fun *setinterleavingList*::$'a$ *trace* $\times$ ($'a$ *event*) *set* $\times$ $'a$ *trace* $\Rightarrow$ ($'a$ *trace*)*list*
where

si-empty1l: *setinterleavingList*([], X, []) = [[]]
| *si-empty2l*: *setinterleavingList*([], X, $(y\ \#\ t)$) =
(*if* $(y \in X)$
then []
else map $(\lambda z.\ y\#z)$ (*setinterleavingList* ([], X, t)))
| *si-empty3l*: *setinterleavingList*($(x\ \#\ s)$, X, []) =
(*if* $(x \in X)$
then []
else map $(\lambda z.\ x\#z)$ (*setinterleavingList* $(s,\ X,\ [])$))

| *si-neql* : *setinterleavingList*($(x\ \#\ s)$, X, $(y\ \#\ t)$) =
(*if* $(x \in X)$
then if $(y \in X)$
then if $(x = y)$
then map $(\lambda z.\ x\#z)$ (*setinterleavingList*(s, X, t))
else []
else map $(\lambda z.\ y\#z)$ (*setinterleavingList* ($(x\#s)$, X, t))
else if $(y \notin X)$
then map $(\lambda z.\ x\#z)$ (*setinterleavingList* $(s,\ X,\ (y\ \#\ t))$)
@ *map* $(\lambda z.\ y\#z)$ (*setinterleavingList* ($(x\#s)$, X, t))
else map $(\lambda z.\ x\#z)$ (*setinterleavingList* $(s,\ X,\ (y\ \#\ t))$))

lemma *finiteSetinterleavingList*: *finite* (*set* (*setinterleavingList*(s, X, t)))
⟨*proof*⟩

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

abbreviation *setinterleaves-syntax* (- *setinterleaves* $'(()'(-,\ -')(),\ -')$ [60,0,0,0] 70)
where u *setinterleaves* $((s,\ t),\ X)$ == $(u \in$ *setinterleaving*(s, X, t))

4.2.2 Consequences

lemma *emptyLeftProperty*:$\forall\ s.\ s$ *setinterleaves* $(([],\ t),\ A) \longrightarrow s{=}t$
⟨*proof*⟩

lemma *emptyLeftSelf*: $(\forall\ t1.\ t1 \in$ *set* $t \longrightarrow t1 \notin A) \longrightarrow t$ *setinterleaves* $(([],\ t),\ A)$
⟨*proof*⟩

lemma *emptyLeftNonSync*: *s setinterleaves* ((*[]*, *t*), *A*) $\Longrightarrow \forall a.\ a \in set\ t \longrightarrow a \notin A$
⟨*proof*⟩

lemma *ftf-Sync1*: $a \notin set(u) \wedge a \notin set(t) \wedge$ *s setinterleaves* ((*t*, *u*), *A*) $\longrightarrow a \notin set(s)$
⟨*proof*⟩

lemma *addNonSyncEmpty*: *sa setinterleaves* ((*[]*, *u*), *A*) $\wedge$ *y1* $\notin$ *A* $\longrightarrow$
(*sa* @ [*y1*]) *setinterleaves* ((*[y1]*, *u*), *A*) $\wedge$ (*sa* @ [*y1*]) *setinterleaves* ((*[]*, *u* @ [*y1*]), *A*)
⟨*proof*⟩

lemma *addNonSync*: *sa setinterleaves* ((*t*,*u*),*A*)$\wedge$ *y1* $\notin$ *A*$\longrightarrow$
(*sa*@[*y1*]) *setinterleaves* ((*t*@[*y1*],*u*),*A*)$\wedge$(*sa*@[*y1*]) *setinterleaves* ((*t*,*u*@[*y1*]),*A*)
⟨*proof*⟩

lemma *addSyncEmpty*: *sa setinterleaves* ((*[]*, *u*), *A*) $\wedge$ *y1* $\in$ *A* $\longrightarrow$
(*sa* @ [*y1*]) *setinterleaves* ((*[y1]*, *u* @ [*y1*]), *A*)
⟨*proof*⟩

lemma *addSync*: *sa setinterleaves* ((*t*,*u*),*A*)$\wedge$ *y1* $\in$ *A*$\longrightarrow$(*sa*@[*y1*]) *setinterleaves* ((*t*@[*y1*],*u*@[*y1*]),*A*)
find-theorems ‹(@)› *name*: *cases*
⟨*proof*⟩

lemma *doubleReverse*: *s1 setinterleaves* ((*t*, *u*), *A*)$\longrightarrow$(*rev s1*) *setinterleaves* ((*rev t*, *rev u*), *A*)
⟨*proof*⟩

lemma *ftf-Sync21*: (*a*$\in$*set*(*u*)$\wedge$*a*$\notin$*set*(*t*)$\vee$*a*$\in$*set*(*t*)$\wedge$*a*$\notin$*set*(*u*))$\wedge$*a*$\in$*A*$\longrightarrow$*setinterleaving*(*u*, *A* ,*t*)= {}
⟨*proof*⟩

lemma *ftf-Sync32*: *u*=*u1*@[*tick*]$\wedge$*t*=*t1*@[*tick*]$\wedge$*s setinterleaves* ((*t*, *u*), *insert tick* (*ev* ‘ *A*))$\Longrightarrow$
$\exists$ *s1*. *s1 setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *A*))$\wedge$ (*s*=*s1*@[*tick*])
⟨*proof*⟩

lemma *SyncWithTick-imp-NTF*:
(*s* @ [*tick*]) *setinterleaves* ((*t*, *u*),*insert tick* (*ev* ‘ *A*))
$\Longrightarrow$ *t*$\in$ $\mathcal{T}$ *P* $\Longrightarrow$ *u* $\in$ $\mathcal{T}$ *Q*
$\Longrightarrow$ $\exists$ *t1 u1*. *t*=*t1*@[*tick*] $\wedge$ *u*=*u1*@[*tick*]$\wedge$ *s setinterleaves* ((*t1*, *u1*), *insert tick* (*ev* ‘ *A*))

⟨*proof*⟩

lemma *synPrefix1*:
$ta = []$
$\Longrightarrow \exists\, t1\ u1.\ (s\ @\ t)\ setinterleaves\ ((ta,\ u),\ A) \longrightarrow$
$t1 \leq ta \wedge u1 \leq u \wedge s\ setinterleaves\ ((t1,\ u1),\ A)$
⟨*proof*⟩

lemma *synPrefix*: $\exists\, t1\ u1.\ (s\ @\ t)\ setinterleaves\ ((ta,\ u),\ A) \longrightarrow$
$t1 \leq ta \wedge u1 \leq u \wedge s\ setinterleaves\ ((t1,\ u1),\ A)$
⟨*proof*⟩

lemma *SyncWithTick-imp-NTF1*:
$(s\ @\ [tick])\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A))$
$\Longrightarrow t \in \mathcal{T}\ P \Longrightarrow u \in \mathcal{T}\ Q$
$\Longrightarrow \exists\ t\ u\ Xa\ Y.\ (t,\ Xa) \in \mathcal{F}\ P \wedge (\ (u,\ Y) \in \mathcal{F}\ Q \wedge$
$s\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A)) \wedge$
$X - \{tick\} = (Xa \cup Y) \cap insert\ tick\ (ev\ `\ A) \cup Xa \cap Y)$
⟨*proof*⟩

lemma *ftf-Sync*:
front-tickFree u
$\Longrightarrow$ *front-tickFree t*
$\Longrightarrow s\ setinterleaves\ ((t,\ u), insert\ tick\ (ev\ `\ A))$
$\Longrightarrow$ *front-tickFree s*
⟨*proof*⟩

lemma *is-processT5-SYNC2*:
$\bigwedge sa\ Y\ t\ u\ Xa\ Ya.\ (t,Xa) \in \mathcal{F}\ P \wedge (u,Ya) \in \mathcal{F}\ Q \wedge (sa\ setinterleaves\ ((t,u), insert\ tick\ (ev\ `\ A)))$
$\Longrightarrow \forall\, c.\ c \in Y \longrightarrow$
$(\forall\ t1\ u1.\ (sa@[c])\ setinterleaves\ ((t1,u1), insert\ tick\ (ev\ `\ A)) \longrightarrow$
$(((t1,\{\}) \in \mathcal{F}\ P \longrightarrow (u1,\ \{\}) \notin \mathcal{F}\ Q) \wedge ((t1,\{\}) \in \mathcal{F}\ Q \longrightarrow (u1,\ \{\}) \notin \mathcal{F}\ P)))$
$\Longrightarrow \exists\, t2\ u2\ Xb.\ (t2,\ Xb) \in \mathcal{F}\ P \wedge$
$(\exists\, Yb.\ (u2,\ Yb) \in \mathcal{F}\ Q \wedge$
$sa\ setinterleaves\ ((t2,\ u2),\ insert\ tick\ (ev\ `\ A)) \wedge$
$(\ Xa \cup Ya) \cap insert\ tick\ (ev\ `\ A) \cup Xa \cap Ya \cup Y =$
$(Xb \cup Yb) \cap insert\ tick\ (ev\ `\ A) \cup Xb \cap Yb)$
⟨*proof*⟩

lemma *pt3*:
$ta \in \mathcal{T}\ P$
$\Longrightarrow u \in \mathcal{T}\ Q \Longrightarrow (s\ @\ t)\ setinterleaves\ ((ta,\ u),\ insert\ tick\ (ev\ `\ A))$
$\Longrightarrow \exists\, t\ u\ X.\ (t,\ X) \in \mathcal{F}\ P \wedge$
$(\exists\, Y.\ (u,\ Y) \in \mathcal{F}\ Q \wedge$
$s\ setinterleaves\ ((t,\ u),\ insert\ tick\ (ev\ `\ A)) \wedge$
$tick \notin X \wedge tick \notin Y \wedge (X \cup Y) \cap ev\ `\ A = \{\} \wedge X \cap Y = \{\})$
⟨*proof*⟩

4.2.3 The Sync Operator Definition

lift-definition *Sync* :: [*'a process*,*'a set*,*'a process*] => *'a process* ((*3*(- ⟦-⟧/ -)) [*64*,*0*,*65*] *64*)
is $\lambda P\ A\ Q.$ ({(*s*,*R*).$\exists\ t\ u\ X\ Y.\ (t,X) \in \mathcal{F}\ P \wedge (u,Y) \in \mathcal{F}\ Q \wedge$
$(s\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})) \wedge$
$R = (X \cup Y) \cap ((ev`A) \cup \{tick\}) \cup X \cap Y\}$ ∪
{(*s*,*R*).$\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v{=}[]) \wedge$
$s = r@v \wedge$
$(r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$,

{*s*. $\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v{=}[]) \wedge$
$s = r@v \wedge$
$(r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\})$
⟨*proof*⟩

4.2.4 The Projections

lemma
F-Sync : $\mathcal{F}\ (P\ [\![\ A\ ]\!]\ Q) =$
{(*s*,*R*).$\exists\ t\ u\ X\ Y.\ (t,X) \in \mathcal{F}\ P \wedge$
$(u,Y) \in \mathcal{F}\ Q \wedge$
$s\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\}) \wedge$
$R = (X \cup Y) \cap ((ev`A) \cup \{tick\}) \cup X \cap Y\} \cup$
{(*s*,*R*).$\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge$
$(tickFree\ r \vee v{=}[]) \wedge$
$s = r@v \wedge$
$r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\}) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$
⟨*proof*⟩

lemma
D-Sync : $\mathcal{D}\ (P\ [\![\ A\ ]\!]\ Q) =$
{*s*. $\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v{=}[]) \wedge$
$s = r@v \wedge r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\}) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$
⟨*proof*⟩

lemma
T-Sync : $\mathcal{T}\ (P\ [\![\ A\ ]\!]\ Q) =$
{*s*. $\exists\ t\ u.\ t \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q \wedge$
$s\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\})\} \cup$
{*s*. $\exists\ t\ u\ r\ v.\ front\text{-}tickFree\ v \wedge (tickFree\ r \vee v{=}[]) \wedge$
$s = r@v \wedge r\ setinterleaves\ ((t,u),(ev`A) \cup \{tick\}) \wedge$
$(t \in \mathcal{D}\ P \wedge u \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge u \in \mathcal{T}\ P)\}$
⟨*proof*⟩

4.2.5 Syntax for Interleave and Parallel Operator

abbreviation *Inter-syntax* ((-|||-) [72,73] 72)
where $P \;|||\; Q \equiv (P \;[\![\; \{\} \;]\!]\; Q)$

abbreviation *Par-syntax* ((-||-) [74,75] 74)
where $P \;||\; Q \;\equiv (P \;[\![\; UNIV \;]\!]\; Q)$

4.2.6 Continuity Rule

lemma *mono-Sync-D1*:
$P \sqsubseteq Q \Longrightarrow \mathcal{D}\ (Q \;[\![\; A \;]\!]\; S) \subseteq \mathcal{D}\ (P \;[\![\; A \;]\!]\; S)$
⟨*proof*⟩

lemma *mono-Sync-D2*:
$P \sqsubseteq Q \Longrightarrow (\forall\ s.\ s \notin \mathcal{D}\ (P \;[\![\; A \;]\!]\; S) \longrightarrow Ra\ (P \;[\![\; A \;]\!]\; S)\ s = Ra\ (Q \;[\![\; A \;]\!]\; S)\ s)$
⟨*proof*⟩

lemma *interPrefixEmpty*: *s setinterleaves* $((t,[]),A) \wedge t1<t \Longrightarrow \exists\, s1<s.$ *s1 setinterleaves* $((t1,[]),A)$
⟨*proof*⟩

lemma *interPrefix*: $\exists\, u1\ s1.$ *s setinterleaves* $((t,u),A) \wedge t1<t \longrightarrow u1 \leq u \wedge s1<s \wedge$ *s1 setinterleaves* $((t1,u1),A)$
⟨*proof*⟩

lemma *mono-Sync2a*:
$r \in$ *min-elems*$(\mathcal{D}\ (P\ [\![A]\!]\ S))$
$\Longrightarrow \exists\, t\ u.$ *r setinterleaves* $((t,u),$*insert tick* $(ev\ `\ A)) \wedge$
$(t \in$ *min-elems*$(\mathcal{D}\ P) \wedge u \in \mathcal{T}\ S \vee t \in$ *min-elems*$(\mathcal{D}\ S) \wedge$
$u \in \mathcal{T}\ P \wedge u \in (\mathcal{T}\ P - (\mathcal{D}\ P -$ *min-elems*$(\mathcal{D}\ P))))$
⟨*proof*⟩

lemma *mono-Sync-D3*:
assumes *ordered*: $P \sqsubseteq Q$
shows *min-elems* $(\mathcal{D}\ (P\ [\![A]\!]\ S)) \subseteq \mathcal{T}(Q\ [\![A]\!]\ S)$
⟨*proof*⟩

lemma *mono-D-Sync*: $\mathcal{D}\ (S \;[\![\; A \;]\!]\; Q) = \mathcal{D}\ (Q \;[\![\; A \;]\!]\; S)$
⟨*proof*⟩

lemma *mono-T-Sync*: $\mathcal{T}\ (S \;[\![\; A \;]\!]\; Q) = \mathcal{T}\ (Q \;[\![\; A \;]\!]\; S)$
⟨*proof*⟩

lemma *mono-F-Sync*: $\mathcal{F}\ (S \;[\![\; A \;]\!]\; Q) \;= \mathcal{F}\ (Q \;[\![\; A \;]\!]\; S)$
⟨*proof*⟩

lemma *mono-Ra-Sync*: $Ra\ (S \;[\![\; A \;]\!]\; Q)\ s = Ra\ (Q \;[\![\; A \;]\!]\; S)\ s$
⟨*proof*⟩

lemma *mono-Sync*[*simp*] : $P \sqsubseteq Q \Longrightarrow (P \llbracket A \rrbracket S) \sqsubseteq (Q \llbracket A \rrbracket S)$
⟨*proof*⟩

lemma *mono-Sync-sym* [*simp*]: $P \sqsubseteq Q \Longrightarrow (S \llbracket A \rrbracket P) \sqsubseteq (S \llbracket A \rrbracket Q)$
⟨*proof*⟩

lemma *chain-Sync1*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i \llbracket A \rrbracket S)$
⟨*proof*⟩

lemma *chain-Sync2*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ S \llbracket A \rrbracket Y\ i)$
⟨*proof*⟩

lemma *empty-setinterleaving* : $[]$ *setinterleaves* $((t, u), A) \Longrightarrow t = []$
⟨*proof*⟩

lemma *inters-fin-fund*: *finite*$\{(t, u).\ s$ *setinterleaves* $((t, u), A)\}$
⟨*proof*⟩

lemma *inters-fin*: *finite*$\{(t, u, r).\ r$ *setinterleaves* $((t, u),$ *insert tick* $(ev\ `\ A)) \wedge$
$(\exists v.\ x = r\ @\ v \wedge$ *front-tickFree* $v \wedge$ (*tickFree* $r \vee v = [])$)$\}$
(**is** *finite ?A*)
⟨*proof*⟩

lemma *limproc-Sync-D*:
chain $Y \Longrightarrow \mathcal{D}$ (*lim-proc*(*range* Y)$\llbracket A \rrbracket$ $S) = \mathcal{D}$(*lim-proc*(*range*$(\lambda i.\ Y\ i \llbracket A \rrbracket S)$))
⟨*proof*⟩

lemma *limproc-Sync-F-annex1*:
chain Y
$\Longrightarrow \forall n.\ (a, b) \in \mathcal{F}\ (Y\ n \llbracket A \rrbracket S)$
$\Longrightarrow \exists x.\ a \notin \mathcal{D}\ (Y\ x \llbracket A \rrbracket S)$
$\Longrightarrow \exists t\ u\ X.\ (\forall x.\ (t, X) \in \mathcal{F}\ (Y\ x)) \wedge$
$(\exists Y.\ (u, Y) \in \mathcal{F}\ S \wedge a$ *setinterleaves*$((t, u),$*insert tick* $(ev\ `\ A)) \wedge$
$b = (X \cup Y) \cap$ *insert tick* $(ev\ `\ A) \cup X \cap Y)$
⟨*proof*⟩

lemma *limproc-Sync-F-annex2*:
chain Y
$\Longrightarrow \forall t\ u\ r.\ r$ *setinterleaves* $((t, u),$ *insert tick* $(ev\ `\ A)) \longrightarrow$
$(\forall v.\ a = r\ @\ v \longrightarrow$ *front-tickFree* $v \longrightarrow$
$\neg$ *tickFree* $r \wedge v \neq [] \vee ((\exists x.\ t \notin \mathcal{D}\ (Y\ x)) \vee u \notin \mathcal{T}\ S) \wedge$
$(t \in \mathcal{D}\ S \longrightarrow (\exists x.\ u \notin \mathcal{T}\ (Y\ x))))$
$\Longrightarrow \exists x.\ a \notin \mathcal{D}(Y\ x \llbracket A \rrbracket S)$
⟨*proof*⟩

lemma *limproc-Sync-F*:
chain Y $\Longrightarrow \mathcal{F}$(*lim-proc* (*range Y*)⟦ *A* ⟧*S*) = $\mathcal{F}$(*lim-proc* (*range* (λ*i*. *Y i* ⟦ *A* ⟧ *S*)))
⟨*proof*⟩

lemma *cont-left-Sync* : *chain Y* $\Longrightarrow$ ((⨆ *i*. *Y i*) ⟦ *A* ⟧ *S*) = (⨆ *i*. (*Y i* ⟦ *A* ⟧ *S*))
⟨*proof*⟩

lemma *cont-right-Sync* : *chain Y* $\Longrightarrow$ (*S* ⟦ *A* ⟧ (⨆ *i*. *Y i*)) = (⨆ *i*. (*S* ⟦ *A* ⟧ *Y i*))
⟨*proof*⟩

lemma *Sync-cont*[*simp*]:
assumes *f*:*cont f*
and *g*:*cont g*
shows *cont* (λ*x*. (*f x*) ⟦*A*⟧ (*g x*))
⟨*proof*⟩

end

4.3 The Multi-Prefix Operator Definition

theory *Mprefix*
imports *Process*
begin

4.3.1 The Definition and some Consequences

lift-definition *Mprefix* :: ['*a set*,'*a* => '*a process*] => '*a process*
is λ*A P*. ({(*tr*,*ref*). *tr* = [] ∧ *ref Int* (*ev* ‘ *A*) = {}} ∪
{(*tr*,*ref*). *tr* ≠ [] ∧ *hd tr* ∈ (*ev* ‘ *A*) ∧
(∃ *a*. *ev a* = (*hd tr*)∧(*tl tr*,*ref*)∈$\mathcal{F}$(*P a*))},
{*d*. *d* ≠ [] ∧ *hd d* ∈ (*ev* ‘ *A*) ∧ (∃ *a*. *ev a* = *hd d* ∧ *tl d* ∈ $\mathcal{D}$ (*P a*))})
⟨*proof*⟩

syntax
-*Mprefix* :: [*pttrn*,'*a set*,'*a process*] ⇒ '*a process* ((3□(-/∈-)/ → (-))[0,0,64]64)

term *Ball A* (λ*x*. *P*)

translations
□*x*∈*A* → *P* ⇌ *CONST Mprefix A* (λ*x*. *P*)

Syntax Check:

term ‹□*x*∈*A* → □*y*∈*A* → □*z*∈*A* → *P z x y* = *Q*›

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

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

4.3.2 Projections in Prefix

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

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

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

4.3.3 Basic Properties

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

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

4.3.4 Proof of Continuity Rule

Backpatch Isabelle 2009-1

definition
contlub :: (*'a*::*cpo* ⇒ *'b*::*cpo*) ⇒ *bool*
where
contlub f = (∀ *Y*. *chain Y* ⟶ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*)))

lemma *contlubE*:
⟦*contlub f*; *chain Y*⟧ ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))
⟨*proof*⟩

lemma *monocontlub2cont*: ⟦*monofun f*; *contlub f*⟧ ⟹ *cont f*
⟨*proof*⟩

lemma *contlubI*:
(⋀*Y*. *chain Y* ⟹ *f* (⨆ *i*. *Y i*) = (⨆ *i*. *f* (*Y i*))) ⟹ *contlub f*
⟨*proof*⟩

lemma *cont2contlub*: *cont f* ⟹ *contlub f*
⟨*proof*⟩

Core of Proof

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

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

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

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

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

lemma *proc-ord2-set*:
$P \sqsubseteq Q \Longrightarrow \{(s, X).\ s \notin \mathcal{D}\ P \wedge (s, X) \in \mathcal{F}\ P\} = \{(s, X).\ s \notin \mathcal{D}\ P \wedge (s, X) \in \mathcal{F}\ Q\}$
⟨*proof*⟩

lemma *proc-ord-proc-eq-spec*: $P \sqsubseteq Q \Longrightarrow \mathcal{D}\ P \subseteq \mathcal{D}\ Q \Longrightarrow P = Q$
⟨*proof*⟩

lemma *Mprefix-chainpreserving*: *chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Mprefix\ A\ (Y\ i))$
⟨*proof*⟩

lemma *limproc-is-thelub-fun*:
assumes *chain S*
shows $(Lub\ S\ c) = lim\text{-}proc\ (range\ (\lambda x.\ (S\ x\ c)))$
⟨*proof*⟩

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

lemma *Mprefix-cont* [*simp*]:
$(\bigwedge x.\ cont\ (f\ x)) \Longrightarrow cont\ (\lambda y.\ \Box\ z \in\ A \rightarrow f\ z\ y)$
⟨*proof*⟩

4.3.5 High-level Syntax for Read and Write

The following syntax introduces the usual channel notation for CSP. Slightly deviating from conventional wisdom, we view a channel not as a tag in a pair, rather than as a function of type $'a \Rightarrow 'b$. This paves the way for *typed* channels over a common universe of events.

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

syntax
-read :: [*id*, *pttrn*, *'a process*] => *'a process*
((*3*(-**?**-) /→ -) [*0,0,78*] *78*)
-readX :: [*id*, *pttrn*, *bool*,*'a process*] => *'a process*
((*3*(-**?**-)|- /→ -) [*0,0,78*] *78*)

-readS :: [*id*, *pttrn*, *'b set*,*'a process*] => *'a process*
((*3*(-**?**-)∈- /→ -) [*0*,*0*,*78*] *78*)
-write :: [*id*, *'b*, *'a process*] => *'a process*
((*3*(-**!**-) /→ -) [*0*,*0*,*78*] *78*)
-writeS :: [*'a*, *'a process*] => *'a process*
((*3*- /→ -) [*0*,*78*]*78*)

4.3.6 CSP$_M$-Style Syntax for Communication Primitives

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

Syntax Check:

term‹ *d***?***y* → *c***!***x* → *P* = *Q*›

lemma *read-cont*[*simp*]: *cont P* ⟹ *cont* (λ*y*. *read c A* (*P y*))
⟨*proof*⟩

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

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

corollary *write0-cont-lub* : *contlub*(*Mprefix* {*a*})
⟨*proof*⟩

lemma *write0-contlub* : *contlub*(*write0 a*)
⟨*proof*⟩

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

end

theory *Mndetprefix*
imports *Process Stop Mprefix Ndet*
begin

4.4 Multiple non deterministic operator

lift-definition *Mndetprefix* :: [$'\alpha$ *set*, $'\alpha \Rightarrow {'\alpha}$ *process*] $\Rightarrow$ $'\alpha$ *process*
is $\lambda A\ P.$ *if* $A = \{\}$ *then Rep-process STOP*
else ($\bigsqcap$ $x{\in}A.$ $\mathcal{F}(x \rightarrow P\ x)$, $\bigsqcup$ $x{\in}A.$ $\mathcal{D}(x \rightarrow P\ x)$)
⟨*proof*⟩

syntax
-Mndetprefix :: *pttrn* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *process* $\Rightarrow$ $'a$ *process*
(($3\sqcap$-$\in$- $\rightarrow$ / -) [0, 0, 70] 70)

translations
$\sqcap x{\in}A{\rightarrow}\ P \rightleftharpoons$ *CONST Mndetprefix A* ($\lambda x.\ P$)

lemma *mt-Mndetprefix*[*simp*] : *Mndetprefix* $\{\}$ P = *STOP*
⟨*proof*⟩

lemma *rep-abs-Mndetprefix*: $A \neq \{\} \Longrightarrow$
(*Rep-process* (*Abs-process*($\bigsqcap x{\in}A.\ \mathcal{F}(x \rightarrow P\ x)$,$\bigsqcup x{\in}A.\ \mathcal{D}\ (x \rightarrow P\ x)$))) =
($\bigsqcap x{\in}A.\ \mathcal{F}(x \rightarrow P\ x)$, $\bigsqcup x{\in}A.\ \mathcal{D}\ (x \rightarrow P\ x)$)
⟨*proof*⟩

lemma *F-Mndetprefix*:
$\mathcal{F}$ (*Mndetprefix A P*) = (*if* $A = \{\}$ *then* $\{(s, X).\ s = []\}$ *else* $\bigsqcup$ $x{\in}A.$ $\mathcal{F}$ $(x \rightarrow P\ x)$)
⟨*proof*⟩

lemma *D-Mndetprefix* : $\mathcal{D}$ (*Mndetprefix A P*) = (*if* $A = \{\}$ *then* $\{\}$ *else* $\bigsqcup$ $x{\in}A.$ $\mathcal{D}$ $(x \rightarrow P\ x)$)
⟨*proof*⟩

lemma *T-Mndetprefix*: $\mathcal{T}$ (*Mndetprefix A P*) = (*if* $A = \{\}$ *then* $\{[]\}$ *else* $\bigsqcup$ $x{\in}A.$ $\mathcal{T}$ $(x \rightarrow P\ x)$)
⟨*proof*⟩

Thus we know now, that Mndetprefix yields processes. Direct consequences are the following distributivities:

lemma *Mndetprefix-unit* : $(\sqcap\ x \in \{a\} \rightarrow P\ x)\ = (a \rightarrow P\ a)$
⟨*proof*⟩

lemma *Mndetprefix-Un-distrib* : $A \neq\{\} \Longrightarrow B \neq\{\} \Longrightarrow (\sqcap\ x \in A{\cup}B \rightarrow P\ x) = ((\sqcap\ x \in A \rightarrow P\ x) \sqcap (\sqcap\ x \in B \rightarrow P\ x))$
⟨*proof*⟩

The two lemmas *Mndetprefix* {*?a*} *?P* = *?a* $\rightarrow$ *?P ?a* and $[\![?A \neq \{\};\ ?B \neq \{\}]\!] \Longrightarrow$ *Mndetprefix* (*?A* $\cup$ *?B*) *?P* = *Mndetprefix ?A ?P* $\sqcap$ *Mndetprefix ?B ?P* together give us that *Mndetprefix* can be represented by a fold in the

finite case.

lemma *Mndetprefix-distrib-unit* : $A-\{a\} \neq \{\} \implies (\sqcap x \in \mathit{insert}\ a\ A \rightarrow P\ x) = ((a \rightarrow P\ a) \sqcap (\sqcap x \in A-\{a\} \rightarrow P\ x))$
⟨*proof*⟩

4.4.1 Finite case Continuity

This also implies that Mndetprefix is continuous for the finite A and an arbitrary body f:

lemma *Mndetprefix-cont-finite*[*simp*]:
assumes *finite A*
and $\bigwedge x.\ \mathit{cont}\ (f\ x)$
shows $\mathit{cont}\ (\lambda y.\ \sqcap z \in A \rightarrow f\ z\ y)$
⟨*proof*⟩

4.4.2 General case Continuity

lemma *mono-Mndetprefix*[*simp*] : *monofun* (*Mndetprefix* (*A*::'*a set*))
⟨*proof*⟩

lemma *Mndetprefix-chainpreserving*: *chain Y* $\implies$ *chain* $(\lambda i.\ (\sqcap z \in A \rightarrow Y\ i\ z))$
⟨*proof*⟩

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

lemma *Mndetprefix-cont*[*simp*]: $(\bigwedge x.\ \mathit{cont}\ (f\ x)) \implies \mathit{cont}\ (\lambda y.\ (\sqcap z \in A \rightarrow (f\ z\ y)))$
⟨*proof*⟩

end

Chapter 5

Advanced Induction Schemata

theory *CSP-Induct*

imports *HOLCF*

begin

default-sort *type*

5.1 k-fixpoint-induction

lemma *nat-k-induct* [*case-names base step*]:
fixes k::*nat*
assumes $\forall\, i{<}k.\ P\ i$ **and** $\forall\, n_0.\ (\forall\, i{<}k.\ P\ (n_0{+}i)) \longrightarrow P\ (n_0{+}k)$
shows $P\ (n$::*nat*$)$
⟨*proof*⟩

thm *fix-ind fix-ind2*

lemma *fix-ind-k* [*case-names admissibility base-k-steps step*]:
fixes k::*nat*
assumes *adm*: *adm P*
and *base-k-steps*: $\forall\, i{<}k.\ P\ (\mathit{iterate}\ i{\cdot}f{\cdot}\bot)$
and *step*: $\bigwedge x.\ (\forall\, i{<}k.\ \ P\ (\mathit{iterate}\ i{\cdot}f{\cdot}x)) \Longrightarrow P\ (\mathit{iterate}\ k{\cdot}f{\cdot}x)$
shows $P\ (\mathit{fix}{\cdot}f)$
⟨*proof*⟩

lemma *nat-k-skip-induct* [*case-names lower-bound base-k step*]:
fixes k::*nat*
assumes $k \geq 1$ **and** $\forall\, i{<}k.\ P\ i$ **and** $\forall\, n_0.\ P\ (n_0) \longrightarrow P\ (n_0{+}k)$
shows $P\ (n$::*nat*$)$

⟨*proof*⟩

lemma *fix-ind-k-skip* [*case-names lower-bound admissibility base-k-steps step*]:
fixes *k*::*nat*
assumes *k-1*: $k \geq 1$
and *adm*: *adm P*
and *base-k-steps*: $\forall i{<}k.\ P\ (iterate\ i{\cdot}f{\cdot}\bot)$
and *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (iterate\ k{\cdot}f{\cdot}x)$
shows $P\ (fix{\cdot}f)$
⟨*proof*⟩

thm *parallel-fix-ind*

5.2 Parallel fixpoint-induction

lemma *parallel-fix-ind-inc*[*case-names admissibility base-fst base-snd step*]:
assumes *adm*: *adm* $(\lambda x.\ P\ (fst\ x)\ (snd\ x))$
and *base-fst*: $\bigwedge y.\ P\ \bot\ y$ **and** *base-snd*: $\bigwedge x.\ P\ x\ \bot$
and *step*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (G{\cdot}x)\ y \Longrightarrow P\ x\ (H{\cdot}y) \Longrightarrow P\ (G{\cdot}x)\ (H{\cdot}y)$
shows $P\ (fix{\cdot}G)\ (fix{\cdot}H)$
⟨*proof*⟩

end

Chapter 6

Refinements

theory *Process-Order*
imports *Process Stop*
begin

definition *trace-refine* :: ‹'*a process* ⇒ '*a process* ⇒ *bool*› (**infix** ‹$\sqsubseteq_T$›
60)
where ‹$P \sqsubseteq_T Q \equiv \mathcal{T}\ Q \subseteq \mathcal{T}\ P$›

definition *failure-refine* :: ‹'*a process* ⇒ '*a process* ⇒ *bool*› (**infix** ‹$\sqsubseteq_F$›
60)
where ‹$P \sqsubseteq_F Q \equiv \mathcal{F}\ Q \subseteq \mathcal{F}\ P$›

definition *divergence-refine* :: ‹'*a process* ⇒ '*a process* ⇒ *bool*› (**infix** ‹$\sqsubseteq_D$›
53)
where ‹$P \sqsubseteq_D Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P$›

definition *failure-divergence-refine* :: ‹'*a process* ⇒ '*a process* ⇒ *bool*› (**infix**
‹$\sqsubseteq_{FD}$› *60*)
where ‹$P \sqsubseteq_{FD} Q \equiv P \leq Q$›

definition *trace-divergence-refine* :: ‹'*a process* ⇒ '*a process* ⇒ *bool*› (**infix** ‹$\sqsubseteq_{DT}$›
53)

where ‹$P \sqsubseteq_{DT} Q \equiv P \sqsubseteq_T Q \wedge P \sqsubseteq_D Q$›

6.1 Idempotency

lemma *idem-F*[*simp*]: ‹$P \sqsubseteq_F P$›
and *idem-D*[*simp*]: ‹$P \sqsubseteq_D P$›
and *idem-T*[*simp*]: ‹$P \sqsubseteq_T P$›
and *idem-FD*[*simp*]: ‹$P \sqsubseteq_{FD} P$›
and *idem-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P$›

⟨*proof*⟩

6.2 Some obvious refinements

lemma *BOT-leF*[*simp*]: ‹$\bot \sqsubseteq_F Q$›
and *BOT-leD*[*simp*]: ‹$\bot \sqsubseteq_D Q$›
and *BOT-leT*[*simp*]: ‹$\bot \sqsubseteq_T Q$›
⟨*proof*⟩

6.3 Antisymmetry

lemma *FD-antisym*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q \sqsubseteq_{FD} P \Longrightarrow P = Q$›
⟨*proof*⟩

lemma *DT-antisym*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} P \Longrightarrow P = Q$›
⟨*proof*⟩

6.4 Transitivity

lemma *trans-F*: ‹$P \sqsubseteq_F Q \Longrightarrow Q \sqsubseteq_F S \Longrightarrow P \sqsubseteq_F S$›
and *trans-D*: ‹$P \sqsubseteq_D Q \Longrightarrow Q \sqsubseteq_D S \Longrightarrow P \sqsubseteq_D S$›
and *trans-T*: ‹$P \sqsubseteq_T Q \Longrightarrow Q \sqsubseteq_T S \Longrightarrow P \sqsubseteq_T S$›
and *trans-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow Q \sqsubseteq_{FD} S \Longrightarrow P \sqsubseteq_{FD} S$›
and *trans-DT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} S \Longrightarrow P \sqsubseteq_{DT} S$›
⟨*proof*⟩

6.5 Relations between refinements

lemma *leF-imp-leT*: ‹$P \sqsubseteq_F Q \Longrightarrow P \sqsubseteq_T Q$›
and *leFD-imp-leF*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow P \sqsubseteq_F Q$›
and *leFD-imp-leD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow P \sqsubseteq_D Q$›
and *leDT-imp-leD*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow P \sqsubseteq_D Q$›
and *leDT-imp-leT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow P \sqsubseteq_T Q$›
and *leF-leD-imp-leFD*: ‹$P \sqsubseteq_F Q \Longrightarrow P \sqsubseteq_D Q \Longrightarrow P \sqsubseteq_{FD} Q$›
and *leD-leT-imp-leDT*: ‹$P \sqsubseteq_D Q \Longrightarrow P \sqsubseteq_T Q \Longrightarrow P \sqsubseteq_{DT} Q$›
⟨*proof*⟩

6.6 More obvious refinements

lemma *BOT-leFD*[*simp*]: ‹$\bot \sqsubseteq_{FD} Q$›
and *BOT-leDT*[*simp*]: ‹$\bot \sqsubseteq_{DT} Q$›
⟨*proof*⟩

lemma *leDT-STOP*[*simp*]: ‹$P \sqsubseteq_{DT} STOP$›
⟨*proof*⟩

lemma *leD-STOP*[*simp*]: ‹$P \sqsubseteq_D STOP$›
and *leT-STOP*[*simp*]: ‹$P \sqsubseteq_T STOP$›
⟨*proof*⟩

6.7 Admissibility

lemma *le-F-adm*[*simp*]: ‹*cont* (u::($'a$::*cpo*) ⇒ $'b$ *process*) ⟹ *monofun* v ⟹ *adm*($\lambda x.\ u\ x \sqsubseteq_F v\ x$)›
⟨*proof*⟩

lemma *le-T-adm*[*simp*]: ‹*cont* (u::($'a$::*cpo*) ⇒ $'b$ *process*) ⟹ *monofun* v ⟹ *adm*($\lambda x.\ u\ x \sqsubseteq_T v\ x$)›
⟨*proof*⟩

lemma *le-D-adm*[*simp*]: ‹*cont* (u::($'a$::*cpo*) ⇒ $'b$ *process*) ⟹ *monofun* v ⟹ *adm*($\lambda x.\ u\ x \sqsubseteq_D v\ x$)›
⟨*proof*⟩

lemmas *le-FD-adm*[*simp*] = *le-adm*[*folded failure-divergence-refine-def*]

lemma *le-DT-adm*[*simp*]: ‹*cont* (u::($'a$::*cpo*) ⇒ $'b$ *process*) ⟹ *monofun* v ⟹ *adm*($\lambda x.\ u\ x \sqsubseteq_{DT} v\ x$)›
⟨*proof*⟩

end

Chapter 7

The "Laws" of CSP

theory *CSP-Laws*
imports *Bot Skip Stop Det Ndet Mprefix Mndetprefix Seq Hiding Sync Renaming*
HOL−Eisbach.Eisbach
begin

method *exI* **for** $y{::}'a$ = (*rule-tac exI*[**where** $x = y$])

7.1 General Laws

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

lemma *BOT-less1*[*simp*]: $\bot \leq (X{::}'a\ process)$
⟨*proof*⟩

lemma *BOT-less2*[*simp*]: $BOT \leq (X{::}'a\ process)$
⟨*proof*⟩

7.2 Deterministic Choice Operator Laws

lemma *mono-Det-FD-onside*[*simp*]: $P \leq P' \Longrightarrow (P \,\Box\, S) \leq (P' \,\Box\, S)$
⟨*proof*⟩

lemma *mono-Det-FD*[*simp*]: $[\![P \leq P';\ S \leq S']\!] \Longrightarrow (P \,\Box\, S) \leq (P' \,\Box\, S')$
⟨*proof*⟩

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

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

lemma *Det-STOP*: $(P \,\Box\, STOP) = P$

⟨*proof*⟩

lemma *Det-id*: $(P \ \Box \ P) = P$
⟨*proof*⟩

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

7.3 NonDeterministic Choice Operator Laws

lemma *mono-Ndet-FD*[*simp*]: $[\![P \leq P'; S \leq S']\!] \Longrightarrow (P \sqcap S) \leq (P' \sqcap S')$
⟨*proof*⟩

lemma *mono-Ndet-FD-left*[*simp*]: $P \leq Q \Longrightarrow (P \sqcap S) \leq Q$
⟨*proof*⟩

lemma *mono-Ndet-FD-right*[*simp*]: $P \leq Q \Longrightarrow (S \sqcap P) \leq Q$
⟨*proof*⟩

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

lemma *Ndet-BOT*: $(P \sqcap \bot) = \bot$
⟨*proof*⟩

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

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

7.3.1 Multi-Operators laws

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

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

lemma *Ndet-FD-Det*: $(P \sqcap Q) \leq (P \ \Box \ Q)$
⟨*proof*⟩

7.4 Sequence Operator Laws

7.4.1 Preliminaries

definition *F-minus-D-Seq* **where**
F-minus-D-Seq P Q $\equiv \{(t, X).\ (t, X \cup \{tick\}) \in \mathcal{F}\ P \wedge tickFree\ t\} \cup$

$\{(t,\ X).\ \exists\ t1\ t2.\ t = t1\ @\ t2 \wedge t1\ @\ [tick] \in \mathcal{T}\ P \wedge (t2,\ X) \in \mathcal{F}\ Q\}$

lemma *F-minus-D-Seq-opt*: $(a,b) \in \mathcal{F}(P\ ;\ Q) = (a \in \mathcal{D}(P\ ;\ Q) \vee (a,b) \in$ *F-minus-D-Seq* $P\ Q)$
⟨*proof*⟩

lemma *Process-eq-spec-optimized-Seq* :
$((P\ ;\ Q) = (U\ ;\ S)) = (\mathcal{D}\ (P\ ;\ Q) = \mathcal{D}\ (U\ ;\ S) \wedge$
F-minus-D-Seq $P\ Q \subseteq \mathcal{F}\ (U\ ;\ S) \wedge$
F-minus-D-Seq $U\ S \subseteq \mathcal{F}\ (P\ ;\ Q))$
⟨*proof*⟩

7.4.2 Laws

lemma *mono-Seq-FD*[*simp*]: $[\![P \leq P';\ S \leq S']\!] \Longrightarrow (P\ ;\ S) \leq (P'\ ;\ S')$
⟨*proof*⟩

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

lemma *BOT-Seq*: $(\bot\ ;\ P) = \bot$
⟨*proof*⟩

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

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

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

7.4.3 Multi-Operators laws

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

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

lemma *Seq-assoc-D*: $\mathcal{D}\ (P\ ;\ (Q\ ;\ S)) = \mathcal{D}\ ((P\ ;\ Q)\ ;\ S)$
⟨*proof*⟩

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

7.5 The Multi-Prefix Operator Laws

lemma *mono-Mprefix-FD*[*simp*]: $\forall\, x \in A.\ P\ x \leq P'\ x \Longrightarrow Mprefix\ A\ P \leq Mprefix\ A\ P'$
⟨*proof*⟩

lemmas *mono-Mprefix-ref* = *mono-Mprefix0*

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

7.5.1 Multi-Operators laws

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

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

7.5.2 Derivative Operators laws

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

lemma *mono-read-FD*: $(\bigwedge x.\ P\ x \leq Q\ x) \Longrightarrow\ (c?x \rightarrow (P\ x)) \leq (c?x \rightarrow (Q\ x))$
⟨*proof*⟩

lemma *mono-write-FD*: $(P \leq Q) \Longrightarrow\ (c!x \rightarrow P) \leq (c!x \rightarrow Q)$
⟨*proof*⟩

lemma *mono-write0-FD*: $P \leq Q \Longrightarrow (a \rightarrow P) \leq (a \rightarrow Q)$
⟨*proof*⟩

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

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

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

lemma *write0-Ndet*: $(a \rightarrow (P \sqcap Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
⟨*proof*⟩

lemma *write0-Det-Ndet*: $((a \rightarrow P) \Box (a \rightarrow Q)) = ((a \rightarrow P) \sqcap (a \rightarrow Q))$
⟨*proof*⟩

lemma *Mprefix-Det*: $\langle(\Box e{\in}A \rightarrow P\ e) \Box (\Box e{\in}A \rightarrow Q\ e) = \Box e{\in}A \rightarrow (P\ e \sqcap Q\ e)\rangle$

⟨*proof*⟩

7.6 The Hiding Operator Laws

7.6.1 Preliminaries

lemma *elemDIselemHD*: ‹$t \in \mathcal{D}\ P \Longrightarrow$ *trace-hide* $t\ (ev\ `\ A) \in \mathcal{D}\ (P \setminus A)$›
⟨*proof*⟩

lemma *length-strict-mono*: *strict-mono* (f::*nat* ⇒ $'a$ *list*) ⟹ *length* ($f\ i$) ≥ i + *length* (f *0*)
⟨*proof*⟩

lemma *mono-trace-hide*: $a \leq b \Longrightarrow$ *trace-hide* $a\ (ev\ `\ A) \leq$ *trace-hide* $b\ (ev\ `\ A)$
⟨*proof*⟩

lemma *mono-constant*:
assumes *mono* (f::*nat* ⇒ $'a$ *event list*) **and** ∀ i. $f\ i \leq a$
shows ∃ i. ∀ $j \geq i$. $f\ j = f\ i$
⟨*proof*⟩

lemma *elemTIselemHT*: ‹$t \in \mathcal{T}\ P \Longrightarrow$ *trace-hide* $t\ (ev\ `\ A) \in \mathcal{T}\ (P \setminus A)$›
⟨*proof*⟩

lemma *Hiding-Un-D1*: ‹$\mathcal{D}\ (P \setminus (A \cup B)) \subseteq \mathcal{D}\ ((P \setminus A) \setminus B)$›
⟨*proof*⟩

lemma *Hiding-Un-D2*: ‹*finite* $A \Longrightarrow \mathcal{D}\ ((P \setminus A) \setminus B) \subseteq \mathcal{D}\ (P \setminus (A \cup B))$›
⟨*proof*⟩

lemma *Hiding-Un-D*: ‹*finite* $A \Longrightarrow \mathcal{D}\ ((P \setminus A) \setminus B) = \mathcal{D}\ (P \setminus (A \cup B))$›
⟨*proof*⟩

7.6.2 Laws

lemma *mono-Hiding-FD*[*simp*]: ‹$P \leq Q \Longrightarrow P \setminus A \leq Q \setminus A$›
⟨*proof*⟩

lemmas *mono-Hiding-ref* = *mono-Hiding*

lemma *Hiding-Un*: ‹*finite* $A \Longrightarrow P \setminus (A \cup B) = (P \setminus A) \setminus B$›
⟨*proof*⟩

lemma *Hiding-set-BOT*: ‹$(\bot \setminus A) = \bot$›
⟨*proof*⟩

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

lemma *Hiding-set-SKIP*: ‹(*SKIP* \ *A*) = *SKIP*›
⟨*proof*⟩

lemma *Hiding-set-empty*: ‹(*P* \ {}) = *P*›
⟨*proof*⟩

7.6.3 Multi-Operators laws

lemma *Hiding-Ndet*: ‹(*P* ⊓ *Q*) \ *A* = ((*P* \ *A*) ⊓ (*Q* \ *A*))›
⟨*proof*⟩

lemma *Hiding-Mprefix-distr*:
‹(*B* ∩ *A*) = {} ⟹ ((*Mprefix A P*) \ *B*) = (*Mprefix A* (λ*x*. ((*P x*) \ *B*)))›
⟨*proof*⟩

lemma *no-Hiding-read*: ‹(∀ *y*. *c y* ∉ *B*) ⟹ ((*c*?*x* → (*P x*)) \ *B*) = (*c*?*x* → ((*P x*) \ *B*))›
⟨*proof*⟩

lemma *no-Hiding-write0*: ‹*a* ∉ *B* ⟹ ((*a* → *P*) \ *B*) = (*a* → (*P* \ *B*))›
⟨*proof*⟩

lemma *Hiding-write0*: ‹*a* ∈ *B* ⟹ ((*a* → *P*) \ *B*) = (*P* \ *B*)›
⟨*proof*⟩

lemma *no-Hiding-write*: ‹(∀ *y*. *c y* ∉ *B*) ⟹ ((*c*!*a* → *P*) \ *B*) = (*c*!*a* → (*P* \ *B*))›
⟨*proof*⟩

lemma *Hiding-write*: ‹(*c a*) ∈ *B* ⟹ ((*c*!*a* → *P*) \ *B*) = (*P* \ *B*)›
⟨*proof*⟩

7.7 The Sync Operator Laws

7.7.1 Preliminaries

lemma *SyncTlEmpty*:*a setinterleaves* ([], *u*), *A*) ⟹ *tl a setinterleaves* (([], *tl u*), *A*)
⟨*proof*⟩

lemma *SyncHd-Tl*:
a setinterleaves ((*t*, *u*), *A*) ∧ *hd t* ∈ *A* ∧ *hd u* ∉ *A*
⟹ *hd a* = *hd u* ∧ *tl a setinterleaves* ((*t*, *tl u*), *A*)
⟨*proof*⟩

lemma *SyncHdAddEmpty*:
(*tl a*) *setinterleaves* ([], *u*), *A*) ∧ *hd a* ∉ *A* ∧ *a* ≠ []
⟹ *a setinterleaves* (([], *hd a* # *u*), *A*)
⟨*proof*⟩

lemma *SyncHdAdd*:
(*tl a*) *setinterleaves* ((*t*, *u*), *A*) ∧ *hd a* ∉ *A* ∧ *hd t* ∈ *A* ∧ *a* ≠ []
⟹ *a setinterleaves* ((*t*, *hd a* # *u*), *A*)
⟨*proof*⟩

lemmas *SyncHdAdd1* = *SyncHdAdd*[*of a*#*r*, *simplified*] **for** *a r*

lemma *SyncSameHdTl*:
a setinterleaves ((*t*, *u*), *A*) ∧ *hd t* ∈ *A* ∧ *hd u* ∈ *A*
⟹ *hd t* = *hd u* ∧ *hd a* = *hd t* ∧ (*tl a*) *setinterleaves* ((*tl t*, *tl u*), *A*)
⟨*proof*⟩

lemma *SyncSingleHeadAdd*:
a setinterleaves ((*t*, *u*), *A*) ∧ *h* ∉ *A*
⟹ (*h*#*a*) *setinterleaves* ((*h*#*t*, *u*), *A*)
⟨*proof*⟩

lemma *TickLeftSync*:
tick ∈ *A* ∧ *front-tickFree t* ∧ *s setinterleaves* (([*tick*], *t*), *A*) ⟹ *s* = *t* ∧ *last t* = *tick*
⟨*proof*⟩

lemma *EmptyLeftSync*:*s setinterleaves* (([], *t*), *A*) ⟹ *s* = *t* ∧ *set t* ∩ *A* = {}
⟨*proof*⟩

lemma *tick-T-F*:*t*@[*tick*] ∈ $\mathcal{T}$ *P* ⟹ (*t*@[*tick*], *X*) ∈ $\mathcal{F}$ *P*
⟨*proof*⟩

lemma *event-set*: (*e*::'*a event*) ∈ *insert tick* (*range ev*)
⟨*proof*⟩

lemma *Mprefix-Sync-distr1-D*:
A ⊆ *S*
⟹ *B* ⊆ *S*
⟹ $\mathcal{D}$ (*Mprefix A P* ⟦*S*⟧ *Mprefix B Q*) = $\mathcal{D}$ (□ *x* ∈ *A* ∩ *B* → (*P x* ⟦*S*⟧ *Q x*))

⟨*proof*⟩

lemma *Hiding-interleave*:
A ∩ *C* = {}
⟹ *r setinterleaves* ((*t*, *u*), *C*)
⟹ (*trace-hide r A*) *setinterleaves* ((*trace-hide t A*, *trace-hide u A*), *C*)
⟨*proof*⟩

lemma *non-Sync-interleaving*:
(*set t* ∪ *set u*) ∩ *C* = {} ⟹ *setinterleaving* (*t*, *C*, *u*) ≠ {}
⟨*proof*⟩

lemma *interleave-Hiding*:

$A \cap C = \{\}$
$\Longrightarrow$ *ra setinterleaves* ((*trace-hide t A*, *trace-hide u A*), *C*)
$\Longrightarrow \exists\, r.$ *ra* = *trace-hide r A* $\wedge$ *r setinterleaves* ((*t*, *u*), *C*)
⟨*proof*⟩

lemma *interleave-size*:
s setinterleaves ((*t*,*u*), *C*) $\Longrightarrow$ *length s* = *length t* + *length u* − *length*(*filter* ($\lambda x.$ $x \in C$) *t*)
⟨*proof*⟩

lemma *interleave-eq-size*:
s setinterleaves ((*t*,*u*), *C*) $\Longrightarrow$ *s′ setinterleaves* ((*t*,*u*), *C*) $\Longrightarrow$ *length s* = *length s′*
⟨*proof*⟩

lemma *interleave-set*: *s setinterleaves* ((*t*,*u*), *C*) $\Longrightarrow$ *set t* $\cup$ *set u* $\subseteq$ *set s*
⟨*proof*⟩

lemma *interleave-order*: *s setinterleaves* ((*t1*@*t2*,*u*), *C*) $\Longrightarrow$ *set*(*t2*) $\subseteq$ *set*(*drop* (*length t1*) *s*)
⟨*proof*⟩

lemma *interleave-append*:
s setinterleaves ((*t1*@*t2*,*u*), *C*)
$\Longrightarrow \exists$ *u1 u2 s1 s2*. *u* = *u1*@*u2* $\wedge$ *s* = *s1*@*s2* $\wedge$
s1 setinterleaves ((*t1*,*u1*), *C*) $\wedge$ *s2 setinterleaves* ((*t2*,*u2*), *C*)
⟨*proof*⟩

lemma *interleave-append-sym*:
s setinterleaves ((*t*,*u1*@*u2*), *C*)
$\Longrightarrow \exists$ *t1 t2 s1 s2*. *t* = *t1*@*t2* $\wedge$ *s* = *s1*@*s2* $\wedge$
s1 setinterleaves ((*t1*,*u1*), *C*) $\wedge$ *s2 setinterleaves* ((*t2*,*u2*), *C*)
⟨*proof*⟩

lemma *interleave-append-tail*:
s setinterleaves ((*t1*,*u*), *C*) $\Longrightarrow$ (*set t2*) $\cap$ *C* = {} $\Longrightarrow$ (*s*@*t2*) *setinterleaves* ((*t1*@*t2*,*u*), *C*)
⟨*proof*⟩

lemma *interleave-append-tail-sym*:
s setinterleaves ((*t*,*u1*), *C*) $\Longrightarrow$ (*set u2*) $\cap$ *C* = {} $\Longrightarrow$ (*s*@*u2*) *setinterleaves* ((*t*,*u1*@*u2*), *C*)
⟨*proof*⟩

lemma *interleave-assoc-1*:
tu setinterleaves ((*t*, *u*), *A*)
$\Longrightarrow$ *tuv setinterleaves* ((*tu*, *v*), *A*)
$\Longrightarrow \exists\, uv.$ *uv setinterleaves* ((*u*, *v*), *A*) $\wedge$ *tuv setinterleaves* ((*t*, *uv*), *A*)
⟨*proof*⟩

lemma *interleave-assoc-2*:
assumes $*$:*uv setinterleaves* $((u,\ v),\ A)$ **and**
$**$:*tuv setinterleaves* $((t,\ uv),\ A)$
shows $\exists\, tu.$ *tu setinterleaves* $((t,\ u),\ A) \wedge$ *tuv setinterleaves* $((tu,\ v),\ A)$
⟨*proof*⟩

7.7.2 Laws

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

lemma *mono-Sync-FD-oneside*[*simp*]: $P \leq P' \Longrightarrow (P \llbracket\ S\ \rrbracket\ Q) \leq (P' \llbracket\ S\ \rrbracket\ Q)$
⟨*proof*⟩

lemma *mono-Sync-FD*[*simp*]: $[\![P \leq P';\ Q \leq Q']\!] \Longrightarrow (P \llbracket\ S\ \rrbracket\ Q) \leq (P' \llbracket\ S\ \rrbracket\ Q')$
⟨*proof*⟩

lemma *mono-Sync-ref*: $[\![P \sqsubseteq P';\ Q \sqsubseteq Q']\!] \Longrightarrow (P \llbracket\ S\ \rrbracket\ Q) \sqsubseteq (P' \llbracket\ S\ \rrbracket\ Q')$
⟨*proof*⟩

lemma *Sync-BOT*: $(P \llbracket\ S\ \rrbracket\ \bot) = \bot$
⟨*proof*⟩

lemma *Sync-SKIP-SKIP*: $(SKIP \llbracket\ S\ \rrbracket\ SKIP) = SKIP$
⟨*proof*⟩

lemma *Sync-SKIP-STOP*: $(SKIP \llbracket S \rrbracket\ STOP) = STOP$
⟨*proof*⟩

7.7.3 Multi-Operators laws

lemma *Mprefix-Sync-distr*:
$\langle(\Box\ x \in A \cup A' \to P\ x) \llbracket\ S\ \rrbracket (\Box\ y \in B \cup B' \to Q\ y) =$
$(\Box\ x \in A \to (P\ x \llbracket\ S\ \rrbracket (\Box\ y \in B \cup B' \to Q\ y))) \Box$
$(\Box\ y \in B \to ((\Box\ x \in A \cup A' \to P\ x) \llbracket\ S\ \rrbracket Q\ y)) \Box$
$(\Box\ x \in A' \cap B' \to (P\ x \llbracket\ S\ \rrbracket Q\ x))\rangle$
(**is** ‹*?lhs A A' P B B' Q* = *?rhs A A' P B B' Q*›)
if *sets-assms*: ‹$A \cap S = \{\}$› ‹$A' \subseteq S$› ‹$B \cap S = \{\}$› ‹$B' \subseteq S$›

⟨*proof*⟩

lemma *Mprefix-Sync-distr-bis*:
$\langle$(*Mprefix A P*) $[\![S]\!]$ (*Mprefix B Q*) =
$(\Box x \in A - S \to (P\ x\ [\![S]\!]$ *Mprefix B Q*))
$\Box\ (\Box y \in B - S \to ($*Mprefix A P* $[\![S]\!]\ Q\ y))$
$\Box\ (\Box x \in A \cap B \cap S \to (P\ x\ [\![S]\!]\ Q\ x))\rangle$
⟨*proof*⟩

lemmas *Mprefix-Sync-distr-subset* = *Mprefix-Sync-distr*[**where** $A = \langle\{\}\rangle$
and $B = \langle\{\}\rangle$, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*]]
and *Mprefix-Sync-distr-indep* = *Mprefix-Sync-distr*[**where** $A' = \langle\{\}\rangle$
and $B' = \langle\{\}\rangle$, *simplified*, *simplified Mprefix-STOP Det-STOP*]
and *Mprefix-Sync-distr-right* = *Mprefix-Sync-distr*[**where** $A = \langle\{\}\rangle$
and $B' = \langle\{\}\rangle$, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*], *rotated*]
and *Mprefix-Sync-distr-left* = *Mprefix-Sync-distr*[**where** $A' = \langle\{\}\rangle$
and $B = \langle\{\}\rangle$, *simplified*, *simplified Mprefix-STOP Det-STOP trans*[*OF Det-commute Det-STOP*]]

lemma *Mprefix-Sync-SKIP*: $(Mprefix\ B\ P\ [\![\ S\ ]\!]\ SKIP) = (\Box\ x \in\ B - S \rightarrow (P\ x\ [\![\ S\ ]\!]\ SKIP))$
⟨*proof*⟩

lemma *Sync-Ndet-left-distrib*: $((P \sqcap Q)\ [\![\ S\ ]\!]\ M) = ((P\ [\![\ S\ ]\!]\ M) \sqcap (Q\ [\![\ S\ ]\!]\ M))$
⟨*proof*⟩

lemma *Sync-Ndet-right-distrib*: $(M\ [\![\ S\ ]\!]\ (P \sqcap Q)) = ((M\ [\![\ S\ ]\!]\ P) \sqcap (M\ [\![\ S\ ]\!]\ Q))$
⟨*proof*⟩

lemma *prefix-Sync-SKIP1*: $a \notin S \Longrightarrow (a \rightarrow P\ [\![S]\!]\ SKIP) = (a \rightarrow (P\ [\![S]\!]\ SKIP))$
⟨*proof*⟩

lemma *prefix-Sync-SKIP2*: $a \in S \Longrightarrow (a \rightarrow P\ [\![S]\!]\ SKIP) = STOP$
⟨*proof*⟩

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

lemma *prefix-Sync1*: $[\![a \in S;\ b \in S;\ a \neq b]\!] \Longrightarrow (a \rightarrow P\ [\![S]\!]\ (b \rightarrow Q)) = STOP$
⟨*proof*⟩

lemma *prefix-Sync2*: $a \in S \Longrightarrow ((a \rightarrow P)\ [\![S]\!]\ (a \rightarrow Q)) = (a \rightarrow (P\ [\![S]\!]\ Q))$
⟨*proof*⟩

lemma *prefix-Sync3*: $[\![a \in S;\ b \notin S]\!] \Longrightarrow (a \rightarrow P\ [\![S]\!]\ (b \rightarrow Q)) = (b \rightarrow ((a \rightarrow P)\ [\![S]\!]\ Q))$
⟨*proof*⟩

lemma *Hiding-Sync-D1*: ‹*finite* $A \Longrightarrow A \cap S = \{\} \Longrightarrow\ \mathcal{D}\ ((P\ [\![S]\!]\ Q) \setminus A) \subseteq \mathcal{D}\ ((P \setminus A)\ [\![S]\!]\ (Q \setminus A))$›
⟨*proof*⟩

lemma *Hiding-Sync-D2*:
assumes $*{:}A \cap S = \{\}$
shows $\langle \mathcal{D}\ ((P \setminus A)\ [\![S]\!]\ (Q \setminus A)) \subseteq \mathcal{D}\ ((P\ [\![S]\!]\ Q) \setminus A)\rangle$
⟨*proof*⟩

lemma *Hiding-Sync*:
$\langle$*finite* $A \Longrightarrow A \cap S = \{\} \Longrightarrow ((P\ [\![S]\!]\ Q) \setminus A) = ((P \setminus A)\ [\![S]\!]\ (Q \setminus A))\rangle$
⟨*proof*⟩

lemma *Sync-assoc-oneside-D*: $\mathcal{D}\ (P\ [\![S]\!]\ (Q\ [\![S]\!]\ T)) \subseteq \mathcal{D}\ (P\ [\![S]\!]\ Q\ [\![S]\!]\ T)$
⟨*proof*⟩

lemma *Sync-assoc-oneside*: $((P\ [\![S]\!]\ Q)\ [\![S]\!]\ T) \leq (P\ [\![S]\!]\ (Q\ [\![S]\!]\ T))$
⟨*proof*⟩

lemma *Sync-assoc*: $((P\ [\![S]\!]\ Q)\ [\![S]\!]\ T) = (P\ [\![S]\!]\ (Q\ [\![S]\!]\ T))$
⟨*proof*⟩

7.7.4 Derivative Operators laws

lemmas *Par-BOT* = *Sync-BOT*[**where** $S = \langle UNIV \rangle$]
and *Par-SKIP-SKIP* = *Sync-SKIP-SKIP*[**where** $S = \langle UNIV \rangle$]
and *Par-SKIP-STOP* = *Sync-SKIP-STOP*[**where** $S = \langle UNIV \rangle$]
and *Par-commute* = *Sync-commute*[**where** $S = \langle UNIV \rangle$]
and *Par-assoc* = *Sync-assoc*[**where** $S = \langle UNIV \rangle$]
and *Par-Ndet-right-distrib* = *Sync-Ndet-right-distrib*[**where** $S = \langle UNIV \rangle$]
and *Par-Ndet-left-distrib* = *Sync-Ndet-left-distrib*[**where** $S = \langle UNIV \rangle$]
and *Mprefix-Par-SKIP* = *Mprefix-Sync-SKIP*[**where** $S = \langle UNIV \rangle$]
and *prefix-Par-SKIP* = *prefix-Sync-SKIP*[**where** $S = \langle UNIV \rangle$, *simplified*]
and *mono-Par-FD* = *mono-Sync-FD*[**where** $S = \langle UNIV \rangle$]
and *mono-Par-ref* = *mono-Sync-ref*[**where** $S = \langle UNIV \rangle$]
and *prefix-Par1* = *prefix-Sync1*[**where** $S = \langle UNIV \rangle$, *simplified*]
and *prefix-Par2* = *prefix-Sync2*[**where** $S = \langle UNIV \rangle$, *simplified*]
and *Mprefix-Par-distr* = *Mprefix-Sync-distr-subset*[**where** $S = \langle UNIV \rangle$, *simplified*]

and *Inter-BOT* = *Sync-BOT*[**where** $S = \langle\{\}\rangle$]
and *Inter-SKIP-SKIP* = *Sync-SKIP-SKIP*[**where** $S = \langle\{\}\rangle$]
and *Inter-SKIP-STOP* = *Sync-SKIP-STOP*[**where** $S = \langle\{\}\rangle$]
and *Inter-commute* = *Sync-commute*[**where** $S = \langle\{\}\rangle$]
and *Inter-assoc* = *Sync-assoc*[**where** $S = \langle\{\}\rangle$]
and *Inter-Ndet-right-distrib* = *Sync-Ndet-right-distrib*[**where** $S = \langle\{\}\rangle$]
and *Inter-Ndet-left-distrib* = *Sync-Ndet-left-distrib*[**where** $S = \langle\{\}\rangle$]
and *Mprefix-Inter-SKIP* = *Mprefix-Sync-SKIP*[**where** $S = \langle\{\}\rangle$, *simplified*, *simplified Mprefix-STOP*]
and *prefix-Inter-SKIP* = *prefix-Sync-SKIP*[**where** $S = \langle\{\}\rangle$, *simplified*]
and *mono-Inter-FD* = *mono-Sync-FD*[**where** $S = \langle\{\}\rangle$]
and *mono-Inter-ref* = *mono-Sync-ref*[**where** $S = \langle\{\}\rangle$]
and *Hiding-Inter* = *Hiding-Sync*[**where** $S = \langle\{\}\rangle$, *simplified*]

and *Mprefix-Inter-distr* = *Mprefix-Sync-distr-indep*[**where** $S = \langle\{\}\rangle$, *simplified*]
and *prefix-Inter* = *Mprefix-Sync-distr-indep*[*of* $\langle\{a\}\rangle$ $\langle\{\}\rangle$ $\langle\{b\}\rangle$ $\langle\lambda x.\ P\rangle$ $\langle\lambda x.\ Q\rangle$,
simplified, *folded write0-def*] **for** $a\ P\ b\ Q$

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

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

lemma *Par-STOP*: $P \neq \bot \Longrightarrow (P \parallel STOP) = STOP$
⟨*proof*⟩

7.8 Multiple Non Deterministic Operator Laws

lemma *Mprefix-refines-Mndetprefix*: $(\sqcap\ x \in A \to P\ x) \leq (\Box\ x \in A \to P\ x)$
⟨*proof*⟩

lemma *Mndetprefix-refine-FD*: $a \in A \Longrightarrow (\sqcap\ x \in A \to (P\ x)) \leq (a \to (P\ a))$
⟨*proof*⟩

lemma *Mndetprefix-subset-FD*:$A \neq \{\} \Longrightarrow (\sqcap xa \in A \cup B \to\ P) \leq (\sqcap xa \in A \to P)$
⟨*proof*⟩

lemma *mono-Mndetprefix-FD*[*simp*]: $\forall\, x \in A.\ P\ x \leq P'\ x \Longrightarrow (\sqcap\ x \in A \to (P\ x)) \leq (\sqcap\ x \in A \to (P'\ x))$
⟨*proof*⟩

lemma *Mndetprefix-FD-subset* : $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow ((\sqcap\ x \in B \to P\ x)\ \leq (\sqcap\ x \in A \to P\ x))$
⟨*proof*⟩

7.9 Infra-structure for Communication Primitives

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

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

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

lemma *write-read-non-Sync*:
$[\![d\ a \notin S;\ \bigwedge y.\ c\ y \in S\ ]\!] \Longrightarrow (d!a \rightarrow P)\ [\![S]\!]\ (c?x \rightarrow Q\ x) = d!a \rightarrow (P\ [\![S]\!]\ (c?x \rightarrow Q\ x))$
⟨*proof*⟩

lemma *write0-read-non-Sync*:
$[\![d \in S;\ \bigwedge y.\ c\ y \notin S]\!] \Longrightarrow (d \rightarrow P)\ [\![S]\!]\ (c?x \rightarrow Q\ x) = c?x \rightarrow ((d \rightarrow P)\ [\![S]\!]\ Q\ x)$
⟨*proof*⟩

lemma *write0-write-non-Sync*:
$[\![d\ a \notin C;\ c \in C]\!] \Longrightarrow (c \rightarrow Q)\ [\![C]\!]\ (d!a \rightarrow P) = d!a \rightarrow ((c \rightarrow Q)\ [\![C]\!]\ P)$
⟨*proof*⟩

7.10 Renaming operator laws

lemma *Renaming-BOT*: ‹*Renaming* $\bot\ f = \bot$›
⟨*proof*⟩

lemma *Renaming-STOP*: ‹*Renaming STOP f* = *STOP*›
⟨*proof*⟩

lemma *Renaming-SKIP*: ‹*Renaming SKIP f* = *SKIP*›
⟨*proof*⟩

lemma *Renaming-Ndet*: ‹*Renaming* $(P \sqcap Q)\ f$ = *Renaming P f* $\sqcap$ *Renaming Q f*›
⟨*proof*⟩

lemma *Renaming-Det*: ‹*Renaming* $(P \mathbin{\square} Q)\ f$ = *Renaming P f* $\mathbin{\square}$ *Renaming Q f*›
⟨*proof*⟩

lemma *mono-Mprefix-eq*: ‹$\forall a \in A.\ P\ a = Q\ a \Longrightarrow Mprefix\ A\ P = Mprefix\ A\ Q$›
⟨*proof*⟩

lemma *mono-Mndetprefix-eq*: ‹$\forall a \in A.\ P\ a = Q\ a \Longrightarrow Mndetprefix\ A\ P = Mndetprefix\ A\ Q$›
⟨*proof*⟩

lemma *Renaming-Mprefix*: ‹$Renaming\ (\Box\ a \in A \rightarrow P\ (f\ a))\ f = \Box\ b \in f\ `\ A \rightarrow Renaming\ (P\ b)\ f$›
⟨*proof*⟩

lemma *Renaming-Mprefix-inj-on*:
‹$Renaming\ (Mprefix\ A\ P)\ f = \Box\ b \in f\ `\ A \rightarrow Renaming\ (P\ (THE\ a.\ a \in A \wedge f\ a = b))\ f$›
if *inj-on-f*: ‹*inj-on f A*›
⟨*proof*⟩

corollary *Renaming-Mprefix-inj*:
‹$Renaming\ (Mprefix\ A\ P)\ f = \Box\ b \in f\ `\ A \rightarrow Renaming\ (P\ (THE\ a.\ f\ a = b))\ f$› **if** *inj-f*: ‹*inj f*›
⟨*proof*⟩

A smart application (as f is of course injective on the singleton $\{a\}$)

corollary *Renaming-prefix*: ‹$Renaming\ (a \rightarrow P)\ f = (f\ a \rightarrow Renaming\ P\ f)$›
⟨*proof*⟩

lemma *Renaming-Mndetprefix*: ‹$Renaming\ (\sqcap\ a \in A \rightarrow P\ (f\ a))\ f = \sqcap\ b \in f\ `\ A \rightarrow Renaming\ (P\ b)\ f$›
⟨*proof*⟩

corollary *Renaming-Mndetprefix-inj-on*:
‹$Renaming\ (Mndetprefix\ A\ P)\ f = \sqcap\ b \in f\ `\ A \rightarrow Renaming\ (P\ (THE\ a.\ a \in A \wedge f\ a = b))\ f$›
if *inj-on-f*: ‹*inj-on f A*›
⟨*proof*⟩

corollary *Renaming-Mndetprefix-inj*:
‹*Renaming* (*Mndetprefix A P*) *f* = ⊓ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*›
if *inj-f*: ‹*inj f*›
⟨*proof*⟩

lemma *Renaming-Seq*: ‹*Renaming* (*P* ; *Q*) *f* = *Renaming P f* ; *Renaming Q f*›
⟨*proof*⟩

lemmas *Sync-rules* = *prefix-Sync2*
read-read-Sync read-read-non-Sync
write-read-Sync write-read-non-Sync
write0-read-non-Sync
write0-write-non-Sync

lemmas *Hiding-rules* = *no-Hiding-read no-Hiding-write Hiding-write Hiding-write0*

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

end

Chapter 8

The refinements laws

theory *CSP*
imports *Process-Order CSP-Laws CSP-Induct*
begin

8.1 Monotonicity

lemma *mono-Det-D*[*simp*]: ‹⟦$P \sqsubseteq_D P'$; $S \sqsubseteq_D S'$⟧ ⟹ $(P \mathbin{\square} S) \sqsubseteq_D (P' \mathbin{\square} S')$›
⟨*proof*⟩

lemma *mono-Det-T*[*simp*]: ‹⟦$P \sqsubseteq_T P'$; $S \sqsubseteq_T S'$⟧ ⟹ $(P \mathbin{\square} S) \sqsubseteq_T (P' \mathbin{\square} S')$›
⟨*proof*⟩

corollary *mono-Det-DT*[*simp*]: ‹⟦$P \sqsubseteq_{DT} P'$; $S \sqsubseteq_{DT} S'$⟧ ⟹ $(P \mathbin{\square} S) \sqsubseteq_{DT} (P' \mathbin{\square} S')$›
⟨*proof*⟩

lemmas *mono-Det-FD*[*simp*]= *mono-Det-FD*[*folded failure-divergence-refine-def*]

— Deterministic choice monotony doesn't hold for $\sqsubseteq_F$

lemma *mono-Ndet-F-left*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (P \sqinter S) \sqsubseteq_F Q$›
⟨*proof*⟩

lemma *mono-Ndet-D-left*[*simp*]: ‹$P \sqsubseteq_D Q \Longrightarrow (P \sqinter S) \sqsubseteq_D Q$›
⟨*proof*⟩

lemma *mono-Ndet-T-left*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (P \sqinter S) \sqsubseteq_T Q$›
⟨*proof*⟩

8.2 Monotonicity

lemma *mono-Ndet-F*[*simp*]: ‹$[\![P \sqsubseteq_F P'; S \sqsubseteq_F S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_F (P' \sqcap S')$›
⟨*proof*⟩

lemma *mono-Ndet-D*[*simp*]: ‹$[\![P \sqsubseteq_D P'; S \sqsubseteq_D S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_D (P' \sqcap S')$›
⟨*proof*⟩

lemma *mono-Ndet-T*[*simp*]: ‹$[\![P \sqsubseteq_T P'; S \sqsubseteq_T S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_T (P' \sqcap S')$›
⟨*proof*⟩

corollary *mono-Ndet-DT*[*simp*]: ‹$[\![P \sqsubseteq_{DT} P'; S \sqsubseteq_{DT} S']\!] \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} (P' \sqcap S')$›
⟨*proof*⟩

lemmas *mono-Ndet-FD*[*simp*]= *mono-Ndet-FD*[*folded failure-divergence-refine-def*]

corollary *mono-Ndet-DT-left*[*simp*]: ‹$P \sqsubseteq_{DT} Q \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} Q$›
and *mono-Ndet-F-right*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (S \sqcap P) \sqsubseteq_F Q$›
and *mono-Ndet-D-right*[*simp*]: ‹$P \sqsubseteq_D Q \Longrightarrow (S \sqcap P) \sqsubseteq_D Q$›
and *mono-Ndet-T-right*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (S \sqcap P) \sqsubseteq_T Q$›
and *mono-Ndet-DT-right*[*simp*]: ‹$P \sqsubseteq_{DT} Q \Longrightarrow (S \sqcap P) \sqsubseteq_{DT} Q$›
⟨*proof*⟩

lemmas *mono-Ndet-FD-left*[*simp*] = *mono-Ndet-FD-left*[*folded failure-divergence-refine-def*]
and *mono-Ndet-FD-right*[*simp*] = *mono-Ndet-FD-right*[*folded failure-divergence-refine-def*]

lemma *mono-Ndet-Det-FD*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_{FD} (P \mathbin{\square} S)$›
⟨*proof*⟩

corollary *mono-Ndet-Det-F*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_F (P \mathbin{\square} S)$›
and *mono-Ndet-Det-D*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_D (P \mathbin{\square} S)$›
and *mono-Ndet-Det-T*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_T (P \mathbin{\square} S)$›
and *mono-Ndet-Det-DT*[*simp*]: ‹$(P \sqcap S) \sqsubseteq_{DT} (P \mathbin{\square} S)$›
⟨*proof*⟩

lemma *mono-Seq-F-right*[*simp*]: ‹$S \sqsubseteq_F S' \Longrightarrow (P ; S) \sqsubseteq_F (P ; S')$›
⟨*proof*⟩

lemma *mono-Seq-D-right*[*simp*]: ‹$S \sqsubseteq_D S' \Longrightarrow (P ; S) \sqsubseteq_D (P ; S')$›
⟨*proof*⟩

lemma *mono-Seq-T-right*[*simp*]: ‹$S \sqsubseteq_T S' \Longrightarrow (P ; S) \sqsubseteq_T (P ; S')$›
⟨*proof*⟩

corollary *mono-Seq-DT-right*[*simp*]: ‹$S \sqsubseteq_{DT} S' \Longrightarrow (P \mathbin{;} S) \sqsubseteq_{DT} (P \mathbin{;} S')$›
⟨*proof*⟩

lemma *mono-Seq-DT-left*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow (P \mathbin{;} S) \sqsubseteq_{DT} (P' \mathbin{;} S)$›
⟨*proof*⟩

corollary *mono-Seq-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow S \sqsubseteq_{DT} S' \Longrightarrow (P \mathbin{;} S) \sqsubseteq_{DT} (P' \mathbin{;} S')$›
⟨*proof*⟩

lemmas *mono-Seq-FD*[*simp*]= *mono-Seq-FD*[*folded failure-divergence-refine-def*]

lemma *mono-Mprefix-F*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_F Mprefix\ A\ Q$›
⟨*proof*⟩

lemma *mono-Mprefix-D*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_D Mprefix\ A\ Q$›
⟨*proof*⟩

lemma *mono-Mprefix-T*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_T Mprefix\ A\ Q$›
⟨*proof*⟩

corollary *mono-Mprefix-DT*[*simp*]: ‹$\forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq_{DT} Mprefix\ A\ Q$›
⟨*proof*⟩

lemmas *mono-Mprefix-FD*[*simp*] = *mono-Mprefix-FD*[*folded failure-divergence-refine-def*]

lemma *mono-Mprefix-DT-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_{DT} Mprefix\ A\ P$›

⟨*proof*⟩

corollary *mono-Mprefix-D-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_D Mprefix\ A\ P$›
and *mono-Mprefix-T-set*[*simp*]: ‹$A \subseteq B \Longrightarrow Mprefix\ B\ P \sqsubseteq_T Mprefix\ A\ P$›
⟨*proof*⟩

lemma *mono-Hiding-F*[*simp*]: ‹$P \sqsubseteq_F Q \Longrightarrow (P \setminus A) \sqsubseteq_F (Q \setminus A)$›
⟨*proof*⟩

lemma *mono-Hiding-T*[*simp*]: ‹$P \sqsubseteq_T Q \Longrightarrow (P \setminus A) \sqsubseteq_T (Q \setminus A)$›
⟨*proof*⟩

lemma *mono-Hiding-DT*[*simp*]: ‹$P \sqsubseteq_{DT} Q \Longrightarrow (P \setminus A) \sqsubseteq_{DT} (Q \setminus A)$›
⟨*proof*⟩

lemmas *mono-Hiding-FD*[*simp*] = *mono-Hiding-FD*[*folded failure-divergence-refine-def*]

— Obviously, Hide monotony doesn't hold for $\sqsubseteq_D$

lemma *mono-Sync-DT*[*simp*]: ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow (P \llbracket A \rrbracket Q) \sqsubseteq_{DT} (P' \llbracket A \rrbracket Q')$›
⟨*proof*⟩

lemmas *mono-Sync-FD*[*simp*] = *mono-Sync-FD*[*folded failure-divergence-refine-def*]

— Synchronization monotony doesn't hold for $\sqsubseteq_F$, $\sqsubseteq_D$ and $\sqsubseteq_T$

lemma *mono-Mndetprefix-F*[*simp*]: ‹$\forall x{\in}A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_F$ *Mndetprefix A Q*›
⟨*proof*⟩

lemma *mono-Mndetprefix-D*[*simp*]: ‹$\forall x{\in}A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_D$ *Mndetprefix A Q*›
⟨*proof*⟩

lemma *mono-Mndetprefix-T*[*simp*]: ‹$\forall x{\in}A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_T$ *Mndetprefix A Q*›
⟨*proof*⟩

corollary *mono-Mndetprefix-DT*[*simp*]: ‹$\forall x{\in}A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{DT}$ *Mndetprefix A Q*›
⟨*proof*⟩

lemmas *mono-Mndetprefix-FD*[*simp*] = *mono-Mndetprefix-FD*[*folded failure-divergence-refine-def*]

lemmas *mono-Mndetprefix-FD-set*[*simp*] = *Mndetprefix-FD-subset*[*folded failure-divergence-refine-def*]

corollary *mono-Mndetprefix-F-set*[*simp*] : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_F$ *Mndetprefix A P*›
and *mono-Mndetprefix-D-set*[*simp*] : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_D$ *Mndetprefix A P*›
and *mono-Mndetprefix-T-set*[*simp*] : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_T$ *Mndetprefix A P*›
and *mono-Mndetprefix-DT-set*[*simp*]: ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{DT}$ *Mndetprefix A P*›
⟨*proof*⟩

lemmas *Mprefix-refines-Mndetprefix-FD*[*simp*] =
Mprefix-refines-Mndetprefix[*folded failure-divergence-refine-def*]

corollary *Mprefix-refines-Mndetprefix-F*[*simp*] : ‹*Mndetprefix A P* $\sqsubseteq_F$ *Mprefix A*

$P\rangle$
and *Mprefix-refines-Mndetprefix-D*[*simp*] : $\langle$*Mndetprefix A P* $\sqsubseteq_D$ *Mprefix A P*$\rangle$
and *Mprefix-refines-Mndetprefix-T*[*simp*] : $\langle$*Mndetprefix A P* $\sqsubseteq_T$ *Mprefix A P*$\rangle$
and *Mprefix-refines-Mndetprefix-DT*[*simp*]: $\langle$*Mndetprefix A P* $\sqsubseteq_{DT}$ *Mprefix A P*$\rangle$
⟨*proof*⟩

lemma *mono-read-FD*[*simp*, *elim*]: $(\bigwedge x.\ P\ x \sqsubseteq_{FD} Q\ x) \Longrightarrow (c?x \rightarrow (P\ x)) \sqsubseteq_{FD} (c?x \rightarrow (Q\ x))$
⟨*proof*⟩

lemma *mono-write-FD*[*simp*, *elim*]: $(P \sqsubseteq_{FD} Q) \Longrightarrow (c!x \rightarrow P) \sqsubseteq_{FD} (c!x \rightarrow Q)$
⟨*proof*⟩

lemma *mono-write0-FD*[*simp*, *elim*]: $P \sqsubseteq_{FD} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{FD} (a \rightarrow Q)$
⟨*proof*⟩

lemma *mono-read-DT*[*simp*, *elim*]: $(\bigwedge x.\ P\ x \sqsubseteq_{DT} Q\ x) \Longrightarrow (c?x \rightarrow (P\ x)) \sqsubseteq_{DT} (c?x \rightarrow (Q\ x))$
⟨*proof*⟩

lemma *mono-write-DT*[*simp*, *elim*]: $(P \sqsubseteq_{DT} Q) \Longrightarrow (c!x \rightarrow P) \sqsubseteq_{DT} (c!x \rightarrow Q)$
⟨*proof*⟩

lemma *mono-write0-DT*[*simp*, *elim*]: $P \sqsubseteq_{DT} Q \Longrightarrow (a \rightarrow P) \sqsubseteq_{DT} (a \rightarrow Q)$
⟨*proof*⟩

lemma *mono-Renaming-D*: $\langle P \sqsubseteq_D Q \Longrightarrow Renaming\ P\ f \sqsubseteq_D Renaming\ Q\ f\rangle$
⟨*proof*⟩

lemma *mono-Renaming-FD*: $\langle P \sqsubseteq_{FD} Q \Longrightarrow Renaming\ P\ f \sqsubseteq_{FD} Renaming\ Q\ f\rangle$
⟨*proof*⟩

lemma *mono-Renaming-DT*: $\langle P \sqsubseteq_{DT} Q \Longrightarrow Renaming\ P\ f \sqsubseteq_{DT} Renaming\ Q\ f\rangle$
⟨*proof*⟩

Some new and very useful results

lemma *Ndet-is-STOP-iff*: ‹$P \sqcap Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP$›
⟨*proof*⟩

lemma *Det-is-STOP-iff*: ‹$P \mathbin{\square} Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP$›
⟨*proof*⟩

lemma *Det-is-BOT-iff*: ‹$P \mathbin{\square} Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
⟨*proof*⟩

lemma *Ndet-is-BOT-iff*: ‹$P \sqcap Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
⟨*proof*⟩

lemma *Sync-is-BOT-iff*: ‹$P \llbracket S \rrbracket Q = \bot \longleftrightarrow P = \bot \vee Q = \bot$›
⟨*proof*⟩

lemma *STOP-neq-BOT*: ‹$STOP \neq \bot$›
⟨*proof*⟩

lemma *SKIP-neq-BOT*: ‹$SKIP \neq \bot$›
⟨*proof*⟩

lemma *Mprefix-neq-BOT*: ‹$Mprefix\ A\ P \neq \bot$›
⟨*proof*⟩

lemma *Mndetprefix-neq-BOT*: ‹$Mndetprefix\ A\ P \neq \bot$›
⟨*proof*⟩

lemma *STOP-T-iff*: ‹$STOP \sqsubseteq_T P \longleftrightarrow P = STOP$›
⟨*proof*⟩

lemma *STOP-F-iff*: ‹$STOP \sqsubseteq_F P \longleftrightarrow P = STOP$›
⟨*proof*⟩

lemma *STOP-FD-iff*: ‹$STOP \sqsubseteq_{FD} P \longleftrightarrow P = STOP$›
⟨*proof*⟩

lemma *SKIP-FD-iff*: ‹$SKIP \sqsubseteq_{FD} P \longleftrightarrow P = SKIP$›
⟨*proof*⟩

lemma *SKIP-F-iff*: ‹$SKIP \sqsubseteq_F P \longleftrightarrow P = SKIP$›
⟨*proof*⟩

lemma *Seq-is-SKIP-iff*: ‹$P \mathbin{;} Q = SKIP \longleftrightarrow P = SKIP \wedge Q = SKIP$›

⟨*proof*⟩

lemma *cont-process-rec*: ‹$P = (\mu\ X.\ f\ X) \Longrightarrow cont\ f \Longrightarrow P = f\ P$› ⟨*proof*⟩

end

theory *Assertions*
imports *CSP*
begin

default-sort *type*

8.3 CSP Assertions

definition $DF :: 'a\ set \Rightarrow 'a\ process$
where $DF\ A \equiv \mu\ X.\ \sqcap\ x \in A \rightarrow X$

lemma *DF-unfold* : $DF\ A = (\sqcap\ z \in A \rightarrow DF\ A)$
⟨*proof*⟩

definition *deadlock-free* :: $'a\ process \Rightarrow bool$
where $deadlock\text{-}free\ P \equiv DF\ UNIV \sqsubseteq_{FD} P$

definition $DF_{SKIP} :: 'a\ set \Rightarrow 'a\ process$
where $DF_{SKIP}\ A \equiv \mu\ X.\ ((\sqcap\ x \in A \rightarrow X) \sqcap SKIP)$

definition *deadlock-free-v2* :: $'a\ process \Rightarrow bool$
where $deadlock\text{-}free\text{-}v2\ P \equiv DF_{SKIP}\ UNIV \sqsubseteq_{F} P$

8.4 Some deadlock freeness laws

lemma *DF-subset*: $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow DF\ B \sqsubseteq_{FD} DF\ A$
⟨*proof*⟩

lemma *DF-Univ-freeness*: $A \neq \{\} \Longrightarrow (DF\ A) \sqsubseteq_{FD} P \Longrightarrow deadlock\text{-}free\ P$
⟨*proof*⟩

lemma *deadlock-free-Ndet*: $deadlock\text{-}free\ P \Longrightarrow deadlock\text{-}free\ Q \Longrightarrow deadlock\text{-}free\ (P \sqcap Q)$
⟨*proof*⟩

8.5 Preliminaries

lemma DF_{SKIP}-*unfold* : $DF_{SKIP}\ A = ((\sqcap\ z \in A \rightarrow DF_{SKIP}\ A) \sqcap SKIP)$
⟨*proof*⟩

8.6 Deadlock Free

lemma *div-free-*DF_{SKIP}: $\mathcal{D}(DF_{SKIP}\ A) = \{\}$
⟨*proof*⟩

lemma *div-free-DF*: $\mathcal{D}(DF\ A) = \{\}$
⟨*proof*⟩

lemma *deadlock-free-implies-div-free*: *deadlock-free* $P \Longrightarrow \mathcal{D}\ P = \{\}$
⟨*proof*⟩

8.7 Run

definition $RUN :: 'a\ set \Rightarrow 'a\ process$
where $RUN\ A \equiv \mu\ X.\ \Box\ x \in A \rightarrow X$

definition $CHAOS :: 'a\ set \Rightarrow 'a\ process$
where $CHAOS\ A \equiv \mu\ X.\ (STOP \sqcap (\Box\ x \in A \rightarrow X))$

definition *lifelock-free* :: *'a process* ⇒ *bool*
where *lifelock-free* $P \equiv CHAOS\ UNIV \sqsubseteq_{FD} P$

8.8 Reference processes and their unfolding rules

definition $CHAOS_{SKIP} :: 'a\ set \Rightarrow 'a\ process$
where $CHAOS_{SKIP}\ A \equiv \mu\ X.\ (SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow X))$

lemma *RUN-unfold* : $RUN\ A = (\Box\ z \in A \rightarrow RUN\ A)$
⟨*proof*⟩

lemma *CHAOS-unfold* : $CHAOS\ A = (STOP \sqcap (\Box\ z \in A \rightarrow CHAOS\ A))$
⟨*proof*⟩

lemma $CHAOS_{SKIP}$-*unfold*: $CHAOS_{SKIP}\ A \equiv SKIP \sqcap STOP \sqcap (\Box\ x \in A \rightarrow CHAOS_{SKIP}\ A)$
⟨*proof*⟩

8.9 Process events and reference processes events

definition *events-of* **where** *events-of* $P \equiv (\bigcup t{\in}\mathcal{T}\ P.\ \{a.\ ev\ a \in set\ t\})$

lemma *events-DF*: *events-of* $(DF\ A) = A$
⟨*proof*⟩

lemma *events-DF*$_{SKIP}$: *events-of* $(DF_{SKIP}\ A) = A$
⟨*proof*⟩

lemma *events-RUN*: *events-of* $(RUN\ A) = A$
⟨*proof*⟩

lemma *events-CHAOS*: *events-of* $(CHAOS\ A) = A$
⟨*proof*⟩

lemma *events-CHAOS*$_{SKIP}$: *events-of* $(CHAOS_{SKIP}\ A) = A$
⟨*proof*⟩

lemma *events-div*: $\mathcal{D}(P) \neq \{\} \Longrightarrow$ *events-of* $(P) = UNIV$
⟨*proof*⟩

lemma *DF*$_{SKIP}$*-subset-FD*: $A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow DF_{SKIP}\ B \sqsubseteq_{FD} DF_{SKIP}\ A$
⟨*proof*⟩

lemma *RUN-subset-DT*: $A \subseteq B \Longrightarrow RUN\ B \sqsubseteq_{DT} RUN\ A$
⟨*proof*⟩

lemma *CHAOS-subset-FD*: $A \subseteq B \Longrightarrow CHAOS\ B \sqsubseteq_{FD} CHAOS\ A$
⟨*proof*⟩

lemma *CHAOS*$_{SKIP}$*-subset-FD*: $A \subseteq B \Longrightarrow CHAOS_{SKIP}\ B \sqsubseteq_{FD} CHAOS_{SKIP}\ A$
⟨*proof*⟩

8.10 Relations between refinements on reference processes

lemma *CHAOS-has-all-tickFree-failures*:
tickFree a $\Longrightarrow \{x.\ ev\ x \in set\ a\} \subseteq A \Longrightarrow (a,b) \in \mathcal{F}\ (CHAOS\ A)$
⟨*proof*⟩

lemma *CHAOS*$_{SKIP}$*-has-all-failures*:
assumes *as*:(*events-of P*) $\subseteq A$
shows $CHAOS_{SKIP}\ A \sqsubseteq_F P$
⟨*proof*⟩

corollary *CHAOS*$_{SKIP}$*-has-all-failures-ev*: $CHAOS_{SKIP}$ (*events-of P*) $\sqsubseteq_F P$
and *CHAOS*$_{SKIP}$*-has-all-failures-Un*: $CHAOS_{SKIP}\ UNIV \sqsubseteq_F P$

⟨*proof*⟩

lemma *DF$_{SKIP}$-DF-refine-F*: $DF_{SKIP}\ A\ \sqsubseteq_F\ DF\ A$
⟨*proof*⟩

lemma *DF-RUN-refine-F*: $DF\ A\ \sqsubseteq_F\ RUN\ A$
⟨*proof*⟩

lemma *CHAOS-DF-refine-F*: $CHAOS\ A\ \sqsubseteq_F\ DF\ A$
⟨*proof*⟩

corollary *CHAOS$_{SKIP}$-CHAOS-refine-F*: $CHAOS_{SKIP}\ A\ \sqsubseteq_F\ CHAOS\ A$
and *CHAOS$_{SKIP}$-DF$_{SKIP}$-refine-F*: $CHAOS_{SKIP}\ A\ \sqsubseteq_F\ DF_{SKIP}\ A$
⟨*proof*⟩

lemma *div-free-CHAOS$_{SKIP}$*: $\mathcal{D}\ (CHAOS_{SKIP}\ A) = \{\}$
⟨*proof*⟩

lemma *div-free-CHAOS*: $\mathcal{D}(CHAOS\ A) = \{\}$
⟨*proof*⟩

lemma *div-free-RUN*: $\mathcal{D}(RUN\ A) = \{\}$
⟨*proof*⟩

corollary *DF$_{SKIP}$-DF-refine-FD*: $DF_{SKIP}\ A\ \sqsubseteq_{FD}\ DF\ A$
and *DF-RUN-refine-FD*: $DF\ A\ \sqsubseteq_{FD}\ RUN\ A$
and *CHAOS-DF-refine-FD*: $CHAOS\ A\ \sqsubseteq_{FD}\ DF\ A$
and *CHAOS$_{SKIP}$-CHAOS-refine-FD*: $CHAOS_{SKIP}\ A\ \sqsubseteq_{FD}\ CHAOS\ A$
and *CHAOS$_{SKIP}$-DF$_{SKIP}$-refine-FD*: $CHAOS_{SKIP}\ A\ \sqsubseteq_{FD}\ DF_{SKIP}\ A$
⟨*proof*⟩

lemma *traces-CHAOS-sub*: $\mathcal{T}(CHAOS\ A) \subseteq \{s.\ set\ s \subseteq ev\ `\ A\}$
⟨*proof*⟩

lemma *traces-RUN-sub*: $\{s.\ set\ s \subseteq ev\ `\ A\} \subseteq \mathcal{T}(RUN\ A)$
⟨*proof*⟩

corollary *RUN-all-tickfree-traces1*: $\mathcal{T}(RUN\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
and *DF-all-tickfree-traces1*: $\mathcal{T}(DF\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
and *CHAOS-all-tickfree-traces1*: $\mathcal{T}(CHAOS\ A) = \{s.\ set\ s \subseteq ev\ `\ A\}$
⟨*proof*⟩

corollary *RUN-all-tickfree-traces2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(RUN\ UNIV)$
and *DF-all-tickfree-traces2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(DF\ UNIV)$

and *CHAOS-all-tickfree-trace2*: $tickFree\ s \Longrightarrow s \in \mathcal{T}(CHAOS\ UNIV)$
⟨*proof*⟩

lemma *traces-CHAOS$_{SKIP}$-sub*: $\mathcal{T}(CHAOS_{SKIP}\ A) \subseteq \{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\}$
⟨*proof*⟩

lemma *traces-DF$_{SKIP}$-sub*:
$\{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\} \subseteq \mathcal{T}(DF_{SKIP}\ A::'a\ process)$
⟨*proof*⟩

corollary *DF$_{SKIP}$-all-front-tickfree-traces1*:
$\mathcal{T}(DF_{SKIP}\ A) = \{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\}$
and *CHAOS$_{SKIP}$-all-front-tickfree-traces1*:
$\mathcal{T}(CHAOS_{SKIP}\ A) = \{s.\ front\text{-}tickFree\ s \wedge set\ s \subseteq (ev\ `\ A \cup \{tick\})\}$
⟨*proof*⟩

corollary *DF$_{SKIP}$-all-front-tickfree-traces2*: $front\text{-}tickFree\ s \Longrightarrow s \in \mathcal{T}(DF_{SKIP}\ UNIV)$
and *CHAOS$_{SKIP}$-all-front-tickfree-traces2*: $front\text{-}tickFree\ s \Longrightarrow s \in \mathcal{T}(CHAOS_{SKIP}\ UNIV)$
⟨*proof*⟩

corollary *DF$_{SKIP}$-has-all-traces*: $DF_{SKIP}\ UNIV \sqsubseteq_T P$
and *CHAOS$_{SKIP}$-has-all-traces*: $CHAOS_{SKIP}\ UNIV \sqsubseteq_T P$
⟨*proof*⟩

lemma *deadlock-free-implies-non-terminating*:
$deadlock\text{-}free\ (P::'a\ process) \Longrightarrow \forall\, s{\in}\mathcal{T}\ P.\ tickFree\ s$
⟨*proof*⟩

lemma *deadlock-free-v2-is-right*:
$deadlock\text{-}free\text{-}v2\ (P::'a\ process) \longleftrightarrow (\forall\, s{\in}\mathcal{T}\ P.\ tickFree\ s \longrightarrow (s,\ UNIV::'a\ event\ set) \notin \mathcal{F}\ P)$
⟨*proof*⟩

lemma *deadlock-free-v2-implies-div-free*: $deadlock\text{-}free\text{-}v2\ P \Longrightarrow \mathcal{D}\ P = \{\}$
⟨*proof*⟩

corollary *deadlock-free-v2-FD*: $deadlock\text{-}free\text{-}v2\ P = DF_{SKIP}\ UNIV \sqsubseteq_{FD} P$
⟨*proof*⟩

lemma *all-events-refusal*:
$(s,\ \{tick\} \cup ev\ `\ (events\text{-}of\ P)) \in \mathcal{F}\ P \Longrightarrow (s,\ UNIV::'a\ event\ set) \in \mathcal{F}\ P$
⟨*proof*⟩

corollary *deadlock-free-v2-is-right-wrt-events*:
deadlock-free-v2 (*P*::*'a process*) ⟷
(∀ *s*∈$\mathcal{T}$ *P*. *tickFree s* ⟶ (*s*, {*tick*} ∪ *ev* ‘ (*events-of P*)) ∉ $\mathcal{F}$ *P*)
⟨*proof*⟩

lemma *deadlock-free-is-deadlock-free-v2*:
deadlock-free P ⟹ *deadlock-free-v2 P*
⟨*proof*⟩

8.11 *deadlock-free* and *deadlock-free-v2* with *SKIP* and *STOP*

lemma *deadlock-free-v2-SKIP*: *deadlock-free-v2 SKIP*
⟨*proof*⟩

lemma *non-deadlock-free-SKIP*: ‹¬ *deadlock-free SKIP*›
⟨*proof*⟩

lemma *non-deadlock-free-v2-STOP*: ‹¬ *deadlock-free-v2 STOP*›
⟨*proof*⟩

lemma *non-deadlock-free-STOP*: ‹¬ *deadlock-free STOP*›
⟨*proof*⟩

8.12 Non-terminating Runs

definition *non-terminating* :: *'a process* ⇒ *bool*
where *non-terminating P* ≡ *RUN UNIV* ⊑$_T$ *P*

corollary *non-terminating-refine-DF*: *non-terminating P* = *DF UNIV* ⊑$_T$ *P*
and *non-terminating-refine-CHAOS*: *non-terminating P* = *CHAOS UNIV* ⊑$_T$ *P*
⟨*proof*⟩

lemma *non-terminating-is-right*: *non-terminating* (*P*::*'a process*) ⟷ (∀ *s*∈$\mathcal{T}$ *P*. *tickFree s*)
⟨*proof*⟩

lemma *nonterminating-implies-div-free*: *non-terminating P* ⟹ $\mathcal{D}$ *P* = {}
⟨*proof*⟩

lemma *non-terminating-implies-F*: *non-terminating P* ⟹ *CHAOS UNIV* ⊑$_F$ *P*
⟨*proof*⟩

corollary *non-terminating-F*: *non-terminating P* = *CHAOS UNIV* ⊑$_F$ *P*
⟨*proof*⟩

corollary *non-terminating-FD*: *non-terminating P* = *CHAOS UNIV* ⊑$_{F\,D}$ *P*

⟨*proof*⟩

8.13 Lifelock Freeness

thm *lifelock-free-def*

corollary *lifelock-free-is-non-terminating*: *lifelock-free P* = *non-terminating P*
⟨*proof*⟩

lemma *div-free-divergence-refine*:
$\mathcal{D}\ P = \{\} \longleftrightarrow CHAOS_{SKIP}\ UNIV \sqsubseteq_D P$
$\mathcal{D}\ P = \{\} \longleftrightarrow CHAOS\ UNIV \sqsubseteq_D P$
$\mathcal{D}\ P = \{\} \longleftrightarrow RUN\ UNIV \sqsubseteq_D P$
$\mathcal{D}\ P = \{\} \longleftrightarrow DF_{SKIP}\ UNIV \sqsubseteq_D P$
$\mathcal{D}\ P = \{\} \longleftrightarrow DF\ UNIV \sqsubseteq_D P$
⟨*proof*⟩

definition *lifelock-free-v2* :: *'a process* ⇒ *bool*
where *lifelock-free-v2 P* ≡ $CHAOS_{SKIP}\ UNIV \sqsubseteq_{FD} P$

lemma *div-free-is-lifelock-free-v2*: *lifelock-free-v2 P* ⟷ $\mathcal{D}\ P = \{\}$
⟨*proof*⟩

lemma *lifelock-free-is-lifelock-free-v2*: *lifelock-free P* ⟹ *lifelock-free-v2 P*
⟨*proof*⟩

corollary *deadlock-free-v2-is-lifelock-free-v2*: *deadlock-free-v2 P* ⟹ *lifelock-free-v2 P*
⟨*proof*⟩

8.14 New laws

lemma *non-terminating-Seq*:
non-terminating P ⟹ (*P* ; *Q*) = *P*
⟨*proof*⟩

lemma *non-terminating-Sync*:
non-terminating P ⟹ *lifelock-free-v2 Q* ⟹ *non-terminating* (*P* ⟦*A*⟧ *Q*)
⟨*proof*⟩

lemmas *non-terminating-Par* = *non-terminating-Sync*[**where** *A* = ‹*UNIV*›]
and *non-terminating-Inter* = *non-terminating-Sync*[**where** *A* = ‹{}›]

end

Chapter 9

Conclusion

9.1 Related Work

As mentioned earlier, this work has its very ancient roots in a first formalization of A. Camilieri in the early 90s in HOL. This work was reformulated and substantially extended in HOL-CSP 1.0 published in 1997. In 2005, Roggenbach and Isobe published CSP-Prover, a formal theory of a (fragment of) the Failures model of CSP. This work led to a couple of publications culminating in [6]; emphasis was put on actually completing the CSP theory up to the point where it is sufficiently tactically supported to serve as a kind of tool. This theory is still maintained and last releases (the latest one was released on 18 February 2019) can be found under https://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html. This theory represents the first half of Roscoes theory of a Failures/Divergence model, i.e. the Failures part. More recently, Pasquale Noce [9, 11, 10] developed a theory of non-interference notions based on an abstract denotational model fragment of the Failure/Divergence Model of CSP (without continuity and algebraic laws); this theory could probably be rebuilt on top of our work.

The present work could be another, more "classic" foundation of test-generation techniques of this kind paving the way to an interaction with FDR and its possibility to generate labelled transition systems as output that could drive specialized tactics in HOL-CSP 2.0.

9.2 Lessons learned

We have ported a first formalization in Isabelle/HOL on the Failure/Divergence model of CSP, done with Isabelle93-7 in 1997, to a modern Isabelle version. Particularly, we use the modern declarative proof style available in Isabelle/Isar instead of imperative proof one, the latter being used in the old version. On the one hand, it is worth noting that some of the old theories

still have a surprisingly high value: Actually it took time to develop the right granularity of abstraction in lemmas, which is thus still quite helpful and valuable to reconstruct the theory in the new version. If a substantially large body of lemmas is available, the degree of automation tends to increase. On the other hand, redevelopment from scratch is unavoidable in parts where basic libraries change. For example, this was a necessary consequence of our decision to base HOL-CSP 2.0 on HOLCF instead of continuing the development of an older fixed-point theory; nearly all continuity proofs had to be redeveloped. Moreover, a fresh look on old proof-obligations may incite unexpected generalizations and some newly proved lemmas that cannot be constructed in the old version even with several attempts. The influence of the chosen strategy (from scratch or refactoring) on the proof length is inconclusive.

Note that our data does not allow to make a prediction on the length of a porting project — the effort was distributed over a too long period and performed by a team with initially very different knowledge about CSP and interactive theorem proving.

9.3 A Summary on New Results

Compared to the original version of HOL-CSP 1.0, the present theory is complete relative to Roscoe's Book[12]. It contains a number of new theorems and some interesting (and unexpected) generalizations:

1. $?P \sqsubseteq ?Q \Longrightarrow ?P \setminus ?A \sqsubseteq ?Q \setminus ?A$ is now also valid for the infinite case (arbitrary hide-set A).

2. $P \setminus A \cup B = P \setminus A \setminus B$ is true for *finite* A (see *finite* $?A \Longrightarrow ?P \setminus ?A \cup ?B = ?P \setminus ?A \setminus ?B$); this was not even proven in HOL-CSP 1.0 for the singleton case! It can be considered as the most complex theorem of this theory.

3. distribution laws of *Hiding* over *Sync* $[\![$*finite* $?A$; $?A \cap ?S = \{\}]\!] \Longrightarrow ?P \ [\![?S]\!] \ ?Q \setminus ?A = (?P \setminus ?A) \ [\![?S]\!] \ (?Q \setminus ?A)$; however, this works only in the finite case. A true monster proof.

4. distribution of *Mprefix* over *Sync* $[\![?A \cap ?S = \{\}$; $?A' \subseteq ?S$; $?B \cap ?S = \{\}$; $?B' \subseteq ?S]\!] \Longrightarrow$ *Mprefix* $(?A \cup ?A')\ ?P \ [\![?S]\!]$ *Mprefix* $(?B \cup ?B')\ ?Q = (\Box x{\in}?A \rightarrow ?P\ x \ [\![?S]\!]$ *Mprefix* $(?B \cup ?B')\ ?Q) \ \Box \ (\Box y{\in}?B \rightarrow$ *Mprefix* $(?A \cup ?A')\ ?P \ [\![?S]\!] \ ?Q\ y) \ \Box \ (\Box x{\in}?A' \cap ?B' \rightarrow ?P\ x \ [\![?S]\!] \ ?Q\ x)$ in the most generalized case. Also a true monster proof, but reworked using symmetries and abstractions to be more reasonable (and faster)

5. the synchronization operator is associative $?P \llbracket ?S \rrbracket ?Q \llbracket ?S \rrbracket ?T = ?P \llbracket ?S \rrbracket (?Q \llbracket ?S \rrbracket ?T)$. (In HOL-CSP 1.0, this had only be shown for special cases like $?P||?Q||?T = ?P||(?Q||?T)$).

6. the generalized non-deterministic choice operator — relevant for proofs of deadlock-freeness — has been added to the theory *Mndetprefix* $\equiv$ *map-fun id* (*map-fun id Abs-process*) ($\lambda A\ P$. *if* $A = \{\}$ *then Rep-process STOP else* ($\bigcup_{x \in A} \mathcal{F}\ (x \rightarrow P\ x)$, $\bigcup_{x \in A} \mathcal{D}\ (x \rightarrow P\ x)$)); it is proven monotone in the general case and continuous for the special case ($\bigwedge x$. *cont* $(?f\ x)) \Longrightarrow$ *cont* $(\lambda y. \sqcap z \in ?A \rightarrow\ ?f\ z\ y)$ relevant for the definition of the deadlock reference processes $DF\ ?A \equiv \mu\ x. \sqcap xa \in ?A \rightarrow\ x$ and $DF_{SKIP}\ ?A \equiv \mu\ x. (\sqcap xa \in ?A \rightarrow\ x) \sqcap SKIP$.

end

Chapter 10

Annex: Refinement Example with Buffer over infinite Alphabet

theory *CopyBuffer*
imports *CSP Assertions*
begin

10.1 Defining the Copy-Buffer Example

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

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

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*›

thm *SYSTEM-def*

10.2 The Standard Proof

10.2.1 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*]: $left\ x \notin SYN$
⟨*proof*⟩

lemma [*simp*]: $right\ x \notin SYN$
⟨*proof*⟩

lemma [*simp*]: $ack \in SYN$
⟨*proof*⟩

lemma [*simp*]: $mid\ x \in SYN$
⟨*proof*⟩

lemma [*simp*]: *inj mid*
⟨*proof*⟩

lemma *finite* $(SYN :: 'a\ channel\ set) \Longrightarrow$ *finite* $\{(t::'a).\ True\}$
⟨*proof*⟩

10.2.2 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*:
$COPY = left?x \rightarrow right!x \rightarrow COPY$
⟨*proof*⟩

lemma *SEND-rec*:
$SEND = left?x \rightarrow mid!x \rightarrow (ack \rightarrow SEND)$
⟨*proof*⟩

lemma *REC-rec*:
$REC = mid?x \rightarrow right!x \rightarrow (ack \rightarrow REC)$
⟨*proof*⟩

10.2.3 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* : $COPY \sqsubseteq_{FD} SYSTEM$
⟨*proof*⟩

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

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

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

10.2.4 Deadlock Freeness Proof

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

lemma *DF-COPY* : $(DF\ (range\ left \cup range\ right)) \sqsubseteq_{FD} COPY$
⟨*proof*⟩

10.3 An Alternative Approach: Using the fixrec-Package

10.3.1 Channels and Synchronisation Sets

As before.

10.3.2 Process Definitions via fixrec-Package

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

thm *COPY'-rec*
definition $SYSTEM' :: ('a\ channel)\ process$
where $\langle SYSTEM' \equiv ((SEND' \llbracket SYN \rrbracket REC') \setminus SYN)\rangle$

10.3.3 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.

thm *COPY′-SEND′-REC′.induct*
lemma *impl-refines-spec′* : $(COPY'::'a\ channel\ process) \sqsubseteq_{FD} SYSTEM'$
⟨*proof*⟩

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

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

end

Bibliography

[1] P. Broadfoot and B. Roscoe. Tutorial on fdr and its applications. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, pages 322–322, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[2] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[3] A. J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer London.

[4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, July 1993.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[6] Y. Isobe and M. Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.

[7] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. Uniform workbench — universelle entwicklungsumgebung für formale methoden. Technical report, Technischer Bericht 8/95, Univ. Bremen, 1995. http://www.informatik.uni-bremen.de/~uniform.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[9] P. Noce. Noninterference security in communicating sequential processes. *Archive of Formal Proofs*, May 2014. http://isa-afp.org/entries/Noninterference_CSP.html, Formal proof development.

[10] P. Noce. Conservation of csp noninterference security under concurrent composition. *Archive of Formal Proofs*, June 2016. http://isa-afp.org/entries/Noninterference_Concurrent_Composition.html, Formal proof development.

[11] P. Noce. Conservation of csp noninterference security under sequential composition. *Archive of Formal Proofs*, Apr. 2016. http://isa-afp.org/entries/Noninterference_Sequential_Composition.html, Formal proof development.

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

[13] 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.