HOL-CSP Version 2.0

Benoît Ballenghien Safouan Taha Burkhart Wolff
Lina Ye

March 19, 2025

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 Brookes "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.

Actually, from the Isabelle-2025 version on, the theory has been extended in two ways:

- new operators known from Roscoe's books had been formally integrated (generalized non deterministic choice, sliding choice, etc.)
- the classical constant tick (✓) of the CSP theory has been replaced by a parameterized version carrying a kind of return value. It turns out that this is a very natural extension of the original setting. Generalizations of the sequential composition and synchronization product that fully enjoy this feature will be added in future versions.

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 paradigmatic 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 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

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

and

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

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

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

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. For this reason, the refinement notations $P \sqsubseteq_{FD} P'$, $P \sqsubseteq_{F} P'$, $P \sqsubseteq_{D} P'$, $P \sqsubseteq_{T} P'$, $P \sqsubseteq_{DT} P'$ have been introduced to be notationally closer to Roscoe's book.

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 preservation of `is_process` for this operator). In this new version, the proof is required immediately after the definition of the operator since we use **lift-definition** instead of definition
- The proof of the projections $\mathcal{T}$ and $\mathcal{F}$ and $\mathcal{D}$ 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 not only contains the "Laws" of CSP, i.e. the derived algebraic equalities, but also monotonicities rules, allowing to reason abstractly over CSP processes. The overall dependency graph is shown in Figure 1.1.

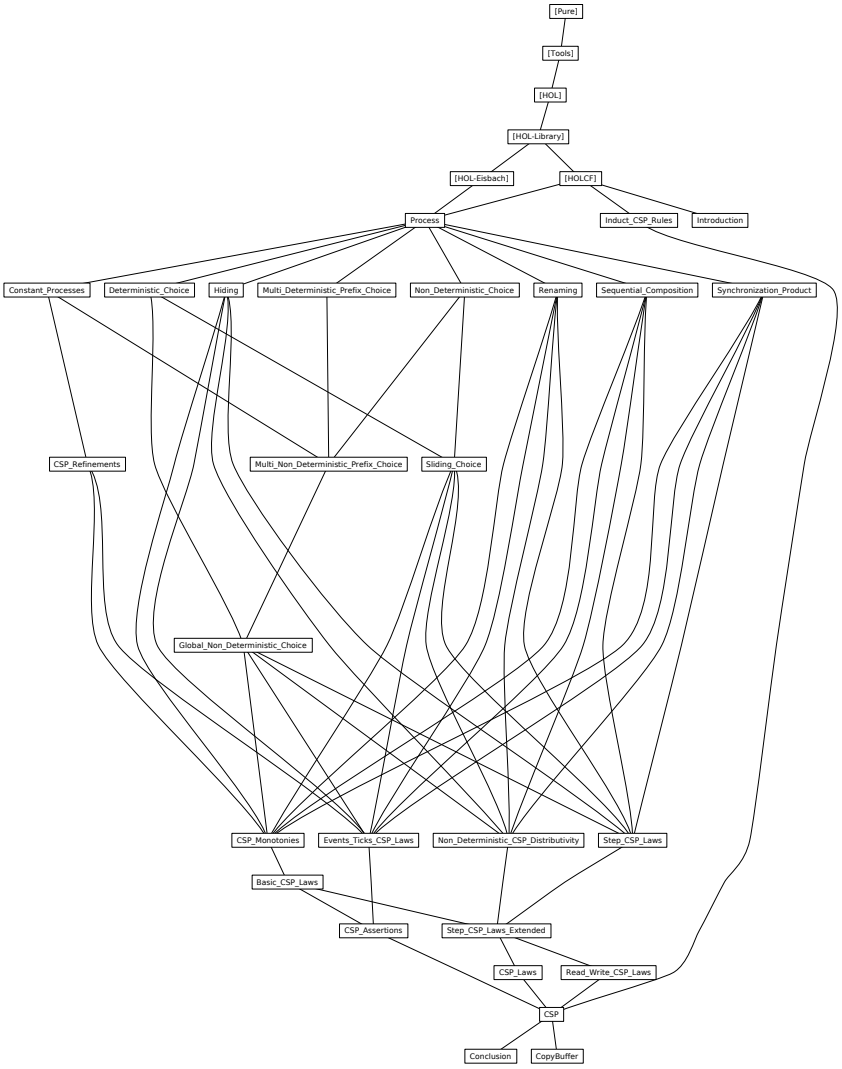

Figure 1.1: The HOL-CSP 2.0 Theory Graph

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.

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.)

From the Isabelle-2025 version on, the classical constant tick (✓) of the CSP theory has been replaced by a parameterized version carrying a kind of return value.

datatype $('a, 'r)\ event_{ptick} =$
  is-ev : *ev* (*of-ev* : $'a$)
| *is-tick* : *tick* (*of-tick* : $'r$) (‹✓'(-')›)

This type $('a, 'r)\ event_{ptick}$ is of course isomorphic to the sum type $'a + 'r$.

"ptick" stands for parameterized tick, and we introduce the type synonym for the classical process event type.

type-synonym $'a\ event = ‹('a, unit)\ event_{ptick}›$

abbreviation *tick-unit* :: ‹'*a event*› (‹✓›) **where** ‹✓ ≡ ✓(())›

definition *sum-of-event*$_{ptick}$ :: ‹('*a*, '*r*) *event*$_{ptick}$ ⇒ '*a* + '*r*›
where ‹*sum-of-event*$_{ptick}$ *e* ≡ *case e of ev a* ⇒ *Inl a* | ✓(*r*) ⇒ *Inr r*›

definition *event*$_{ptick}$*-of-sum* :: ‹'*a* + '*r* ⇒ ('*a*, '*r*) *event*$_{ptick}$›
where ‹*event*$_{ptick}$*-of-sum s* ≡ *case s of Inl a* ⇒ *ev a* | *Inr r* ⇒ ✓(*r*)›

lemma *type-definition-event*$_{ptick}$: ‹*type-definition sum-of-event*$_{ptick}$ *event*$_{ptick}$*-of-sum UNIV*›
⟨*proof*⟩

setup-lifting *type-definition-event*$_{ptick}$

lemma *range-tick-Un-range-ev-is-UNIV* [*simp*] : ‹*range tick* ∪ *range ev* = *UNIV*›
⟨*proof*⟩

The generalization is done in a very straightforward way: the old version is recovered by considering ('*a*, *unit*) *event*$_{ptick}$.

lemma *not-is-ev* [*simp*] : ‹¬ *is-ev e* ⟷ *is-tick e*›
and *not-is-tick* [*simp*] : ‹¬ *is-tick e* ⟷ *is-ev e*›
⟨*proof*⟩

type-synonym ('*a*, '*r*) *trace*$_{ptick}$ = ‹('*a*, '*r*) *event*$_{ptick}$ *list*›

We recover the classical version with *unit*.

type-synonym '*a trace* = ‹('*a*, *unit*) *trace*$_{ptick}$›

We chose as standard ordering on traces the prefix ordering.

Some facts on the prefix ordering.

lemma *nil-le* [*simp*]: ‹[] ≤ *s*›
and *nil-le2* [*simp*]: ‹*s* ≤ [] ⟷ *s* = []›
and *nil-less* [*simp*]: ‹¬ *t* < []›
and *nil-less2* [*simp*]: ‹[] < *t* @ [*a*]›
and *less-self* [*simp*]: ‹*t* < *t* @ [*a*]›
and *le-cons* [*simp*]: ‹*a* # *s* ≤ *a* # *t* ⟷ *s* ≤ *t*›
and *le-append* [*simp*]: ‹*b* @ *s* ≤ *b* @ *t* ⟷ *s* ≤ *t*›
and *less-cons* [*simp*]: ‹*a* # *s* < *a* # *t* ⟷ *s* < *t*›
and *less-append*[*simp*]: ‹*b* @ *s* < *b* @ *t* ⟷ *s* < *t*›

and *le-length-mono*: ‹*s* ≤ *t* ⟹ *length s* ≤ *length t*›
and *less-length-mono*: ‹*s* < *t* ⟹ *length s* < *length t*›
and *le-tail*: ‹*s* ≠ [] ⟹ *s* ≤ *t* ⟹ *tl s* ≤ *tl t*›
and *less-tail*: ‹*s* ≠ [] ⟹ *s* < *t* ⟹ *tl s* < *tl t*›
⟨*proof*⟩

lemma *le-same-imp-eq-or-less*: ‹$(s :: 'a\ list) \leq u \Longrightarrow t \leq u \Longrightarrow t = s \vee s < t \vee t < s$›
⟨*proof*⟩

lemma *append-eq-first-pref-spec*: ‹$s\ @\ t = r\ @\ [x] \Longrightarrow t \neq [] \Longrightarrow s \leq r$›
⟨*proof*⟩

lemma *prefixes-fin*: ‹*finite* $\{t.\ t \leq s\} \wedge$ *card* $\{t.\ t \leq s\} =$ *Suc* (*length* $s)$›
⟨*proof*⟩

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

lemma *suffixes-fin*: ‹*finite* $\{t.\ \exists\, t1.\ s = t1\ @\ t\}$›
⟨*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* :: ‹$('a, 'r)\ trace_{ptick} \Rightarrow bool$› (‹*tF*›)
where ‹$tF\ s \equiv range\ tick \cap set\ s = \{\}$›

definition *front-tickFree* :: ‹$('a, 'r)\ trace_{ptick} \Rightarrow bool$› (‹*ftF*›)
where ‹$ftF\ s \equiv s = [] \vee tickFree\ (tl\ (rev\ s))$›

lemma *tickFree-Nil* [*simp*] : ‹$tF\ []$›
and *tickFree-Cons-iff* [*simp*] : ‹$tF\ (a\ \#\ t) \longleftrightarrow is\text{-}ev\ a \wedge tF\ t$›
and *tickFree-append-iff* [*simp*] : ‹$tF\ (s\ @\ t) \longleftrightarrow tF\ s \wedge tF\ t$›
and *tickFree-rev-iff* [*simp*] : ‹$tF\ (rev\ t) \longleftrightarrow tF\ t$›
and *non-tickFree-tick* [*simp*] : ‹$\neg\ tF\ [✓(r)]$›
⟨*proof*⟩

lemma *tickFree-iff-is-map-ev* : ‹$tF\ t \longleftrightarrow (\exists\, u.\ t = map\ ev\ u)$›
⟨*proof*⟩

lemma *front-tickFree-Nil* [*simp*] : ‹$ftF\ []$›
and *front-tickFree-single*[*simp*] : ‹$ftF\ [a]$›
⟨*proof*⟩

lemma *tickFree-tl* : ‹$tF\ s \Longrightarrow tF\ (tl\ s)$›
⟨*proof*⟩

lemma *non-tickFree-imp-not-Nil*: ‹$\neg\ tF\ s \Longrightarrow s \neq []$›

⟨*proof*⟩

lemma *tickFree-butlast*: ‹*tF s* ⟷ *tF* (*butlast s*) ∧ (*s* ≠ [] ⟶ *is-ev* (*last s*))›
⟨*proof*⟩

lemma *front-tickFree-iff-tickFree-butlast*: ‹*ftF s* ⟷ *tF* (*butlast s*)›
⟨*proof*⟩

lemma *front-tickFree-Cons-iff*: ‹*ftF* (*a* # *s*) ⟷ *s* = [] ∨ *is-ev a* ∧ *ftF s*›
⟨*proof*⟩

lemma *front-tickFree-append-iff*:
‹*ftF* (*s* @ *t*) ⟷ (*if t* = [] *then ftF s else tF s* ∧ *ftF t*)›
⟨*proof*⟩

lemma *tickFree-imp-front-tickFree* [*simp*] : ‹*tF s* ⟹ *ftF s*›
⟨*proof*⟩

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

lemma *nonTickFree-n-frontTickFree*: ‹¬ *tF s* ⟹ *ftF s* ⟹ ∃ *t r*. *s* = *t* @ [✓(*r*)]›
⟨*proof*⟩

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

lemma *front-tickFree-append*: ‹*tF s* ⟹ *ftF t* ⟹ *ftF* (*s* @ *t*)›
⟨*proof*⟩

lemma *tickFree-imp-front-tickFree-snoc*: ‹*tF s* ⟹ *ftF* (*s* @ [*a*])›
⟨*proof*⟩

lemma *front-tickFree-nonempty-append-imp*: ‹*ftF* (*t* @ *r*) ⟹ *r* ≠ [] ⟹ *tF t* ∧ *ftF r*›
⟨*proof*⟩

lemma *tickFree-map-ev* [*simp*] : ‹*tF* (*map ev t*)›
⟨*proof*⟩

lemma *tickFree-map-tick-iff* [*simp*] : ‹*tF* (*map tick t*) ⟷ *t* = []›
⟨*proof*⟩

lemma *front-tickFree-map-tick-iff* [*simp*] : ‹*ftF* (*map tick t*) ⟷ *t* = [] ∨ (∃ *r*. *t* = [*r*])›
⟨*proof*⟩

lemma *tickFree-map-ev-comp* [*simp*] : ‹*tF* (*map* (*ev* ∘ *f*) *t*)›

⟨*proof*⟩

lemma *tickFree-map-tick-comp-iff* [*simp*] : ‹*tF* (*map* (*tick* ∘ *f*) *t*) ⟷ *t* = []›
⟨*proof*⟩

lemma *front-tickFree-map-tick-comp-iff* [*simp*] : ‹*ftF* (*map* (*tick* ∘ *f*) *t*) ⟷ *t* = [] ∨ (∃ *r*. *t* = [*r*])›
⟨*proof*⟩

2.2 Basic Types, Traces, Failures and Divergences

type-synonym ('*a*, '*r*) *refusal*$_{ptick}$ = ‹('*a*, '*r*) *event*$_{ptick}$ *set*›
type-synonym '*a refusal* = ‹('*a*, *unit*) *refusal*$_{ptick}$›
type-synonym ('*a*, '*r*) *failure*$_{ptick}$ = ‹('*a*, '*r*) *trace*$_{ptick}$ × ('*a*, '*r*) *refusal*$_{ptick}$›
type-synonym '*a failure* = ‹('*a*, *unit*) *failure*$_{ptick}$›
type-synonym ('*a*, '*r*) *divergence*$_{ptick}$ = ‹('*a*, '*r*) *trace*$_{ptick}$›
type-synonym '*a divergence* = ‹('*a*, *unit*) *divergence*$_{ptick}$›
type-synonym ('*a*, '*r*) *process*$_0$ = ‹('*a*, '*r*) *failure*$_{ptick}$ *set* × ('*a*, '*r*) *divergence*$_{ptick}$ *set*›

definition *FAILURES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *failure*$_{ptick}$ *set*›
where ‹*FAILURES P* ≡ *fst P*›

definition *TRACES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *trace*$_{ptick}$ *set*›
where ‹*TRACES P* ≡ {*tr*. ∃ *ref*. (*tr*, *ref*) ∈ *FAILURES P*}›

definition *DIVERGENCES* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *divergence*$_{ptick}$ *set*›
where ‹*DIVERGENCES P* ≡ *snd P*›

definition *REFUSALS* :: ‹('*a*, '*r*) *process*$_0$ ⇒ ('*a*, '*r*) *refusal*$_{ptick}$ *set*›
where ‹*REFUSALS P* ≡ {*ref*. ([], *ref*) ∈ *FAILURES P*}›

2.3 The Process Type Invariant

definition *is-process* :: ‹('*a*, '*r*) *process*$_0$ ⇒ *bool*› **where**
‹*is-process P* ≡
([], {}) ∈ *FAILURES P* ∧
(∀ *s X*. (*s*, *X*) ∈ *FAILURES P* ⟶ *ftF s*) ∧
(∀ *s t*. (*s* @ *t*, {}) ∈ *FAILURES P* ⟶ (*s*, {}) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *Y*) ∈ *FAILURES P* ∧ *X* ⊆ *Y* ⟶ (*s*, *X*) ∈ *FAILURES P*) ∧
(∀ *s X Y*. (*s*, *X*) ∈ *FAILURES P* ∧ (∀ *c*. *c* ∈ *Y* ⟶ (*s* @ [*c*], {}) ∉ *FAILURES P*)
⟶ (*s*, *X* ∪ *Y*) ∈ *FAILURES P*) ∧
(∀ *s r X*. (*s* @ [✓(*r*)], {}) ∈ *FAILURES P* ⟶ (*s*, *X* − {✓(*r*)}) ∈ *FAILURES P*) ∧
(∀ *s t*. *s* ∈ *DIVERGENCES P* ∧ *tF s* ∧ *ftF t* ⟶ *s* @ *t* ∈ *DIVERGENCES P*) ∧
(∀ *s X*. *s* ∈ *DIVERGENCES P* ⟶ (*s*, *X*) ∈ *FAILURES P*) ∧

$(\forall\, s\ r.\ s\ @\ [\checkmark(r)] \in \mathit{DIVERGENCES}\ P \longrightarrow s \in \mathit{DIVERGENCES}\ P)\rangle$

lemma *is-process-spec*:
$\langle \mathit{is\text{-}process}\ P \longleftrightarrow$
$([], \{\}) \in \mathit{FAILURES}\ P \wedge$
$(\forall\, s\ X.\ (s, X) \in \mathit{FAILURES}\ P \longrightarrow \mathit{ftF}\ s) \wedge$
$(\forall\, s\ t.\ (s\ @\ t, \{\}) \notin \mathit{FAILURES}\ P \vee (s, \{\}) \in \mathit{FAILURES}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, Y) \notin \mathit{FAILURES}\ P \vee \neg\ X \subseteq Y \vee (s, X) \in \mathit{FAILURES}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, X) \in \mathit{FAILURES}\ P \wedge (\forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathit{FAILURES}\ P)$
$\longrightarrow (s, X \cup Y) \in \mathit{FAILURES}\ P) \wedge$
$(\forall\, s\ r\ X.\ (s\ @\ [\checkmark(r)], \{\}) \in \mathit{FAILURES}\ P \longrightarrow (s, X - \{\checkmark(r)\}) \in \mathit{FAILURES}\ P) \wedge$
$(\forall\, s\ t.\ s \notin \mathit{DIVERGENCES}\ P \vee \neg\ \mathit{tF}\ s \vee \neg\ \mathit{ftF}\ t \vee s\ @\ t \in \mathit{DIVERGENCES}\ P) \wedge$
$(\forall\, s\ X.\ s \notin \mathit{DIVERGENCES}\ P \vee (s, X) \in \mathit{FAILURES}\ P) \wedge$
$(\forall\, s\ r.\ s\ @\ [\checkmark(r)] \notin \mathit{DIVERGENCES}\ P \vee s \in \mathit{DIVERGENCES}\ P)\rangle$
$\langle proof \rangle$

lemma *Process-eqI* :
$\langle \mathit{FAILURES}\ P = \mathit{FAILURES}\ Q \Longrightarrow \mathit{DIVERGENCES}\ P = \mathit{DIVERGENCES}\ Q \Longrightarrow P = Q\rangle$
$\langle proof \rangle$

lemma *process-eq-spec*:
$\langle P = Q \longleftrightarrow \mathit{FAILURES}\ P = \mathit{FAILURES}\ Q \wedge \mathit{DIVERGENCES}\ P = \mathit{DIVERGENCES}\ Q\rangle$
$\langle proof \rangle$

lemma *process-surj-pair*: $\langle (\mathit{FAILURES}\ P, \mathit{DIVERGENCES}\ P) = P\rangle$
$\langle proof \rangle$

lemma *Fa-eq-imp-Tr-eq*: $\langle \mathit{FAILURES}\ P = \mathit{FAILURES}\ Q \Longrightarrow \mathit{TRACES}\ P = \mathit{TRACES}\ Q\rangle$
$\langle proof \rangle$

lemma *is-process1* : $\langle ([], \{\}) \in \mathit{FAILURES}\ P\rangle$
and *is-process2* : $\langle (s, X) \in \mathit{FAILURES}\ P \Longrightarrow \mathit{ftF}\ s\rangle$
and *is-process3* : $\langle (s\ @\ t, \{\}) \in \mathit{FAILURES}\ P \Longrightarrow (s, \{\}) \in \mathit{FAILURES}\ P\rangle$
and *is-process4* : $\langle [\![\mathit{is\text{-}process}\ P; (s, Y) \in \mathit{FAILURES}\ P; X \subseteq Y]\!] \Longrightarrow (s, X) \in \mathit{FAILURES}\ P\rangle$
and *is-process5* : $\langle [\![\mathit{is\text{-}process}\ P; (s, X) \in \mathit{FAILURES}\ P; \forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathit{FAILURES}\ P]\!]$
$\Longrightarrow (s, X \cup Y) \in \mathit{FAILURES}\ P\rangle$
and *is-process6* : $\langle (s\ @\ [\checkmark(r)], \{\}) \in \mathit{FAILURES}\ P \Longrightarrow (s, X - \{\checkmark(r)\}) \in$

FAILURES P›
and *is-process7* : ‹⟦*s* ∈ *DIVERGENCES P*; *tF s*; *ftF t*⟧ ⟹ *s* @ *t* ∈ *DIVERGENCES P*›
and *is-process8* : ‹*s* ∈ *DIVERGENCES P* ⟹ (*s*, *X*) ∈ *FAILURES P*›
and *is-process9* : ‹*s* @ [✓(*r*)] ∈ *DIVERGENCES P* ⟹ *s* ∈ *DIVERGENCES P*›
if ‹*is-process P*›
⟨*proof*⟩

lemma *trace-with-Tick-imp-tickFree-front* :
‹*is-process P* ⟹ *t* @ [✓(*r*)] ∈ *TRACES P* ⟹ *tF t*›
⟨*proof*⟩

2.4 The Abstraction to the process-Type

typedef ('*a*, '*r*) *process*$_{ptick}$ = ‹{*p* :: ('*a*, '*r*) *process*$_0$. *is-process p*}›
morphisms *process*$_0$*-of-process process-of-process*$_0$
⟨*proof*⟩

Again, the old version without parameterized termination can be recovered by considering ('*a*, *unit*) *process*$_{ptick}$.

type-synonym '*a process* = ‹('*a*, *unit*) *process*$_{ptick}$›

setup-lifting *type-definition-process*$_{ptick}$

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

lift-definition *Failures* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *failure*$_{ptick}$ *set*› (‹$\mathcal{F}$›) **is** *FAILURES* ⟨*proof*⟩

lift-definition *Traces* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *trace*$_{ptick}$ *set*› (‹$\mathcal{T}$›) **is** *TRACES* ⟨*proof*⟩

lift-definition *Divergences* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *divergence*$_{ptick}$ *set*› (‹$\mathcal{D}$›) **is** *DIVERGENCES* ⟨*proof*⟩

lift-definition *Refusals* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *refusal*$_{ptick}$ *set*› (‹$\mathcal{R}$›) **is** *REFUSALS* ⟨*proof*⟩

lemma *Refusals-def-bis* : ‹$\mathcal{R}$ *P* = {*X*. ([], *X*) ∈ $\mathcal{F}$ *P*}›
⟨*proof*⟩

lemma *Refusals-iff* : ‹*X* ∈ $\mathcal{R}$ *P* ⟷ ([], *X*) ∈ $\mathcal{F}$ *P*›
⟨*proof*⟩

lemma *T-def-spec*: ‹$\mathcal{T}$ *P* = {*tr*. ∃*f*. *f* ∈ $\mathcal{F}$ *P* ∧ *tr* = *fst f*}›

⟨*proof*⟩

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

lemma *Process-spec*: ‹*process-of-process*$_0$ $(\mathcal{F}\ P, \mathcal{D}\ P) = P$›
⟨*proof*⟩

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

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

lemma *is-processT*:
‹$([], \{\}) \in \mathcal{F}\ P \wedge$
$(\forall\, s\ X.\ (s, X) \in \mathcal{F}\ P \longrightarrow ftF\ s) \wedge$
$(\forall\, s\ t.\ (s\ @\ t, \{\}) \in \mathcal{F}\ P \longrightarrow (s, \{\}) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, X) \in \mathcal{F}\ P \wedge (\forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathcal{F}\ P) \longrightarrow (s, X \cup Y) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ r\ X.\ (s\ @\ [\checkmark(r)], \{\}) \in \mathcal{F}\ P \longrightarrow (s, X - \{\checkmark(r)\}) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ t.\ s \in \mathcal{D}\ P \wedge tF\ s \wedge ftF\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P) \wedge$
$(\forall\, s\ r\ X.\ s \in \mathcal{D}\ P \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge (\forall\, s.\ s\ @\ [\checkmark(r)] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P)$›
⟨*proof*⟩

When the second type is set to *unit*, we recover the classical definition as defined in the book by Roscoe.

lemma *is-processT-unit*:
‹$([], \{\}) \in \mathcal{F}\ P \wedge$
$(\forall\, s\ X.\ (s, X) \in \mathcal{F}\ P \longrightarrow ftF\ s) \wedge$
$(\forall\, s\ t.\ (s\ @\ t, \{\}) \in \mathcal{F}\ P \longrightarrow (s, \{\}) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, Y) \in \mathcal{F}\ P \wedge X \subseteq Y \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s, X) \in \mathcal{F}\ P \wedge (\forall\, c.\ c \in Y \longrightarrow (s\ @\ [c], \{\}) \notin \mathcal{F}\ P) \longrightarrow (s, X \cup Y) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X.\ (s\ @\ [\checkmark], \{\}) \in \mathcal{F}\ P \longrightarrow (s, X - \{\checkmark\}) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ t.\ s \in \mathcal{D}\ P \wedge tF\ s \wedge ftF\ t \longrightarrow s\ @\ t \in \mathcal{D}\ P) \wedge$
$(\forall\, s\ X.\ s \in \mathcal{D}\ P \longrightarrow (s, X) \in \mathcal{F}\ P) \wedge (\forall\, s.\ s\ @\ [\checkmark] \in \mathcal{D}\ P \longrightarrow s \in \mathcal{D}\ P)$›
⟨*proof*⟩

lemma *process-charn*:
‹$([], \{\}) \in \mathcal{F}\ P \wedge$
$(\forall\, s\ X.\ (s, X) \in \mathcal{F}\ P \longrightarrow ftF\ s) \wedge$
$(\forall\, s\ t.\ (s\ @\ t, \{\}) \notin \mathcal{F}\ P \vee (s, \{\}) \in \mathcal{F}\ P) \wedge$

$(\forall\, s\ X\ Y.\ (s,\ Y) \notin \mathcal{F}\ P \vee \neg\ X \subseteq Y \vee (s,\ X) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ X\ Y.\ (s,\ X) \in \mathcal{F}\ P \wedge (\forall\, c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin \mathcal{F}\ P) \longrightarrow (s,\ X \cup Y) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ r\ X.\ (s\ @\ [\checkmark(r)],\ \{\}) \in \mathcal{F}\ P \longrightarrow (s,\ X - \{\checkmark(r)\}) \in \mathcal{F}\ P) \wedge$
$(\forall\, s\ t.\ s \notin \mathcal{D}\ P \vee \neg\ tF\ s \vee \neg\ ftF\ t \vee s\ @\ t \in \mathcal{D}\ P) \wedge$
$(\forall\, s\ r\ X.\ s \notin \mathcal{D}\ P \vee (s,\ X) \in \mathcal{F}\ P) \wedge (\forall\, s.\ s\ @\ [\checkmark(r)] \notin \mathcal{D}\ P \vee s \in \mathcal{D}\ P)\rangle$
⟨*proof*⟩

split of `is_processT`:

lemma *is-processT1* : $\langle([],\ \{\}) \in \mathcal{F}\ P\rangle$
and *is-processT1-TR* : $\langle[] \in \mathcal{T}\ P\rangle$
and *is-processT2* : $\langle(s,\ X) \in \mathcal{F}\ P \Longrightarrow ftF\ s\rangle$
and *is-processT2-TR* : $\langle s \in \mathcal{T}\ P \Longrightarrow ftF\ s\rangle$
and *is-processT3* : $\langle(s\ @\ t,\ \{\}) \in \mathcal{F}\ P \Longrightarrow (s,\ \{\}) \in \mathcal{F}\ P\rangle$
and *is-processT3-pref* : $\langle(t,\ \{\}) \in \mathcal{F}\ P \Longrightarrow s \leq t \Longrightarrow (s,\ \{\}) \in \mathcal{F}\ P\rangle$
and *is-processT3-TR* : $\langle t \in \mathcal{T}\ P \Longrightarrow s \leq t \Longrightarrow s \in \mathcal{T}\ P\rangle$
and *is-processT3-TR-pref* : $\langle(t,\ \{\}) \in \mathcal{F}\ P \Longrightarrow s \leq t \Longrightarrow (s,\ \{\}) \in \mathcal{F}\ P\rangle$
and *is-processT4* : $\langle(s,\ Y) \in \mathcal{F}\ P \Longrightarrow X \subseteq Y \Longrightarrow (s,\ X) \in \mathcal{F}\ P\rangle$
and *is-processT5* : $\langle(s,\ X) \in \mathcal{F}\ P \Longrightarrow \forall\, c.\ c \in Y \longrightarrow (s\ @\ [c],\ \{\}) \notin \mathcal{F}\ P \Longrightarrow (s,\ X \cup Y) \in \mathcal{F}\ P\rangle$
and *is-processT6* : $\langle(s\ @\ [\checkmark(r)],\ \{\}) \in \mathcal{F}\ P \Longrightarrow (s,\ X - \{\checkmark(r)\}) \in \mathcal{F}\ P\rangle$
and *is-processT6-TR* : $\langle s\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow (s,\ X - \{\checkmark(r)\}) \in \mathcal{F}\ P\rangle$
and *is-processT7* : $\langle s \in \mathcal{D}\ P \Longrightarrow tF\ s \Longrightarrow ftF\ t \Longrightarrow s\ @\ t \in \mathcal{D}\ P\rangle$
and *is-processT8* : $\langle s \in \mathcal{D}\ P \Longrightarrow (s,\ X) \in \mathcal{F}\ P\rangle$
and *is-processT9* : $\langle s\ @\ [\checkmark(r)] \in \mathcal{D}\ P \Longrightarrow s \in \mathcal{D}\ P\rangle$
⟨*proof*⟩

lemma *is-processT6-notin* : $\langle(s\ @\ [\checkmark(r)],\ \{\}) \in \mathcal{F}\ P \Longrightarrow \checkmark(r) \notin X \Longrightarrow (s,\ X) \in \mathcal{F}\ P\rangle$
and *is-processT6-TR-notin* : $\langle s\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow \checkmark(r) \notin X \Longrightarrow (s,\ X) \in \mathcal{F}\ P\rangle$
⟨*proof*⟩

lemma *is-processT3-TR-append* : $\langle t\ @\ u \in \mathcal{T}\ P \Longrightarrow t \in \mathcal{T}\ P\rangle$
⟨*proof*⟩

lemma *nonempty-divE* :
$\langle \mathcal{D}\ P \neq \{\} \Longrightarrow (\bigwedge t.\ tF\ t \Longrightarrow t \in \mathcal{D}\ P \Longrightarrow thesis) \Longrightarrow thesis\rangle$
⟨*proof*⟩

lemma *div-butlast-when-non-tickFree-iff* :
$\langle ftF\ s \Longrightarrow (if\ tF\ s\ then\ s\ else\ butlast\ s) \in \mathcal{D}\ P \longleftrightarrow s \in \mathcal{D}\ P\rangle$
⟨*proof*⟩

2.5 Some Consequences of the Process Characterization

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

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

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

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

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 *trace-tick-continuation-or-all-tick-failuresE*:
‹$[\![(s, \{\}) \in \mathcal{F}\ P; \bigwedge r.\ s\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow thesis; (s, range\ tick) \in \mathcal{F}\ P \Longrightarrow thesis]\!] \Longrightarrow thesis$›
⟨*proof*⟩

lemmas *Nil-elem-T* [*simp*] = *is-processT1-TR*

lemmas *F-imp-front-tickFree* = *is-processT2*
and *D-imp-front-tickFree* = *is-processT8*[*THEN is-processT2*]
and *T-imp-front-tickFree* = *T-F*[*THEN is-processT2*]

lemma *D-front-tickFree-subset* : ‹$\mathcal{D}\ P \subseteq Collect\ ftF$›
⟨*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 @ $s \in \mathcal{T}\ P \Longrightarrow s \neq [] \Longrightarrow tF\ t$›
⟨*proof*⟩

lemma *tick-T-F*: ‹t @ [✓(r)] $\in \mathcal{T}\ P \Longrightarrow (t$ @ [✓(r)], $X) \in \mathcal{F}\ P$›
⟨*proof*⟩

lemma *is-processT9-tick*: ‹[✓(r)] $\in \mathcal{D}\ P \Longrightarrow ftF\ s \Longrightarrow s \in \mathcal{D}\ P$›
⟨*proof*⟩

lemma *T-nonTickFree-imp-decomp*: ‹$t \in \mathcal{T}\ P \Longrightarrow \neg\ tF\ t \Longrightarrow \exists\, s\ r.\ t = s$ @ [✓(r)]›
⟨*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 $\mathcal{R}_a$ (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* $X \equiv \{s \in X.\ \forall\, t \in X.\ \neg\ (t < s)\}$›

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 ftF*) = {[]}›
⟨*proof*⟩

lemma *min-elems5* : ‹(s :: '*a list*) $\in A \Longrightarrow \exists\, t{\leq}s.\ t \in$ *min-elems* A›

⟨*proof*⟩

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

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

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

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

definition *Refusals-after* :: ‹$[('a,\ 'r)\ process_{ptick},\ ('a,\ 'r)\ trace_{ptick}] \Rightarrow ('a,\ 'r)\ refusal_{ptick}\ set$› (‹$\mathcal{R}_a$›)
where ‹$\mathcal{R}_a\ P\ tr \equiv \{ref.\ (tr,\ ref) \in \mathcal{F}\ P\}$›

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
$process_{ptick}$ :: $(type,\ type)\ below$
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*: ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow ((s,\ X) \in \mathcal{F}\ Q) = ((s,\ X) \in \mathcal{F}\ P)$›
⟨*proof*⟩

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

lemma *le-approx2T*: ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow s \in \mathcal{T}\ Q \longleftrightarrow s \in \mathcal{T}\ P$›
⟨*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* : ‹$P \sqsubseteq Q \Longrightarrow s \notin \mathcal{D}\ P \Longrightarrow (s, X) \in \mathcal{F}\ P \longleftrightarrow (s, X) \in \mathcal{F}\ Q$›
⟨*proof*⟩

instance $process_{ptick}$ :: (*type*, *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:

- `po_class.chainE` : *chain ?Y* $\Longrightarrow$ *?Y ?i* $\sqsubseteq$ *?Y* (*Suc ?i*)
- `po_class.chain_mono` : ⟦*chain ?Y*; *?i* $\leq$ *?j*⟧ $\Longrightarrow$ *?Y ?i* $\sqsubseteq$ *?Y ?j*
- `po_class.is_ubD` : ⟦*?S* <| *?u*; *?x* $\in$ *?S*⟧ $\Longrightarrow$ *?x* $\sqsubseteq$ *?u*
- `po_class.ub_rangeI` :
  ($\bigwedge i$. *?S i* $\sqsubseteq$ *?x*) $\Longrightarrow$ *range ?S* <| *?x*
- `po_class.ub_imageD` : ⟦*?f* ‘ *?S* <| *?u*; *?x* $\in$ *?S*⟧ $\Longrightarrow$ *?f ?x* $\sqsubseteq$ *?u*
- `po_class.is_ub_upward` : ⟦*?S* <| *?x*; *?x* $\sqsubseteq$ *?y*⟧ $\Longrightarrow$ *?S* <| *?y*
- `po_class.is_lubD1` : *?S* <<| *?x* $\Longrightarrow$ *?S* <| *?x*
- `po_class.is_lubI` : ⟦*?S* <| *?x*; $\bigwedge u$. *?S* <| *u* $\Longrightarrow$ *?x* $\sqsubseteq$ *u*⟧ $\Longrightarrow$ *?S* <<| *?x*
- `po_class.is_lub_maximal` : ⟦*?S* <| *?x*; *?x* $\in$ *?S*⟧ $\Longrightarrow$ *?S* <<| *?x*
- `po_class.is_lub_lub` : *?M* <<| *?x* $\Longrightarrow$ *?M* <<| *lub ?M*
- `po_class.is_lub_range_shift`:
  chain ?S $\Longrightarrow$ *range* (λi. *?S* (*i* + *?j*)) <<| *?x* = *range ?S* <<| *?x*

- `po_class.is_lub_rangeD1`: $range\ ?S <<|\ ?x \Longrightarrow ?S\ ?i \sqsubseteq ?x$
- `po_class.lub_eqI`: $?M <<|\ ?l \Longrightarrow lub\ ?M = ?l$
- `po_class.is_lub_unique`:$[\![?S <<|\ ?x;\ ?S <<|\ ?y]\!] \Longrightarrow ?x = ?y$

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

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

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

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

context fixes S :: ‹$nat \Rightarrow ('a, 'r)\ process_{ptick}$›
assumes ‹$chain\ S$›
begin

lift-definition *lim-proc* :: ‹$('a, 'r)\ process_{ptick}$›
is ‹$(\bigcap\ (\mathcal{F}\ `\ range\ S), \bigcap\ (\mathcal{D}\ `\ range\ S))$›
⟨*proof*⟩

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

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

lemma *T-LUB*: ‹$\mathcal{T}\ lim\text{-}proc = \bigcap\ (\mathcal{T}\ `\ range\ S)$›

⟨*proof*⟩

lemmas *LUB-projs* = *F-LUB D-LUB T-LUB*

lemma *Refusals-LUB*: ‹$\mathcal{R}$ *lim-proc* = $\bigcap$ ($\mathcal{R}$ ‘ *range S*)›
⟨*proof*⟩

lemma *Refusals-after-LUB*: ‹$\mathcal{R}_a$ *lim-proc s* = ($\bigcap i.$ ($\mathcal{R}_a$ (*S i*) *s*))›
⟨*proof*⟩

lemma *F-LUB-2*: ‹(*s*, *X*) ∈ $\mathcal{F}$ *lim-proc* ⟷ (∀ *i*. (*s*, *X*) ∈ $\mathcal{F}$ (*S i*))›
and *D-LUB-2*: ‹*t* ∈ $\mathcal{D}$ *lim-proc* ⟷ (∀ *i*. *t* ∈ $\mathcal{D}$ (*S i*))›
and *T-LUB-2*: ‹*t* ∈ $\mathcal{T}$ *lim-proc* ⟷ (∀ *i*. *t* ∈ $\mathcal{T}$ (*S i*))›
and *Refusals-LUB-2*: ‹*X* ∈ $\mathcal{R}$ *lim-proc* ⟷ (∀ *i*. *X* ∈ $\mathcal{R}$ (*S i*))›
and *Refusals-after-LUB-2*: ‹*X* ∈ $\mathcal{R}_a$ *lim-proc s* ⟷ (∀ *i*. *X* ∈ $\mathcal{R}_a$ (*S i*) *s*)›
⟨*proof*⟩

end

By exiting the context, terms like $\mathcal{F}$ *lim-proc* will become $\mathcal{F}$ (*lim-proc S*) and the assumption *chain S* will be added.

2.7 Process Refinement is a Partial Ordering

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

instantiation $process_{ptick}$ :: (*type*, *type*) *ord*
begin

definition $less\text{-}eq\text{-}process_{ptick}$ :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $process_{ptick}$ ⇒ *bool*›
where ‹$less\text{-}eq\text{-}process_{ptick}$ *P Q* ≡ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P* ∧ $\mathcal{F}$ *Q* ⊆ $\mathcal{F}$ *P*›

definition $less\text{-}process_{ptick}$ :: ‹('*a*, '*r*) $process_{ptick}$ ⇒ ('*a*, '*r*) $process_{ptick}$ ⇒ *bool*›
where ‹$less\text{-}process_{ptick}$ *P Q* ≡ *P* ≤ *Q* ∧ *P* ≠ *Q*›

instance ⟨*proof*⟩

end

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

lemma *le-ref1* : ‹*P* ≤ *Q* ⟹ $\mathcal{D}$ *Q* ⊆ $\mathcal{D}$ *P*›
and *le-ref2* : ‹*P* ≤ *Q* ⟹ $\mathcal{F}$ *Q* ⊆ $\mathcal{F}$ *P*›
and *le-ref2T* : ‹*P* ≤ *Q* ⟹ $\mathcal{T}$ *Q* ⊆ $\mathcal{T}$ *P*›
and *le-approx-imp-le-ref*: ‹(*P*::('*a*, '*r*) $process_{ptick}$) ⊑ *Q* ⟹ *P* ≤ *Q*›

⟨*proof*⟩

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

lemma *D-extended-is-D* :
‹$\{t\ @\ u\ |t\ u.\ t \in \mathcal{D}\ P \wedge tF\ t \wedge ftF\ u\} = \mathcal{D}\ P$›
⟨*proof*⟩

lemma *Process-eq-optimizedI* :
‹$[\![\bigwedge t.\ t \in \mathcal{D}\ P \Longrightarrow t \in \mathcal{D}\ Q;\ \bigwedge t.\ t \in \mathcal{D}\ Q \Longrightarrow t \in \mathcal{D}\ P;$
$\bigwedge t\ X.\ (t,\ X) \in \mathcal{F}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow t \notin \mathcal{D}\ Q \Longrightarrow (t,\ X) \in \mathcal{F}\ Q;$
$\bigwedge t\ X.\ (t,\ X) \in \mathcal{F}\ Q \Longrightarrow t \notin \mathcal{D}\ Q \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow (t,\ X) \in \mathcal{F}\ P]\!] \Longrightarrow P = Q$›
⟨*proof*⟩

instance $process_{ptick}$ :: (*type*, *type*) *order*
⟨*proof*⟩

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

lemma *chain-min-elem-div-is-min-for-sequel*:
‹*chain S* ⟹ *s* ∈ *min-elems* (*D* (*S i*)) ⟹ *i* ≤ *j* ⟹ *s* ∈ *D* (*S j*) ⟹ *s* ∈ *min-elems* (*D* (*S j*))›
⟨*proof*⟩

lemma *limproc-is-lub*: ‹*range S* <<| *lim-proc S*› **if** ‹*chain S*›
⟨*proof*⟩

lemma *limproc-is-thelub*: ‹*chain S* ⟹ (⨆ *i*. *S i*) = *lim-proc S*›
⟨*proof*⟩

instance $process_{ptick}$ :: (*type*, *type*) *cpo*
⟨*proof*⟩

instance $process_{ptick}$ :: (*type*, *type*) *pcpo*

⟨*proof*⟩

2.8 Process Refinement is Admissible

lemma *le-FD-adm* : ‹*cont* (*u* :: ('*b*::*cpo*) ⇒ ('*a*, '*r*) *process*$_{ptick}$) ⟹ *monofun v* ⟹ *adm* (λ*x*. *u x* ≤ *v x*)›
⟨*proof*⟩

lemmas *le-FD-adm-cont*[*simp*] = *le-FD-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*]:
‹⟦⋀*x*. *P x* ⟹ *cont* (*f x*); ⋀*x*. ¬ *P x* ⟹ *cont* (*g x*)⟧ ⟹
cont (λ*y*. *if P x then f x y else g x y*)›
for *f* :: ‹'*c* ⇒ '*b* :: *cpo* ⇒ ('*a*, '*r*) *process*$_{ptick}$›
⟨*proof*⟩

2.10 Tools for proving continuity

lemma *cont-process-rec*: ‹*P* = (μ *X*. *f X*) ⟹ *cont f* ⟹ *P* = *f P*›
⟨*proof*⟩

lemma *Inter-nonempty-finite-chained-sets*: ‹(⋂ *i*. *S i*) ≠ {}›
if ‹⋀*i*. *j* ≤ *i* ⟹ *S i* ≠ {}› ‹*finite* (*S j*)› ‹⋀*i*. *S* (*Suc i*) ⊆ *S i*› **for** *S* :: ‹*nat* ⇒ '*a set*›
⟨*proof*⟩

method *prove-finite-subset-of-prefixes* **for** *t* :: ‹('*a*, '*r*) *trace*$_{ptick}$› =
— Useful for establishing the second hypothesis
solves ‹(*rule finite-UnI*; *prove-finite-subset-of-prefixes t*) |
(*rule finite-subset*[*of* - ‹{*u*. *u* ≤ *t*}›],
use prefixI in blast, *simp add*: *prefixes-fin*)›

Chapter 3

Constant Processes

3.1 The Undefined Process: $\bot$

This is the key result: $\bot$ — which we know to exist from the process instantiation — can be explicitly written with its projections.

lemma *F-BOT* : ‹$\mathcal{F}\ \bot = \{(s :: ('a, 'r)\ trace_{ptick},\ X).\ ftF\ s\}$›
  and *D-BOT* : ‹$\mathcal{D}\ \bot = \{d :: ('a, 'r)\ trace_{ptick}.\ ftF\ d\}$›
  and *T-BOT* : ‹$\mathcal{T}\ \bot = \{s :: ('a, 'r)\ trace_{ptick}.\ ftF\ s\}$›
⟨*proof*⟩

lemmas *BOT-projs* = *F-BOT D-BOT T-BOT*

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

lemma *BOT-iff-tick-D* : ‹$P = \bot \longleftrightarrow (\exists\, r.\ [\checkmark(r)] \in \mathcal{D}\ P)$›
  ⟨*proof*⟩

3.2 The SKIP Process

In this new parameterized version, the SKIP process is of course also parameterized.

lift-definition *SKIP* :: ‹$'r \Rightarrow ('a, 'r)\ process_{ptick}$›
  is ‹$\lambda r.\ (\{([], X) |\ X.\ \checkmark(r) \notin X\} \cup \{(s, X).\ s = [\checkmark(r)]\}, \{\})$›
  ⟨*proof*⟩

abbreviation *SKIP-unit* :: ‹*'a process*› (‹*Skip*›) **where** ‹*Skip* ≡ *SKIP* ()›

lemma *F-SKIP* :
  ‹$\mathcal{F}\ (SKIP\ r) = \{([], X) |\ X.\ \checkmark(r) \notin X\} \cup \{(s, X).\ s = [\checkmark(r)]\}$›
  ⟨*proof*⟩

lemma *D-SKIP* : ‹$\mathcal{D}$ (*SKIP r*) = {}›
⟨*proof*⟩

lemma *T-SKIP* : ‹$\mathcal{T}$ (*SKIP r*) = {[], [✓(*r*)]}›
⟨*proof*⟩

lemmas *SKIP-projs* = *F-SKIP D-SKIP T-SKIP*

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

3.3 The STOP Process

lift-definition *STOP* :: ‹('*a*, '*r*) *process*$_{ptick}$›
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*⟩

lemmas *STOP-projs* = *F-STOP D-STOP T-STOP*

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

Chapter 4

The Binary Choice Operators

4.1 Deterministic Choice

4.1.1 The Det Operator Definition

lift-definition *Det* :: ‹[('a, 'r)*process*$_{ptick}$, ('a, 'r)*process*$_{ptick}$] ⇒ ('a, 'r)*process*$_{ptick}$›
(**infixl** ‹[+]› *82*)
  is ‹λ*P Q*. ({(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ $\mathcal{F}$ *P* ∩ $\mathcal{F}$ *Q*} ∪
        {(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ $\mathcal{F}$ *P* ∪ $\mathcal{F}$ *Q*} ∪
        {(*s*, *X*). *s* = [] ∧ *s* ∈ $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*} ∪
        {(*s*, *X*). ∃ *r*. *s* = [] ∧ ✓(*r*) ∉ *X* ∧ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∪ $\mathcal{T}$ *Q*},
        $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*)›
⟨*proof*⟩

notation *Det* (**infixl** ‹□› *82*)

4.1.2 The Projections

lemma *F-Det* :
  ‹$\mathcal{F}$ (*P* □ *Q*) = {(*s*, *X*). *s* = [] ∧ (*s*, *X*) ∈ $\mathcal{F}$ *P* ∩ $\mathcal{F}$ *Q*} ∪
            {(*s*, *X*). *s* ≠ [] ∧ (*s*, *X*) ∈ $\mathcal{F}$ *P* ∪ $\mathcal{F}$ *Q*} ∪
            {(*s*, *X*). *s* = [] ∧ *s* ∈ $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*} ∪
            {(*s*, *X*). ∃ *r*. *s* = [] ∧ ✓(*r*) ∉ *X* ∧ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∪ $\mathcal{T}$ *Q*}›
  ⟨*proof*⟩

lemma *D-Det* : ‹$\mathcal{D}$ (*P* □ *Q*) = $\mathcal{D}$ *P* ∪ $\mathcal{D}$ *Q*›
  ⟨*proof*⟩

lemma *T-Det* : ‹$\mathcal{T}$ (*P* □ *Q*) = $\mathcal{T}$ *P* ∪ $\mathcal{T}$ *Q*›
  ⟨*proof*⟩

lemmas *Det-projs* = *F-Det D-Det T-Det*

4.1.3 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*⟩

lemma *Det-id* [*simp*] : ‹$P \Box P = P$›
⟨*proof*⟩

4.1.4 The Continuity-Rule

lemma *mono-Det* : ‹$(P :: ('a, 'r)process_{ptick}) \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \Box Q \sqsubseteq P' \Box Q'$›
⟨*proof*⟩

lemma *chain-Det* : ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i \Box S)$›
⟨*proof*⟩

lemma *cont-Det-prem* : ‹$((\bigsqcup\ i.\ Y\ i) \Box S) = (\bigsqcup\ i.\ (Y\ i \Box S))$› **if** ‹*chain* Y›
⟨*proof*⟩

lemma *Det-cont*[*simp*]: ‹*cont* $(\lambda x.\ f\ x \Box g\ x)$› **if** ‹*cont* f› **and** ‹*cont* g›
⟨*proof*⟩

4.2 Non Deterministic Choice

4.2.1 The Ndet Operator Definition

lift-definition *Ndet* :: ‹$[('a, 'r)process_{ptick}, ('a, 'r)process_{ptick}] \Rightarrow ('a, 'r)process_{ptick}$›
(**infixl** ‹$|-|$› *83*)
is ‹$\lambda P\ Q.\ (\mathcal{F}\ P \cup \mathcal{F}\ Q\ ,\ \mathcal{D}\ P \cup \mathcal{D}\ Q)$›
⟨*proof*⟩

notation *Ndet* (**infixl** ‹$\sqcap$› *83*)

4.2.2 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*⟩

lemmas *Ndet-projs* = *F-Ndet D-Ndet T-Ndet*

4.2.3 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*⟩

lemma *Ndet-id* [*simp*] : ‹$P \sqcap P = P$› ⟨*proof*⟩

4.2.4 The Continuity Rule

lemma *mono-Ndet* : ‹$P \sqcap Q \sqsubseteq P' \sqcap Q'$› **if** ‹$P \sqsubseteq P'$› **and** ‹$Q \sqsubseteq Q'$›
⟨*proof*⟩

lemma *cont-Ndet-prem* : ‹$(\bigsqcup i.\ Y\ i) \sqcap S = (\bigsqcup i.\ Y\ i \sqcap S)$› **if** ‹*chain* Y›
⟨*proof*⟩

lemma *Ndet-cont*[*simp*]: ‹*cont* $(\lambda x.\ f\ x \sqcap g\ x)$› **if** ‹*cont* f› **and** ‹*cont* g›
⟨*proof*⟩

4.3 Sliding Choice

4.3.1 Definition

definition *Sliding* :: ‹$('a, 'r)process_{ptick} \Rightarrow ('a, 'r)process_{ptick} \Rightarrow ('a, 'r)process_{ptick}$›
(**infixl** ‹$\rhd$› *80*)
where ‹$P \rhd Q \equiv (P \mathbin{\square} Q) \sqcap Q$›

4.3.2 Projections

lemma *F-Sliding*:
‹$\mathcal{F}\ (P \rhd Q) = \mathcal{F}\ Q \cup \{(s, X).\ s \neq [] \wedge (s, X) \in \mathcal{F}\ P \vee$
$s = [] \wedge (\exists r.\ s \in \mathcal{D}\ P \vee tick\ r \notin X \wedge [tick\ r] \in \mathcal{T}\ P)\}$›
⟨*proof*⟩

lemma *D-Sliding*: ‹$\mathcal{D}\ (P \rhd Q) = \mathcal{D}\ P \cup \mathcal{D}\ Q$›
⟨*proof*⟩

lemma *T-Sliding*: ‹$\mathcal{T}\ (P \rhd Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›
⟨*proof*⟩

lemmas *Sliding-projs* = *F-Sliding D-Sliding T-Sliding*

4.3.3 Properties

lemma *Sliding-id*: ‹$P \rhd P = P$›
⟨*proof*⟩

4.3.4 Continuity

From the definition, monotony and continuity is obvious.

lemma *mono-Sliding* : ‹$P \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P \rhd Q \sqsubseteq P' \rhd Q'$›
⟨*proof*⟩

lemma *Sliding-cont*[*simp*] : ‹*cont* $f \Longrightarrow$ *cont* $g \Longrightarrow$ *cont* $(\lambda x.\ f\ x \rhd g\ x)$›
⟨*proof*⟩

Chapter 5

The Prefix Choice Operators

5.1 Multiple Deterministic Prefix Choice

5.1.1 The Definition and some Consequences

lift-definition *Mprefix* :: ‹['a set, 'a ⇒ ('a, 'r)process$_{ptick}$] ⇒ ('a, 'r)process$_{ptick}$›
  is ‹λ*A P*. ({(*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*))})›
⟨*proof*⟩

syntax -*Mprefix* :: ‹[*pttrn*, 'a set, ('a, 'r)process$_{ptick}$] ⇒ ('a, 'r)process$_{ptick}$›
  (‹(3□((-)/∈(-))/ → (-))› [78,78,77] 77)
syntax-consts -*Mprefix* ⇌ *Mprefix*
translations □*a*∈*A* → *P* ⇌ *CONST Mprefix A* (λ*a*. *P*)

Syntax Check:

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

5.1.2 Projections in Prefix

lemma *F-Mprefix* :
  $\mathcal{F}$ (□*a* ∈ *A* → *P a*) = {([], *X*)| *X*. *X* ∩ *ev* ‘ *A* = {}} ∪
                {(*ev a* # *s*, *X*)| *a s X*. *a* ∈ *A* ∧ (*s*, *X*) ∈ $\mathcal{F}$ (*P a*)}
  ⟨*proof*⟩

lemma *D-Mprefix*: ‹$\mathcal{D}$ (□*a* ∈ *A* → *P a*) = {*ev a* # *s*| *a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{D}$ (*P a*)}›
  ⟨*proof*⟩

lemma *T-Mprefix*: ‹$\mathcal{T}$ (□*a* ∈ *A* → *P a*) = *insert* [] {*ev a* # *s*| *a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{T}$ (*P a*)}›
  ⟨*proof*⟩

lemmas *Mprefix-projs* = *F-Mprefix D-Mprefix T-Mprefix*

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

5.1.3 Basic Properties

lemma *tick-notin-T-Mprefix* [*simp*]: ‹$[\checkmark(r)] \notin \mathcal{T}\ (\Box x \in A \rightarrow P\ x)$›
⟨*proof*⟩

lemma *Nil-notin-D-Mprefix* [*simp*]: ‹$[] \notin \mathcal{D}\ (\Box x \in A \rightarrow P\ x)$›
⟨*proof*⟩

5.1.4 Proof of Continuity Rule

Backpatch Isabelle 2009-1

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

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

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

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

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

Core of Proof

lemma *mono-Mprefix* : ‹$\Box a \in A \rightarrow P\ a \sqsubseteq \Box a \in A \rightarrow Q\ a$› (**is** ‹$?P \sqsubseteq ?Q$›)
if ‹$\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a$›
⟨*proof*⟩

lemma *chain-Mprefix* : ‹$chain\ Y \Longrightarrow chain\ (\lambda i.\ \Box a{\in}A \rightarrow Y\ i\ a)$›
⟨*proof*⟩

lemma *cont-Mprefix-prem* : ‹$\Box a \in A \to (\bigsqcup i.\ Y\ i\ a) = (\bigsqcup i.\ \Box a \in A \to Y\ i\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
⟨*proof*⟩

lemma *Mprefix-cont* [*simp*] : ‹*cont* ($\lambda b.\ \Box a \in A \to f\ a\ b$)› **if** ∗ : ‹$\bigwedge a.\ a \in A \Longrightarrow$ *cont* ($f\ a$)›
⟨*proof*⟩

5.1.5 High-level Syntax for Read and Write0

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*, $'a \Rightarrow ('b, 'r)process_{ptick}$] $\Rightarrow ('b, 'r)process_{ptick}$›
where ‹*read c A P* ≡ *Mprefix* (*c* ‘ *A*) (*P* ∘ (*inv-into A c*))›

definition *write0* :: ‹[$'a$, $('a, 'r)process_{ptick}$] $\Rightarrow ('a, 'r)process_{ptick}$› (**infixr** ‹→› *77*)
where ‹*write0 a P* ≡ *Mprefix* {*a*} ($\lambda x.\ P$)›

5.1.6 CSP$_M$-Style Syntax for Communication Primitives

syntax
-read :: ‹[$'a \Rightarrow 'b$, *pttrn*, $('b, 'r)process_{ptick}$] $\Rightarrow ('b, 'r)process_{ptick}$›
(‹(*3*((-)**?**(-)) /→ -)› [*78*,*78*,*77*] *77*)
-readX :: ‹[$'a \Rightarrow 'b$, *pttrn*, *bool*, $('b, 'r)process_{ptick}$] $\Rightarrow ('b, 'r)process_{ptick}$›
(‹(*3*((-)**?**(-))|- /→ -)› [*78*,*78*,*78*,*77*] *77*)
-readS :: ‹[$'a \Rightarrow 'b$, *pttrn*, $'a$ *set*, $('b, 'r)process_{ptick}$] $\Rightarrow ('b, 'r)process_{ptick}$›
(‹(*3*((-)**?**(-))∈- /→ -)› [*78*,*78*,*78*,*77*] *77*)

translations
-read c p P ⇌ *CONST read c CONST UNIV* ($\lambda p.\ P$)
-readX c p b P => *CONST read c* {*p*. *b*} ($\lambda p.\ P$)

-readS c p A P => *CONST read c A* ($\lambda p.\ P$)

syntax-consts *-read* ⇌ *read*
and *-readX* ⇌ *read*
and *-readS* ⇌ *read*

Syntax Check:

term ‹*a* → *P*›

term ‹*c***?***x* → *d***?***y* → *P a y*›

term ‹$c?x{\in}X \to P\ x$›
term ‹$c?x|(x{<}0) \to P\ x$›

term ‹$c?x \to d?y{\in}B \to e \to u?t|(t \geq 1) \to P\ a\ y$›

term ‹$(c \circ d)?a \to P\ a$›

lemma *mono-write0* : ‹$P \sqsubseteq Q \Longrightarrow a \to P \sqsubseteq a \to Q$›
⟨*proof*⟩

lemma *mono-read* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a) \Longrightarrow c?a{\in}A \to P\ a \sqsubseteq c?a{\in}A \to Q\ a$›
⟨*proof*⟩

lemma *read-cont*[*simp*]:
‹$(\bigwedge a.\ a \in A \Longrightarrow cont\ (\lambda b.\ P\ b\ a)) \Longrightarrow cont\ (\lambda y.\ read\ c\ A\ (P\ y))$›
⟨*proof*⟩

lemma *read-cont'*[*simp*]: ‹$cont\ P \Longrightarrow cont\ (\lambda y.\ read\ c\ A\ (P\ y))$›
⟨*proof*⟩

lemma *read-cont''*[*simp*]: ‹$(\bigwedge a.\ cont\ (f\ a)) \Longrightarrow cont\ (\lambda y.\ c?x \to f\ x\ y)$› ⟨*proof*⟩

lemma *write0-cont*[*simp*]: ‹$cont\ (P{::}('b{::}cpo \Rightarrow ('a,\ 'r)process_{ptick})) \Longrightarrow cont(\lambda x.\ a \to P\ x)$›
⟨*proof*⟩

lemma *Mprefix-singl* : ‹$\Box x \in \{a\} \to P\ x = a \to P\ a$›
⟨*proof*⟩

5.2 Multiple non Deterministic Prefix Choice

5.3 Multiple non deterministic prefix operator

lift-definition *Mndetprefix* :: ‹$['a\ set,\ 'a \Rightarrow ('a,\ 'r)process_{ptick}] \Rightarrow ('a,\ 'r)process_{ptick}$›
is ‹$\lambda A\ P.\quad if\ A = \{\}\ then\ process_0\text{-}of\text{-}process\ STOP$
$else\ (\bigcup a{\in}A.\ \mathcal{F}\ (a \to P\ a),\ \bigcup a{\in}A.\ \mathcal{D}\ (a \to P\ a))$›
⟨*proof*⟩

syntax -*Mndetprefix* :: *pttrn* ⇒ *'a set* ⇒ $('a,\ 'r)process_{ptick} \Rightarrow ('a,\ 'r)process_{ptick}$

(‹(3⊓((-)/∈(-))/ → (-))› [78,78,77] 77)
syntax-consts *-Mndetprefix* ⇌ *Mndetprefix*

translations ⊓a ∈ A → P ⇌ *CONST Mndetprefix A* (λa. P)

lemma *F-Mndetprefix*:
‹ℱ (⊓a ∈ A → P a) = (*if* A = {} *then* {(s, X). s = []} *else* ⋃x∈A. ℱ (x → P x))›
⟨*proof*⟩

lemma *D-Mndetprefix* : ‹𝒟 (⊓a ∈ A → P a) = (*if* A = {} *then* {} *else* ⋃x∈A. 𝒟 (x → P x))›
⟨*proof*⟩

lemma *T-Mndetprefix*: ‹𝒯 (⊓a ∈ A → P a) = (*if* A = {} *then* {[]} *else* ⋃x∈A. 𝒯 (x → P x))›
⟨*proof*⟩

lemma *mono-Mndetprefix-eq*: ‹(⋀a. a ∈ A ⟹ P a = Q a) ⟹ ⊓a ∈ A → P a = ⊓a ∈ A → Q a›
⟨*proof*⟩

lemma *F-Mndetprefix'* :
‹ℱ (⊓a ∈ A → P a) =
(*if* A = {} *then* {(s, X). s = []}
else {([], X)| X. ∃ a ∈ A. ev a ∉ X} ∪ {(ev a # s, X) |a s X. a ∈ A ∧ (s, X) ∈ ℱ (P a)})›
⟨*proof*⟩

lemma *D-Mndetprefix'* : ‹𝒟 (⊓a ∈ A → P a) = {ev a # s |a s. a ∈ A ∧ s ∈ 𝒟 (P a)}›
⟨*proof*⟩

lemma *T-Mndetprefix'* : ‹𝒯 (⊓a ∈ A → P a) = *insert* [] {ev a # s |a s. a ∈ A ∧ s ∈ 𝒯 (P a)}›
⟨*proof*⟩

lemmas *Mndetprefix-projs* = *F-Mndetprefix' D-Mndetprefix' T-Mndetprefix'*

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

lemma *Mndetprefix-singl* [*simp*] : ‹⊓ a ∈ {a} → P a = a → P a›
⟨*proof*⟩

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

The two lemmas *Mndetprefix* {*?a*} *?P* = *?a* → *?P ?a* and ⟦*?A* ≠ {}; *?B* ≠ {}⟧ ⟹ *Mndetprefix* (*?A* ∪ *?B*) *?P* = *Mndetprefix ?A ?P* ⊓ *Mndetprefix ?B ?P* together give us that *Mndetprefix* can be represented by a fold in the finite case.

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

This also implies that *Mndetprefix* is continuous when *finite A*, but this is not really useful since we have the general case.

5.3.1 General case Continuity

lemma *mono-Mndetprefix* : ‹$\sqcap a \in A \rightarrow P\ a \sqsubseteq \sqcap a \in A \rightarrow Q\ a$›
(**is** ‹*?P* ⊑ *?Q*›) **if** ‹$\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a$›
⟨*proof*⟩

lemma *chain-Mndetprefix* : ‹*chain Y* ⟹ *chain* (λ*i*. ⊓*a*∈*A* → *Y i a*)›
⟨*proof*⟩

lemma *cont-Mndetprefix-prem* : ‹$\sqcap a \in A \rightarrow (\bigsqcup i.\ Y\ i\ a) = (\bigsqcup i.\ \sqcap a \in A \rightarrow Y\ i\ a)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain Y*›
⟨*proof*⟩

lemma *Mndetprefix-cont* [*simp*] : ‹*cont* (λ*b*. ⊓*a* ∈ *A* → *f a b*)› **if** ∗ : ‹$\bigwedge a.\ a \in A \Longrightarrow \mathit{cont}\ (f\ a)$›
⟨*proof*⟩

5.3.2 High-level Syntax for Write

A version with a non deterministic choice is also introduced.

definition *ndet-write* :: ‹['*a* ⇒ '*b*, '*a set*, '*a* ⇒ ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
where ‹*ndet-write c A P* ≡ *Mndetprefix* (*c* ‘ *A*) (*P o* (*inv-into A c*))›

definition *write* :: ‹['*a* ⇒ '*b*, '*a*, ('*b*, '*r*)*process*$_{ptick}$] ⇒ ('*b*, '*r*)*process*$_{ptick}$›
where ‹*write c a P* ≡ *Mprefix* {*c a*} (λ*x*. *P*)›

syntax
-ndet-write :: ‹['a ⇒ 'b, pttrn, ('b, 'r)process$_{ptick}$] ⇒ ('b, 'r)process$_{ptick}$›
(‹(3((-)!!(-)) /→ -)› [78,78,77] 77)
-ndet-writeX :: ‹['a ⇒ 'b, pttrn, bool,('b, 'r)process$_{ptick}$] ⇒ ('b, 'r)process$_{ptick}$›
(‹(3((-)!!(-))|- /→ -)› [78,78,78,77] 77)
-ndet-writeS :: ‹['a ⇒ 'b, pttrn, 'b set,('b, 'r)process$_{ptick}$] ⇒ ('b, 'r)process$_{ptick}$›
(‹(3((-)!!(-))∈- /→ -)› [78,78,78,77] 77)
-write :: ‹['a ⇒ 'b, 'a, ('a, 'r)process$_{ptick}$] ⇒ ('a, 'r)process$_{ptick}$› (‹(3(-)!(-) /→ -)› [78,78,77] 77)

syntax-consts *-ndet-write* ⇌ *ndet-write*
and *-ndet-writeX* ⇌ *ndet-write*
and *-ndet-writeS* ⇌ *ndet-write*
and *-write* ⇌ *ndet-write*

translations
-ndet-write c p P ⇌ *CONST ndet-write c CONST UNIV* (λ*p*. *P*)
-ndet-writeX c p b P => *CONST ndet-write c* {*p*. *b*} (λ*p*. *P*)
-ndet-writeS c p A P => *CONST ndet-write c A* (λ*p*. *P*)
-write c a P ⇌ *CONST write c a P*

Syntax checks.

term ‹*c*!!*x* → *P x*›
term ‹*c*!!*x*∈*A* → *P x*›
term ‹*c*!!*x*|(*0*<*x*) → *P x*›

term ‹(*c* ∘ *c'*)!!*a*∈*A* → *d*?*b*∈*B* → *event* → *e*!*a'* → *P a b*›

term ‹*c*!*x* → *P*›

lemma *mono-ndet-write*: ‹(⋀*a*. *a* ∈ *A* ⟹ *P a* ⊑ *Q a*) ⟹ (*c*!!*a*∈*A* → *P a*) ⊑ (*c*!!*a*∈*A* → *Q a*)›
⟨*proof*⟩

lemma *ndet-write-cont*[*simp*]:
‹(⋀*a*. *a* ∈ *A* ⟹ *cont* (λ*b*. *P b a*)) ⟹ *cont* (λ*y*. *c*!!*a*∈*A* → *P y a*)›
⟨*proof*⟩

lemma *mono-write*: ‹*P* ⊑ *Q* ⟹ *c*!*a* → *P* ⊑ *c*!*a* → *Q*›
⟨*proof*⟩

lemma *write-cont*[*simp*]: ‹*cont P* ⟹ *cont* (λ*x*. *c*!*a* → *P x*)›
⟨*proof*⟩

lemma *read-singl*[*simp*] : ‹$c?a{\in}\{x\} \to P\ a = c!x \to P\ x$›
⟨*proof*⟩

lemma *ndet-write-singl*[*simp*] : ‹$c!!a{\in}\{x\} \to P\ a = c!x \to P\ x$›
⟨*proof*⟩

lemma *write-is-write0* : ‹$c!x \to P = c\ x \to P$›
⟨*proof*⟩

Chapter 6

The Global non Deterministic Choice

6.1 General non Deterministic Choice Definition

This is an experimental definition of a generalized non deterministic choice $a \sqcap b$ for an arbitrary set. The present version is "totalised" for the case of $A = \{\}$ by *STOP*, which is not the neutral element of the $(\sqcap)$ operator (because there is no neutral element for $(\sqcap)$).

lemma ‹$\nexists P.\ \forall Q.\ (P :: ('a, 'r)\ process_{ptick}) \sqcap Q = Q$›
⟨*proof*⟩

lift-definition *GlobalNdet* :: ‹['b *set*, 'b ⇒ ('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
  is ‹$\lambda A\ P.$ *if* $A = \{\}$
        then $(\{(s, X).\ s = []\}, \{\})$
        else $(\bigcup a{\in}A.\ \mathcal{F}\ (P\ a), \bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$›
⟨*proof*⟩

syntax -*GlobalNdet* :: ‹[*pttrn*,'b *set*,('a, 'r) $process_{ptick}$] ⇒ ('a, 'r) $process_{ptick}$›
  (‹(3⊓((-)/∈(-))./ (-))› [78,78,77] 77)
syntax-consts -*GlobalNdet* == *GlobalNdet*
translations $\sqcap a \in A.\ P \rightleftharpoons CONST\ GlobalNdet\ A\ (\lambda a.\ P)$

Note that the global non deterministic choice $\sqcap a{\in}A.\ P\ a$ is different from the multiple non deterministic prefix choice which guarantees continuity even when A is *infinite* due to the fact that it communicates its choice via an internal prefix operator.

6.2 The projections

lemma *F-GlobalNdet*:
‹$\mathcal{F}\ (\sqcap\ a \in A.\ P\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{(s, X).\ s = []\}\ \mathit{else}\ (\bigcup\ a{\in}A.\ \mathcal{F}\ (P\ a)))$›
⟨*proof*⟩

lemma *D-GlobalNdet*: ‹$\mathcal{D}\ (\sqcap\ a \in A.\ P\ a) = (\bigcup\ a{\in}A.\ \mathcal{D}\ (P\ a))$›
⟨*proof*⟩

lemma *T-GlobalNdet*:
‹$\mathcal{T}\ (\sqcap\ a \in A.\ P\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{[]\}\ \mathit{else}\ (\bigcup\ a{\in}A.\ \mathcal{T}\ (P\ a)))$›
⟨*proof*⟩

lemma *T-GlobalNdet'*: ‹$\mathcal{T}\ (\sqcap\ a \in A.\ P\ a) = \mathit{insert}\ []\ (\bigcup\ a{\in}A.\ \mathcal{T}\ (P\ a))$›
⟨*proof*⟩

lemmas *GlobalNdet-projs = F-GlobalNdet D-GlobalNdet T-GlobalNdet'*

lemma *mono-GlobalNdet-eq*:
‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a = Q\ a) \Longrightarrow \mathit{GlobalNdet}\ A\ P = \mathit{GlobalNdet}\ A\ Q$›
⟨*proof*⟩

lemma *mono-GlobalNdet-eq2*:
‹$(\bigwedge a.\ a \in A \Longrightarrow P\ (f\ a) = Q\ a) \Longrightarrow \mathit{GlobalNdet}\ (f\ `\ A)\ P = \mathit{GlobalNdet}\ A\ Q$›
⟨*proof*⟩

6.3 Factorization of $(\sqcap)$ in front of *GlobalNdet*

lemma *GlobalNdet-factorization-union*:
‹$[\![A \neq \{\};\ B \neq \{\}]\!] \Longrightarrow$
$(\sqcap\ p \in A.\ P\ p) \sqcap (\sqcap\ p \in B.\ P\ p) = (\sqcap\ p \in (A \cup B)\ .\ P\ p)$›
⟨*proof*⟩

lemma *GlobalNdet-Union* :
‹$(\sqcap a \in (\bigcup i \in I.\ A\ i).\ P\ a) = \sqcap i \in \{i \in I.\ A\ i \neq \{\}\}.\ \sqcap a \in A\ i.\ P\ a$›
⟨*proof*⟩

6.4 First properties

lemma *GlobalNdet-id* [*simp*] : ‹$A \neq \{\} \Longrightarrow (\sqcap\ p \in A.\ P) = P$›
⟨*proof*⟩

lemma *GlobalNdet-unit*[*simp*] : ‹$(\sqcap\ x \in \{a\}.\ P\ x) = P\ a$›
⟨*proof*⟩

lemma *GlobalNdet-distrib-unit*:
‹$A - \{a\} \neq \{\} \Longrightarrow (\sqcap x \in \mathit{insert}\ a\ A.\ P\ x) = P\ a \sqcap (\sqcap x \in (A - \{a\}).\ P\ x)$›
⟨*proof*⟩

lemma *GlobalNdet-distrib-unit-bis* :
‹$(\sqcap x \in \mathit{insert}\ a\ A.\ P\ x) = (\mathit{if}\ A - \{a\} = \{\}\ \mathit{then}\ P\ a\ \mathit{else}\ P\ a \sqcap (\sqcap x \in (A - \{a\}).\ P\ x))$›
⟨*proof*⟩

6.5 Behaviour of *GlobalNdet* with (⊓)

lemma *GlobalNdet-Ndet*:
‹$(\sqcap a \in A.\ P\ a) \sqcap (\sqcap a \in A.\ Q\ a) = \sqcap a \in A.\ P\ a \sqcap Q\ a$›
⟨*proof*⟩

6.6 Commutativity

lemma *GlobalNdet-sets-commute*:
‹$(\sqcap a \in A. \sqcap b \in B.\ P\ a\ b) = \sqcap b \in B. \sqcap a \in A.\ P\ a\ b$›
⟨*proof*⟩

6.7 Behaviour with injectivity

lemma *inj-on-mapping-over-GlobalNdet*:
‹$\mathit{inj\text{-}on}\ f\ A \Longrightarrow (\sqcap x \in A.\ P\ x) = \sqcap x \in f\ `\ A.\ P\ (\mathit{inv\text{-}into}\ A\ f\ x)$›
⟨*proof*⟩

6.8 Cartesian product results

lemma *GlobalNdet-cartprod-σs-set-σs-set*:
‹$(\sqcap (s,\ t) \in A \times B.\ P\ (s\ @\ t)) = \sqcap u \in \{s\ @\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *GlobalNdet-cartprod-s-set-σs-set*:
‹$(\sqcap (s,\ t) \in A \times B.\ P\ (s\ \#\ t)) = \sqcap u \in \{s\ \#\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *GlobalNdet-cartprod-s-set-s-set*:
‹$(\sqcap (s,\ t) \in A \times B.\ P\ [s,\ t]) = \sqcap u \in \{[s,\ t]\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *GlobalNdet-cartprod*: ‹$(\sqcap(s,\ t) \in A \times B.\ P\ s\ t) = \sqcap s \in A. \sqcap t \in B.\ P\ s\ t$›
⟨*proof*⟩

6.9 Link with *Mndetprefix*

This is a trick to make proof of *Mndetprefix* using *GlobalNdet* as it has an easier denotational definition.

lemma *Mndetprefix-GlobalNdet*: ‹$\sqcap\ a \in A \rightarrow P\ a = \sqcap\ a \in A.\ a \rightarrow P\ a$›
⟨*proof*⟩

lemma *write0-GlobalNdet*:
‹$x \rightarrow \sqcap\ a \in A.\ P\ a = (\mathit{if}\ A = \{\}\ \mathit{then}\ x \rightarrow \mathit{STOP}\ \mathit{else} \sqcap\ a \in A.\ x \rightarrow P\ a)$›
⟨*proof*⟩

lemma *ndet-write-is-GlobalNdet-write0* :
‹$c!!a{\in}A \rightarrow P\ a = \sqcap b{\in}c\ `\ A.\ b \rightarrow P\ (\mathit{inv\text{-}into}\ A\ c\ b)$›
⟨*proof*⟩

lemma *ndet-write-is-GlobalNdet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow c!!a{\in}A \rightarrow P\ a = \sqcap a{\in}A.\ c!a \rightarrow P\ a$›
⟨*proof*⟩

lemma *GlobalNdet-Mprefix-distr*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ a \in A.\ \Box\ b \in B \rightarrow P\ a\ b) = \Box\ b \in B \rightarrow (\sqcap\ a \in A.\ P\ a\ b)$›
⟨*proof*⟩

lemma *GlobalNdet-Det-distrib*:
‹$(\sqcap\ a \in A.\ P\ a \Box\ Q\ a) = (\sqcap\ a \in A.\ P\ a) \Box (\sqcap\ a \in A.\ Q\ a)$›
if ‹$\exists\ Q'\ b.\ \forall\ a.\ Q\ a = (b \rightarrow Q'\ a)$›
⟨*proof*⟩

6.10 Continuity

lemma *mono-GlobalNdet* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq \sqcap a \in A.\ Q\ a$›
⟨*proof*⟩

lemma *chain-GlobalNdet* : ‹$\mathit{chain}\ Y \Longrightarrow \mathit{chain}\ (\lambda i.\ \sqcap a \in A.\ Y\ i\ a)$›
⟨*proof*⟩

lemma *GlobalNdet-cont* [*simp*] : ‹⟦*finite A*; ⋀*a*. *a* ∈ *A* ⟹ *cont* (*P a*)⟧ ⟹ *cont* (λ*y*. ⊓ *z*∈*A*. *P z y*)›
⟨*proof*⟩

Chapter 7

The Sequential Composition

7.1 Definition

lift-definition *Seq* :: ‹[('*a*, '*r*) *process*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$] ⇒ ('*a*, '*r*) *process*$_{ptick}$› (**infixl** ‹;› 74)
is ‹λ*P Q*. ({(*t*, *X*) |*t X*. (*t*, *X* ∪ *range tick*) ∈ $\mathcal{F}$ *P* ∧ *tF t*} ∪
{(*t* @ *u*, *X*) |*t u r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ (*u*, *X*) ∈ $\mathcal{F}$ *Q*} ∪
{(*t*, *X*). *t* ∈ $\mathcal{D}$ *P*},
$\mathcal{D}$ *P* ∪ {*t* @ *u* |*t u r*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ *u* ∈ $\mathcal{D}$ *Q*})›
⟨*proof*⟩

7.2 The Projections

lemma *F-Seq* : ‹$\mathcal{F}$ (*P* ; *Q*) = {(*t*, *X*) |*t X*. (*t*, *X* ∪ *range tick*) ∈ $\mathcal{F}$ *P* ∧ *tF t*} ∪
{(*t* @ *u*, *X*) |*t u r X*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ (*u*, *X*) ∈ $\mathcal{F}$ *Q*} ∪
{(*t*, *X*). *t* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemma *D-Seq* : ‹$\mathcal{D}$ (*P* ; *Q*) = $\mathcal{D}$ *P* ∪ {*t* @ *u* |*t u r*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ *u* ∈ $\mathcal{D}$ *Q*}›
⟨*proof*⟩

lemma *T-Seq-bis* :
‹$\mathcal{T}$ (*P* ; *Q*) = {*t*. (*t*, *range tick*) ∈ $\mathcal{F}$ *P* ∧ *tF t*} ∪
{*t* @ *u* |*t u r*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q*} ∪
$\mathcal{D}$ *P*›
⟨*proof*⟩

lemma *T-Seq* :
‹$\mathcal{T}$ (*P* ; *Q*) = {*t* ∈ $\mathcal{T}$ *P*. *tF t*} ∪ {*t* @ *u* |*t u r*. *t* @ [✓(*r*)] ∈ $\mathcal{T}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q*} ∪
$\mathcal{D}$ *P*›
— Often easier to use

⟨*proof*⟩

lemmas *Seq-projs* = *F-Seq D-Seq T-Seq*

7.3 Continuity Rule

lemma *mono-Seq* : ‹$P \sqsubseteq Q \Longrightarrow R \sqsubseteq S \Longrightarrow P \mathbin{;} R \sqsubseteq Q \mathbin{;} S$› **for** P :: ‹('*a*, '*r*) *process*$_{ptick}$›
⟨*proof*⟩

context begin

private lemma *chain-Seq-left*: ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i \mathbin{;} S)$›
⟨*proof*⟩ **lemma** *chain-Seq-right*: ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ S \mathbin{;} Y\ i)$›
⟨*proof*⟩ **lemma** *cont-left-prem-Seq* :
‹$(\bigsqcup i.\ Y\ i) \mathbin{;} S = (\bigsqcup i.\ Y\ i \mathbin{;} S)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain* Y›
⟨*proof*⟩ **lemma** *cont-right-prem-Seq* :
‹$S \mathbin{;} (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ S \mathbin{;} Y\ i)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹*chain* Y›
⟨*proof*⟩

lemma *Seq-cont* [*simp*] : ‹*cont* $(\lambda x.\ f\ x \mathbin{;} g\ x)$› **if** ‹*cont* f› **and** ‹*cont* g›
⟨*proof*⟩

end

Chapter 8

The Synchronization Product

8.1 Basic Concepts

fun *setinterleaving* :: ‹('*a*, '*r*) $trace_{ptick}$ × ('*a*, '*r*) $refusal_{ptick}$ × ('*a*, '*r*) $trace_{ptick}$ ⇒ ('*a*, '*r*) $trace_{ptick}$ *set*›
where

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

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

fun *setinterleavingList* :: ‹('*a*, '*r*) $trace_{ptick}$ × ('*a*, '*r*) $event_{ptick}$ *set* × ('*a*, '*r*) $trace_{ptick}$ ⇒ ('*a*, '*r*) $trace_{ptick}$ *list*›
where

si-empty1l: ‹*setinterleavingList* ([], *X*, []) = [[]]›
| *si-empty2l*: ‹*setinterleavingList* ([], *X*, (*y* # *t*)) =
(*if* *y* ∈ *X*
then []
else [*y* # *z*. *z* ← *setinterleavingList* ([], *X*, *t*)])›
| *si-empty3l*: ‹*setinterleavingList* (*x* # *s*, *X*, []) =
(*if* *x* ∈ *X*
then []
else [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, [])])›

| *si-neql* : ‹*setinterleavingList* ((*x* # *s*), *X*, (*y* # *t*)) =
(*if* *x* ∈ *X*
then if *y* ∈ *X*
then if *x* = *y*
then [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *t*)]
else []
else [*y* # *z*. *z* ← *setinterleavingList* (*x* # *s*, *X*, *t*)]
else if *y* ∉ *X*
then [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *y* # *t*)] @
[*y* # *z*. *z* ← *setinterleavingList* (*x* # *s*, *X*, *t*)]
else [*x* # *z*. *z* ← *setinterleavingList* (*s*, *X*, *y* # *t*)])›

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

lemma *setinterleaving-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* ∈ *setinterleaving*(*s*, *X*, *t*))

8.2 Consequences

lemma *emptyLeftProperty*: ‹*s setinterleaves* (([], *t* :: (′*a*, ′*r*) *trace*$_{ptick}$), *A*) ⟹ *s* = *t*›
⟨*proof*⟩

lemma *emptyRightProperty*: ‹*s setinterleaves* ((*t*, []), *A*) ⟹ *s* = *t*›
⟨*proof*⟩

lemma *emptyLeftSelf*: ‹∀ *t1*. *t1* ∈ *set* *t* ⟶ *t1* ∉ *A* ⟹ *t setinterleaves* (([], *t*), *A*)›
⟨*proof*⟩

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

lemma *emptyLeftNonSync*: ‹*s setinterleaves* ((*[]*, *t*), *A*) ⟹ ∀ *a* ∈ *set t*. *a* ∉ *A*›
⟨*proof*⟩

lemma *ftf-Sync1*: ‹⟦*a* ∉ *set u*; *a* ∉ *set t*; *s setinterleaves* ((*t*, *u*), *A*)⟧ ⟹ *a* ∉ *set s*›
⟨*proof*⟩

lemma *addNonSync*:
‹*sa setinterleaves* ((*t*, *u*), *A*) ⟹ *y1* ∉ *A* ⟹
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u*), *A*) ∧ (*sa* @ [*y1*]) *setinterleaves* ((*t*, *u* @ [*y1*]), *A*)›
⟨*proof*⟩

lemma *addSync*:‹*sa setinterleaves* ((*t*, *u*), *A*) ⟹ *y1* ∈ *A* ⟹
(*sa* @ [*y1*]) *setinterleaves* ((*t* @ [*y1*], *u* @ [*y1*]), *A*)›
⟨*proof*⟩

lemma *doubleReverse*: ‹*s1 setinterleaves* ((*t*, *u*), *A*) ⟹ *rev s1 setinterleaves* ((*rev t*, *rev u*), *A*)›
⟨*proof*⟩

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 $((t, u), A) \Longrightarrow hd\ t \in A \Longrightarrow hd\ u \in A$
$\Longrightarrow hd\ t = hd\ u \wedge hd\ a = hd\ t \wedge (tl\ a)$ *setinterleaves* $((tl\ t, tl\ u), A)$
⟨*proof*⟩

lemma *SyncSingleHeadAdd*:
a setinterleaves $((t, u), A) \Longrightarrow h \notin A$
$\Longrightarrow (h \# a)$ *setinterleaves* $((h \# t, u), A)$
⟨*proof*⟩

lemma *TickLeftSync*:
‹$[\![tick\ r \in A$; *front-tickFree* t; s *setinterleaves* $(([tick\ r], t :: ('a, 'r)\ trace_{ptick}), A)]\!] \Longrightarrow s = t \wedge last\ t = tick\ r$›
⟨*proof*⟩

lemma *EmptyLeftSync*: ‹s *setinterleaves* $(([], t), A) \Longrightarrow s = t \wedge set\ t \cap A = \{\}$›
⟨*proof*⟩

lemma *EmptyRightSync*: ‹s *setinterleaves* $((t, []), A) \Longrightarrow s = t \wedge set\ t \cap A = \{\}$›
⟨*proof*⟩

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

lemma *ftf-Sync32*:
assumes ‹$t = t1\ @\ [tick\ r]$› **and** ‹$u = u1\ @\ [tick\ r]$›
and ‹s *setinterleaves* $((t, u), range\ tick \cup ev\ `\ A)$›
shows ‹$\exists\ s1.\ s1$ *setinterleaves* $((t1, u1), range\ tick \cup ev\ `\ A) \wedge s = s1\ @\ [tick\ r]$›
⟨*proof*⟩

lemma *SyncWithTick-imp-NTF*:
assumes ‹$(s\ @\ [tick\ r])$ *setinterleaves* $((t, u), range\ tick \cup ev\ `\ A)$›
and ‹*front-tickFree* t› **and** ‹*front-tickFree* u›
shows ‹$\exists\ t1\ u1.\ t = t1\ @\ [tick\ r] \wedge u = u1\ @\ [tick\ r] \wedge s$ *setinterleaves* $((t1, u1), range\ tick \cup ev\ `\ A)$›

⟨*proof*⟩

lemma *suffix-tick-le-ftf-imp-eq*: ‹*front-tickFree t* ⟹ *s* @ [*tick r*] ≤ *t* ⟹ *s* @ [*tick r*] = *t*›
⟨*proof*⟩

lemma *synPrefix*: ‹(*s* @ *t*) *setinterleaves* ((*ta*, *u*), *A*) ⟹ ∃ *t1 u1*. *t1* ≤ *ta* ∧ *u1* ≤ *u* ∧ *s setinterleaves* ((*t1*, *u1*), *A*)›
⟨*proof*⟩

lemma *interleave-less-left* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *t1* < *t* ⟹
∃ *u1 s1*. *u1* ≤ *u* ∧ *s1* < *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
⟨*proof*⟩

lemma *interleave-less-right* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *u1* < *u* ⟹
∃ *t1 s1*. *t1* ≤ *t* ∧ *s1* < *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
⟨*proof*⟩

lemma *interleave-le-left* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *t1* ≤ *t* ⟹
∃ *u1 s1*. *u1* ≤ *u* ∧ *s1* ≤ *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
⟨*proof*⟩

lemma *interleave-le-right* :
‹*s setinterleaves* ((*t*, *u*), *A*) ⟹ *u1* ≤ *u* ⟹
∃ *t1 s1*. *t1* ≤ *t* ∧ *s1* ≤ *s* ∧ *s1 setinterleaves* ((*t1*, *u1*), *A*)›
⟨*proof*⟩

lemma *SyncWithTick-imp-NTF1*:
assumes ‹(*s* @ [*tick r*]) *setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
and ‹*t* ∈ 𝒯 *P*› **and** ‹*u* ∈ 𝒯 *Q*›
shows ‹∃ *t u X-P X-Q*. (*t*, *X-P*) ∈ ℱ *P* ∧ (*u*, *X-Q*) ∈ ℱ *Q* ∧
s setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
X − {*tick r*} = (*X-P* ∪ *X-Q*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X-P* ∩ *X-Q*›
⟨*proof*⟩

lemma *interleave-size*:
‹*s setinterleaves* ((*t*, *u*), *C*) ⟹ *length s* = *length t* + *length u* − *length* (*filter* (λ*x*. *x* ∈ *C*) *t*)›
⟨*proof*⟩

lemma *interleave-eq-size*:
‹*s setinterleaves* ((*t*, *u*), *C*) ⟹ *s'* *setinterleaves* ((*t*, *u*), *C*) ⟹ *length s* = *length s'*›
⟨*proof*⟩

lemma *ftf-Sync*:
assumes ‹*front-tickFree t*› **and** ‹*front-tickFree u*›
and ‹*s setinterleaves* ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*)›
shows ‹*front-tickFree s*›
⟨*proof*⟩

8.3 The Sync Operator Definition

lift-definition *Sync* :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a set*, ('*a*, '*r*) *process*$_{ptick}$] ⇒ ('*a*, '*r*) *process*$_{ptick}$›
(‹(*3*(- ⟦-⟧/ -))› [*70*, *0*, *71*] *70*)
is ‹λ*P A Q*. ({(*s*, *R*). ∃ *t u X Y*. (*t*, *X*) ∈ $\mathcal{F}$ *P* ∧ (*u*, *Y*) ∈ $\mathcal{F}$ *Q* ∧
s setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
R = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X* ∩ *Y*} ∪
{(*s*, *R*). ∃ *t u r v*. *ftF v* ∧ (*tF r* ∨ *v* = []) ∧ *s* = *r* @ *v* ∧
r setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
(*t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t* ∈ $\mathcal{D}$ *Q* ∧ *u* ∈ $\mathcal{T}$ *P*)},
{*s*. ∃ *t u r v*. *ftF v* ∧ (*tF r* ∨ *v* = []) ∧ *s* = *r* @ *v* ∧
r setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
(*t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t* ∈ $\mathcal{D}$ *Q* ∧ *u* ∈ $\mathcal{T}$ *P*)})›
⟨*proof*⟩

8.4 The Projections

lemma *F-Sync* :
‹$\mathcal{F}$ (*P* ⟦ *A* ⟧ *Q*) =
{(*s*, *R*). ∃ *t u X Y*. (*t*, *X*) ∈ $\mathcal{F}$ *P* ∧ (*u*, *Y*) ∈ $\mathcal{F}$ *Q* ∧
s setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
R = (*X* ∪ *Y*) ∩ (*range tick* ∪ *ev* ‘ *A*) ∪ *X* ∩ *Y*} ∪
{(*s*, *R*). ∃ *t u r v*. *front-tickFree v* ∧ (*tickFree r* ∨ *v* = []) ∧ *s* = *r* @ *v* ∧
r setinterleaves ((*t*, *u*), *range tick* ∪ *ev* ‘ *A*) ∧
(*t* ∈ $\mathcal{D}$ *P* ∧ *u* ∈ $\mathcal{T}$ *Q* ∨ *t* ∈ $\mathcal{D}$ *Q* ∧ *u* ∈ $\mathcal{T}$ *P*)}›
⟨*proof*⟩

lemma *D-Sync* :
‹$\mathcal{D}$ (*P* ⟦ *A* ⟧ *Q*) =

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

⟨*proof*⟩

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

⟨*proof*⟩

lemmas *Sync-projs = F-Sync D-Sync T-Sync*

8.5 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 $P\ ||\ Q\ \equiv (P\ [\![\ \mathit{UNIV}\ ]\!]\ Q)$

lemma *setinterleaving-UNIV-iff* : $\langle s\ \mathit{setinterleaves}\ ((t,\ u),\ \mathit{UNIV}) \longleftrightarrow s = t \wedge s = u\rangle$
for $s :: \langle('a,\ 'r)\ \mathit{trace}_{ptick}\rangle$
⟨*proof*⟩

lemma *F-Par* :
$\langle \mathcal{F}\ (P\ ||\ Q) = \{(s,\ R).\ \exists X\ Y.\ (s,\ X) \in \mathcal{F}\ P \wedge (s,\ Y) \in \mathcal{F}\ Q \wedge R = X \cup Y \cup X \cap Y\} \cup$
$\{(s,\ R).\ \exists t\ v.\ \mathit{ftF}\ v \wedge (\mathit{tF}\ t \vee v = []) \wedge s = t\ @\ v \wedge$
$(t \in \mathcal{D}\ P \wedge t \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge t \in \mathcal{T}\ P)\}\rangle$

⟨*proof*⟩

lemma *D-Par* :
$\langle \mathcal{D}\ (P\ ||\ Q) = \{s.\ \exists t\ v.\ \mathit{ftF}\ v \wedge (\mathit{tF}\ t \vee v = []) \wedge s = t\ @\ v \wedge$
$(t \in \mathcal{D}\ P \wedge t \in \mathcal{T}\ Q \vee t \in \mathcal{D}\ Q \wedge t \in \mathcal{T}\ P)\}\rangle$

⟨*proof*⟩

lemma *T-Par* :
$\langle \mathcal{T}\ (P\ ||\ Q) = \mathcal{T}\ P \cap \mathcal{T}\ Q \cup \{t\ @\ v|\ t\ v.\ \mathit{ftF}\ v \wedge (\mathit{tF}\ t \vee v = []) \wedge$
$t \in \mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P\}\rangle$

⟨*proof*⟩

lemmas *Par-projs = F-Par D-Par T-Par*

lemma *superset-T-Inter* : ‹$\{t.\ tF\ t \wedge t \in \mathcal{T}\ P \cup \mathcal{T}\ Q\} \subseteq \mathcal{T}\ (P\ |||\ Q)$›
⟨*proof*⟩

lemma *superset-D-Inter* : ‹$\mathcal{D}\ P \cup \mathcal{D}\ Q \subseteq \mathcal{D}\ (P\ |||\ Q)$›
⟨*proof*⟩

8.6 Continuity Rule

lemma *Sync-commute*: ‹$P\ [\![S]\!]\ Q = Q\ [\![S]\!]\ P$›
— This will simplify the following proofs.
⟨*proof*⟩

lemma *mono-Sync* : ‹$P \sqsubseteq P' \Longrightarrow Q \sqsubseteq Q' \Longrightarrow P\ [\![A]\!]\ Q \sqsubseteq P'\ [\![A]\!]\ Q'$›
for P :: ‹$('a, 'r)\ process_{ptick}$›
⟨*proof*⟩

lemma *chain-Sync-left* : ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ Y\ i\ [\![A]\!]\ S)$›
and *chain-Sync-right* : ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda i.\ S\ [\![A]\!]\ Y\ i)$›
⟨*proof*⟩

lemma *finite-interleaves*: ‹*finite* $\{(t,\ u).\ (s :: ('a, 'r)\ trace_{ptick})\ setinterleaves\ ((t, u),\ A)\}$›
⟨*proof*⟩

lemma *finite-interleaves-Sync*:
‹*finite* $\{(t,\ u,\ r).\ r\ setinterleaves\ ((t,\ u),\ range\ tick \cup ev\ `\ A) \wedge$
$(\exists v.\ s = r\ @\ v \wedge front\text{-}tickFree\ v \wedge (tickFree\ r \vee v = []))\}$›
(**is** ‹*finite ?A*›)
⟨*proof*⟩

lemma *Cont-Sync-prem*:
‹$(\bigsqcup\ i.\ Y\ i)\ [\![A]\!]\ Q = (\bigsqcup\ i.\ Y\ i\ [\![A]\!]\ Q)$› **if** *chain*: ‹*chain* Y›
⟨*proof*⟩

lemma *Sync-cont*[*simp*]: ‹*cont* $(\lambda x.\ f\ x\ [\![A]\!]\ g\ x)$› **if** ‹*cont* f› ‹*cont* g›
⟨*proof*⟩

Chapter 9

The Renaming Operator

9.1 Some preliminaries

term ‹f $-`$ B›

definition *finitary* :: ‹($'a \Rightarrow 'b$) $\Rightarrow$ *bool*›
where ‹*finitary* $f \equiv \forall x.$ *finite*(f $-`$ $\{x\}$)›

We start with some simple results.

lemma ‹f $-`$ $\{\} = \{\}$› ⟨*proof*⟩

lemma ‹$X \subseteq Y \Longrightarrow f$ $-`$ $X \subseteq f$ $-`$ Y› ⟨*proof*⟩

lemma ‹f $-`(X \cup Y) = f$ $-`$ $X \cup f$ $-`$ Y› ⟨*proof*⟩

lemma *map-event*$_{ptick}$*-eq-ev-iff* : ‹*map-event*$_{ptick}$ f g $e = ev$ $b \longleftrightarrow (\exists a.\ e = ev$ $a \ \wedge\ b = f$ $a)$›
and *map-event*$_{ptick}$*-eq-tick-iff* : ‹*map-event*$_{ptick}$ f g $e = \checkmark(s) \longleftrightarrow (\exists r.\ e = \checkmark(r)$ $\wedge\ s = g$ $r)$›
and *ev-eq-map-event*$_{ptick}$*-iff* : ‹ev $b =$ *map-event*$_{ptick}$ f g $e \longleftrightarrow (\exists a.\ e = ev$ a $\wedge\ b = f$ $a)$›
and *tick-eq-map-event*$_{ptick}$*-iff* : ‹$\checkmark(s) =$ *map-event*$_{ptick}$ f g $e \longleftrightarrow (\exists r.\ e = \checkmark(r)$ $\wedge\ s = g$ $r)$›
⟨*proof*⟩

lemma *inj-left-map-event*$_{ptick}$: ‹*inj* ($\lambda f.$ *map-event*$_{ptick}$ f g)›
⟨*proof*⟩

lemma *inj-right-map-event*$_{ptick}$: ‹*inj* (*map-event*$_{ptick}$ f)›
⟨*proof*⟩

lemma *map-event*$_{ptick}$*-tickFree* : ‹*tickFree* (*map* (*map-event*$_{ptick}$ f g) s) $\longleftrightarrow$ *tick-*

Free s›
⟨*proof*⟩

lemma *map-event*$_{ptick}$*-front-tickFree* : ‹*front-tickFree* (*map* (*map-event*$_{ptick}$ *f g*) *s*) ⟷ *front-tickFree s*›
⟨*proof*⟩

lemma *map-map-event*$_{ptick}$*-eq-tick-iff* :
‹*map* (*map-event*$_{ptick}$ *f g*) *t* = [✓(*s*)] ⟷ (∃ *r*. *s* = *g r* ∧ *t* = [✓(*r*)])›
and *tick-eq-map-map-event*$_{ptick}$*-iff* :
‹[✓(*s*)] = *map* (*map-event*$_{ptick}$ *f g*) *t* ⟷ (∃ *r*. *s* = *g r* ∧ *t* = [✓(*r*)])›
⟨*proof*⟩

9.2 The Renaming Operator Definition

Our new renaming operator does not only deal with events but also with termination.

lift-definition *Renaming* :: ‹[('*a*, '*r*) *process*$_{ptick}$, '*a* ⇒ '*b*, '*r* ⇒ '*s*] ⇒ ('*b*, '*s*) *process*$_{ptick}$›
is ‹λ*P f g*. ({(*map* (*map-event*$_{ptick}$ *f g*) *s1*, *X*)| *s1 X*. (*s1*, (*map-event*$_{ptick}$ *f g*) −‘ *X*) ∈ *F P*} ∪
{((*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*, *X*)| *s1 s2 X*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ *D P*},
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ *D P*})›
⟨*proof*⟩

Some syntactic sugar.

syntax
-Renaming :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- ⟦-⟧ ⟦-⟧› [*100*, *100*, *100*])
-Renaming-left :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- ⟦-⟧ [[]]› [*100*, *100*])
-Renaming-right :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ *updbinds* ⇒ ('*a*, '*r*) *process*$_{ptick}$› (‹- [[]] ⟦-⟧› [*100*, *100*])
syntax-consts *-Renaming* ⇌ *Renaming*
and *-Renaming-left* ⇌ *Renaming*
and *-Renaming-right* ⇌ *Renaming*
translations
-Renaming P f-updates g-updates ⇌ *CONST Renaming P* (*-Update* (*CONST id*) *f-updates*) (*-Update* (*CONST id*) *g-updates*)
-Renaming-left P f-updates ⇌ *CONST Renaming P* (*-Update* (*CONST id*) *f-updates*) (*CONST id*)
-Renaming-right P g-updates ⇌ *CONST Renaming P* (*CONST id*) (*-Update* (*CONST id*) *g-updates*)

Now we can write $P \llbracket a := b \rrbracket \llbracket c := d \rrbracket$. If we only want to rename events, or

results, we use respectively $P \llbracket a := b \rrbracket \, [\![]$ and $P \, [\![] \, \llbracket c := d \rrbracket$. Like in *HOL.Fun*, we can write this kind of expression with as many updates we want: $P \llbracket a := b,\ c := d,\ e := f,\ g := h \rrbracket \llbracket r1 := r2,\ r3 := r4 \rrbracket$. By construction we also inherit all the results about *fun-upd*, for example $?z \neq ?x \Longrightarrow (?f(?x := ?y))\ ?z = ?f\ ?z$

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

$?a \neq ?c \Longrightarrow ?m(?a := ?b,\ ?c := ?d) = ?m(?c := ?d,\ ?a := ?b)$.

lemma ‹$P \llbracket x := y,\ x := z \rrbracket \, [\![] = P \llbracket x := z \rrbracket \, [\![]$› ⟨*proof*⟩

lemma ‹$a \neq c \Longrightarrow P \llbracket a := b,\ c := d \rrbracket \llbracket r1 := r2 \rrbracket = P \llbracket c := d,\ a := b \rrbracket \llbracket r1 := r2 \rrbracket$› ⟨*proof*⟩

9.3 The Projections

lemma *F-Renaming*: ‹$\mathcal{F}$ (*Renaming P f g*) =
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*, *X*)| *s1 X*. (*s1*, (*map-event*$_{ptick}$ *f g*) −‘ *X*) ∈ $\mathcal{F}$ *P*} ∪
{((*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*, *X*)| *s1 s2 X*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemma *D-Renaming*: ‹$\mathcal{D}$ (*Renaming P f g*) =
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemma *T-Renaming*: ‹$\mathcal{T}$ (*Renaming P f g*) =
{ *map* (*map-event*$_{ptick}$ *f g*) *s1*| *s1*. *s1* ∈ $\mathcal{T}$ *P*} ∪
{(*map* (*map-event*$_{ptick}$ *f g*) *s1*) @ *s2*| *s1 s2*. *tickFree s1* ∧ *front-tickFree s2* ∧ *s1* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemmas *Renaming-projs* = *F-Renaming D-Renaming T-Renaming*

9.4 Continuity Rule

9.4.1 Monotonicity of *Renaming*.

lemma *mono-Renaming* : ‹*Renaming P f g* ⊑ *Renaming Q f g*› **if** ‹*P* ⊑ *Q*›
⟨*proof*⟩

lemma ‹{*ev c* |*c*. *f c* = *a*} = *ev*‘{*c* . *f c* = *a*}› ⟨*proof*⟩

lemma *vimage-map-event*$_{ptick}$*-subset-vimage*: ‹*map-event*$_{ptick}$ *f g* −‘ (*ev* ‘ *A*) ⊆ *range tick* ∪ (*ev* ‘ (*f* −‘ *A*))›

and *vimage-subset-vimage-map-event$_{ptick}$*: ‹$(ev\ `\ (f\ -`\ A)) \subseteq (map\text{-}event_{ptick}\ f\ g\ -`\ (ev\ `\ A)) - range\ tick$›
⟨*proof*⟩

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

lemma *inj-imp-finitary*[*simp*] : ‹*inj f* ⟹ *finitary f*› ⟨*proof*⟩

lemma *finitary-comp-iff* : ‹*finitary g* ⟹ *finitary* (*g o f*) ⟷ *finitary f*›
⟨*proof*⟩

lemma *finitary-updated-iff*[*simp*] : ‹*finitary* (*f* (*a* := *b*)) ⟷ *finitary f*›
⟨*proof*⟩

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

lemma ‹*finitary f* ⟷ *finitary* (*f* (*a* := *b*, *c* := *d*, *e*:= *g*, *h* := *i*))› ⟨*proof*⟩

lemma *Cont-RenH2*:
‹*finitary* ((*map-event$_{ptick}$ f g*) :: (′*a*, ′*r*) *event$_{ptick}$* ⇒ (′*b*, ′*s*) *event$_{ptick}$*) ⟷ *finitary f* ∧ *finitary g*›
⟨*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* ⟹ *finitary g* ⟹ *finite* (⋃ *t* ∈ {*t*. *t* ≤ *s*}. {*s*. *t*=*map* (*map-event$_{ptick}$ f g*) *s*})›
⟨*proof*⟩

lemma *Cont-RenH6*: ‹{*t*. ∃ *t2*. *tickFree t* ∧ *front-tickFree t2* ∧ *x* = *map* (*map-event$_{ptick}$ f g*) *t* @ *t2*}
⊆ {*y*. ∃ *xa*. *xa* ∈ {*t*. *t* ≤ *x*} ∧ *y* ∈ {*s*. *xa* = *map* (*map-event$_{ptick}$ f g*) *s*}}›

⟨*proof*⟩

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

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

9.4.3 Finally, continuity !

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

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

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

⟨*proof*⟩

lemma ‹*cont P* ⟹ *cont* (λ*x*. (*P x*)⟦*a* := *b*, *c* := *d*, *e* := *f*, *g* := *a*⟧ ⟦*r1* := *r2*, *r3*:= *r4*, *r5*:= *r6*⟧)›
⟨*proof*⟩

9.5 Some nice properties

lemma *map-event*$_{ptick}$*-comp*: ‹*map-event*$_{ptick}$ (*f2* ∘ *f1*) (*g2* ∘ *g1*) = *map-event*$_{ptick}$ *f2 g2* ∘ *map-event*$_{ptick}$ *f1 g1*›
⟨*proof*⟩

lemma *Renaming-comp* : ‹*Renaming P* (*f2* ∘ *f1*) (*g2* ∘ *g1*) = *Renaming* (*Renaming P f1 g1*) *f2 g2*›
⟨*proof*⟩

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

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

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

Chapter 10

The Hiding Operator

10.1 Preliminaries : primitives and lemmas

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

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

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

lemma *trace-hide-union*: ‹*trace-hide t* (*A* ∪ *B*) = *trace-hide* (*trace-hide t A*) *B*›
⟨*proof*⟩

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

abbreviation *isInfHiddenRun* :: ‹[*nat* ⇒ (*'a*, *'r*) *event*$_{ptick}$ *list*, (*'a*, *'r*) *process*$_{ptick}$, *'a set*] ⇒ *bool*›
where ‹*isInfHiddenRun f P A* ≡ *strict-mono f* ∧ (∀ *i*. *f i* ∈ *T P*) ∧
(∀ *i*. *trace-hide* (*f i*) (*ev* ‘ *A*) = *trace-hide* (*f 0*) (*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*)›
⟨*proof*⟩

abbreviation *div-hide* :: ‹(*'a*, *'r*) *process*$_{ptick}$ ⇒ *'a set* ⇒ (*'a*, *'r*) *event*$_{ptick}$ *list set*›
where ‹*div-hide P A* ≡ {*s*. ∃ *t u*. *ftF u* ∧ *tF t* ∧
s = *trace-hide t* (*ev* ‘ *A*) @ *u* ∧

$(t \in \mathcal{D}\ P \vee (\exists f.\ isInfHiddenRun\ f\ P\ A \wedge t \in range\ f))\}$›

lemma *inf-hidden*:
‹$\exists f.\ isInfHiddenRun\ f\ P\ A \wedge s \in range\ f$›
if ‹$\forall t.\ trace\text{-}hide\ t\ (ev\ `\ A) = trace\text{-}hide\ s\ (ev\ `\ A) \longrightarrow (t,\ ev\ `\ A) \notin \mathcal{F}\ P$› **and** ‹$s \in \mathcal{T}\ P$›
⟨*proof*⟩

lemma *trace-hide-append* :
‹$s\ @\ t = trace\text{-}hide\ u\ (ev\ `\ A) \Longrightarrow$
$\exists s'\ t'.\ u = s'\ @\ t' \wedge s = trace\text{-}hide\ s'\ (ev\ `\ A) \wedge t = trace\text{-}hide\ t'\ (ev\ `\ A)$›
⟨*proof*⟩

10.2 The Hiding Operator Definition

lift-definition *Hiding* :: ‹$[('a,\ 'r)\ process_{ptick}, 'a\ set] \Rightarrow ('a,\ 'r)\ process_{ptick}$› (**infixl** ‹$\setminus$› *69*)
is ‹$\lambda P\ A.\ (\{(s,\ X).\ \exists t.\ s = trace\text{-}hide\ t\ (ev\ `\ A) \wedge (t,\ X \cup ev\ `\ A) \in \mathcal{F}\ P\} \cup$
$\{(s,\ X).\ s \in div\text{-}hide\ P\ A\},$
$div\text{-}hide\ P\ A)$›
⟨*proof*⟩

10.3 Projections

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

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

lemma *T-Hiding*: ‹$\mathcal{T}\ (P \setminus A) = \{trace\text{-}hide\ t\ (ev\ `\ A)\ |t.\ (t,\ ev\ `\ A) \in \mathcal{F}\ P\} \cup div\text{-}hide\ P\ A$›
⟨*proof*⟩

lemma *Hiding-empty*: ‹$P \setminus \{\} = P$›
⟨*proof*⟩

lemma *mem-D-imp-mem-D-Hiding*: ‹$trace\text{-}hide\ t\ (ev\ `\ A) \in \mathcal{D}\ (P \setminus A)$› **if** ‹$t \in \mathcal{D}\ P$›
⟨*proof*⟩

lemma *mem-T-imp-mem-T-Hiding*: ‹$trace\text{-}hide\ t\ (ev\ `\ A) \in \mathcal{T}\ (P \setminus A)$› **if** ‹$t \in \mathcal{T}\ P$›

⟨*proof*⟩

10.4 Continuity Rule

lemma *mono-Hiding* : ‹$(P \setminus A) \sqsubseteq (Q \setminus A)$› **if** ‹$(P::('a, 'r)\ process_{ptick}) \sqsubseteq Q$›
⟨*proof*⟩

lemma *chain-Hiding* : ‹*chain* $Y \Longrightarrow$ *chain* $(\lambda\ i.\ Y\ i \setminus A)$›
⟨*proof*⟩

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

lemma *div-Hiding-lub* :
‹*finite* $(A::'a\ set) \Longrightarrow$ *chain* $Y \Longrightarrow \mathcal{D}\ (\bigsqcup\ i.\ (Y\ i \setminus A)) \subseteq \mathcal{D}\ ((\bigsqcup\ i.\ Y\ i) \setminus A)$›
for Y :: ‹$nat \Rightarrow ('a, 'r)\ process_{ptick}$›
⟨*proof*⟩

lemma *Cont-Hiding-prem* : ‹$(\bigsqcup\ i.\ Y\ i) \setminus A = (\bigsqcup\ i.\ (Y\ i \setminus A))$› **if** ‹*finite* A›
‹*chain* Y›
⟨*proof*⟩

lemma *Hiding-cont*[*simp*]: ‹*cont* $(\lambda a.\ f\ a \setminus A)$› **if** ‹*finite* A› **and** ‹*cont* f›
⟨*proof*⟩

lemma *length-strict-mono*: ‹*strict-mono* $f \Longrightarrow i +$ *length* $(f\ 0) \leq$ *length* $(f\ i)$›
for f :: ‹$nat \Rightarrow 'a\ list$›
⟨*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* $(f::nat \Rightarrow ('a, 'r)\ event_{ptick}\ list)$ **and** $\forall\, i.\ f\ i \leq a$

shows $\exists\, i.\ \forall\, j{\geq}i.\ f\ j = f\ i$
⟨*proof*⟩

We can actually optimize the divergences of the $(\setminus)$ operator.

lemma *mono-take* : ‹$s \leq t \Longrightarrow$ *take n s* $\leq$ *take n t*›
⟨*proof*⟩

lemma *shift-isInfHiddenRun* : ‹$\exists f.$ *isInfHiddenRun f P A* $\wedge$ $t = f\ 0$›
if ‹*isInfHiddenRun f P A* $\wedge$ $t \in$ *range f*›
⟨*proof*⟩

lemma *uniformize-length-isInfHiddenRun* :
assumes $*$: ‹*isInfHiddenRun f P A*› ‹$t = f\ 0$›
defines ‹$g \equiv \lambda i.$ *take* $(i +$ *length t*$)$ $(f\ i)$›
shows ‹*isInfHiddenRun g P A* $\wedge$ $(\forall\, i.$ *length* $(g\ i) = i +$ *length t*$) \wedge t = g\ 0$›
⟨*proof*⟩

abbreviation *isInfHiddenRun-optimized* ::
‹[*nat* ⇒ $('a,\ 'r)$ $event_{ptick}$ *list*, $('a,\ 'r)$ $process_{ptick}$, $'a$ *set*, $('a,\ 'r)$ $trace_{ptick}$] ⇒ *bool*›
where ‹*isInfHiddenRun-optimized f P A t* ≡
strict-mono f $\wedge$ $(\forall\, i.\ f\ i \in \mathcal{T}\ P) \wedge (\forall\, i.\ f\ i \notin \mathcal{D}\ P) \wedge$
$(\forall\, i.$ *trace-hide* $(f\ i)$ $(ev\ `\ A) =$ *trace-hide* $(f\ 0)$ $(ev\ `\ A)) \wedge$
$(\forall\, i.$ *length* $(f\ i) =$ *length t*$) \wedge t = f\ 0$›

abbreviation *div-hide-optimized* :: ‹$('a,\ 'r)$ $process_{ptick}$ ⇒ $'a$ *set* ⇒ $('a,\ 'r)$ $event_{ptick}$ *list set*›
where ‹*div-hide-optimized P A* ≡
$\{s.\ \exists\, t\ u.$ *ftF u* $\wedge$ *tF t* $\wedge$
s = *trace-hide t* $(ev\ `\ A)$ @ *u* $\wedge$
$(t \in \mathcal{D}\ P \vee (\exists f.$ *isInfHiddenRun-optimized f P A t*$))\}$›

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

lemma *T-Hiding-optimized* :
‹$\mathcal{T}\ (P \setminus A) = \{$*trace-hide t* $(ev\ `\ A)\ |t.\ (t,\ ev\ `\ A) \in \mathcal{F}\ P\} \cup$ *div-hide-optimized P A*›
⟨*proof*⟩

Actually, $f\ i$ can be rewritten as t @ *map* x $[0..{<}i]$ where x is the sequence such that $f\ (Suc\ i) = f\ i$ @ $[x\ i]$.

definition *seqRun* :: ‹[$('a,\ 'r)$ $trace_{ptick}$, *nat* ⇒ $('a,\ 'r)$ $event_{ptick}$] ⇒ *nat* ⇒ $('a,\ 'r)$ $trace_{ptick}$›
where ‹*seqRun t x i* ≡ *t* @ *map x* $[0..{<}i]$›

lemma *seqRun-is-rec-nat* : ‹*seqRun t x = rec-nat t* (λ*i t. t* @ [*x i*])›
⟨*proof*⟩

lemma *seqRun-0* [*simp*] : ‹*seqRun t x 0 = t*›
and *seqRun-Suc* [*simp*] : ‹*seqRun t x* (*Suc i*) = *seqRun t x i* @ [*x i*]›
and *seqRun-Nil* [*simp*] : ‹*seqRun* [] *x i* = *map x* [*0..<i*]›
and *seqRun-Cons* [*simp*] : ‹*seqRun* (*a* # *t*) *x i* = *a* # *seqRun t x i*›
⟨*proof*⟩

lemma *strict-mono-seqRun* [*simp*] : ‹*strict-mono* (*seqRun t x*)›
⟨*proof*⟩

lemma *length-seqRun* [*simp*] : ‹*length* (*seqRun t x i*) = *i* + *length t*›
⟨*proof*⟩

lemma *t-le-seqRun* [*simp*] : ‹*t* ≤ *seqRun t x i*› ⟨*proof*⟩

lemma *take-t-le-seqRun* [*simp*] : ‹*take i t* ≤ *seqRun t x j*›
⟨*proof*⟩

lemma *nth-seqRun* :
‹*j* < *i* + *length t* ⟹
seqRun t x i ! *j* = (*if j* < *length t then t* ! *j else x* (*j* − *length t*))›
⟨*proof*⟩

lemma *take-seqRun* [*simp*] :
‹*take j* (*seqRun t x i*) = (*if j* ≤ *length t then take j t else seqRun t x* (*min i* (*j* − *length t*)))›
⟨*proof*⟩

lemma *seqRun-eq-iff* [*simp*] : ‹*seqRun t x i* = *seqRun t x j* ⟷ *i* = *j*›
⟨*proof*⟩

lemma *seqRun-le-iff* [*simp*] : ‹*seqRun t x i* ≤ *seqRun t x j* ⟷ *i* ≤ *j*›
⟨*proof*⟩

lemma *seqRun-less-iff* [*simp*] : ‹*seqRun t x i* < *seqRun t x j* ⟷ *i* < *j*›
⟨*proof*⟩

lemma *trace-hide-is-Nil-iff* : ‹*trace-hide s A* = [] ⟷ *set s* ⊆ *A*›
⟨*proof*⟩

lemma *trace-hide-seqRun-eq-iff* :
‹*trace-hide* (*seqRun t x i*) *A* = *trace-hide t A* ⟷ (∀ *j*<*i. x j* ∈ *A*)›
⟨*proof*⟩

abbreviation *isInfHidden-seqRun* ::
‹⌈*nat* ⇒ ('*a*, '*r*) *event*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$, '*a set*, ('*a*, '*r*) *trace*$_{ptick}$⌉ ⇒ *bool*›
where ‹*isInfHidden-seqRun x P A t* ≡ ∀ *i*. *seqRun t x i* ∈ $\mathcal{T}$ *P* ∧ *x i* ∈ *ev* ' *A*›

abbreviation *isInfHidden-seqRun-strong* ::
‹⌈*nat* ⇒ ('*a*, '*r*) *event*$_{ptick}$, ('*a*, '*r*) *process*$_{ptick}$, '*a set*, ('*a*, '*r*) *trace*$_{ptick}$⌉ ⇒ *bool*›
where ‹*isInfHidden-seqRun-strong x P A t* ≡
∀ *i*. *seqRun t x i* ∈ $\mathcal{T}$ *P* ∧ *seqRun t x i* ∉ $\mathcal{D}$ *P* ∧ *x i* ∈ *ev* ' *A*›

abbreviation *div-hide-seqRun* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ '*a set* ⇒ ('*a*, '*r*) *event*$_{ptick}$ *list set*›
where ‹*div-hide-seqRun P A* ≡
{*s*. ∃ *t u*. *ftF u* ∧ *tF t* ∧ *s* = *trace-hide t* (*ev* ' *A*) @ *u* ∧
(*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *x*. *isInfHidden-seqRun x P A t*))}›

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

lemma *T-Hiding-seqRun* :
‹$\mathcal{T}$ (*P* \ *A*) = {*trace-hide t* (*ev* ' *A*) |*t*. (*t*, *ev* ' *A*) ∈ $\mathcal{F}$ *P*} ∪ *div-hide-seqRun P A*›
⟨*proof*⟩

lemma *F-Hiding-seqRun* :
‹$\mathcal{F}$ (*P* \ *A*) =
{(*s*, *X*). ∃ *t*. *s* = *trace-hide t* (*ev* ' *A*) ∧ (*t*, *X* ∪ *ev* ' *A*) ∈ $\mathcal{F}$ *P*} ∪
{(*s*, *X*). *s* ∈ *div-hide-seqRun P A*}›
⟨*proof*⟩

lemma *D-Hiding-seqRunI* :
‹⟦*ftF u*; *tF t*; *s* = *trace-hide t* (*ev* ' *A*) @ *u*;
t ∈ $\mathcal{D}$ *P* ∨ (∃ *x*. *isInfHidden-seqRun x P A t*)⟧ ⟹ *s* ∈ $\mathcal{D}$ (*P* \ *A*)›
⟨*proof*⟩

lemma *D-Hiding-seqRunE* :
assumes ‹*s* ∈ $\mathcal{D}$ (*P* \ *A*)›
obtains *t u* **where** ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev* ' *A*) @ *u*›
‹*t* ∈ $\mathcal{D}$ *P* ∨ (∃ *x*. *isInfHidden-seqRun-strong x P A t*)›
⟨*proof*⟩

lemma *T-Hiding-seqRunE* :
assumes ‹*s* ∈ $\mathcal{T}$ (*P* \ *A*)›
obtains *t* **where** ‹*s* = *trace-hide t* (*ev* ' *A*)› ‹(*t*, *ev* ' *A*) ∈ $\mathcal{F}$ *P*›
| *t u* **where** ‹*ftF u*› ‹*tF t*› ‹*s* = *trace-hide t* (*ev* ' *A*) @ *u*›

‹$t \in \mathcal{D}\ P \vee (\exists\, x.\ isInfHidden\text{-}seqRun\text{-}strong\ x\ P\ A\ t)$›
⟨*proof*⟩

lemma *butlast-seqRun* : ‹$butlast\ (seqRun\ t\ x\ i) = (case\ i\ of\ 0 \Rightarrow butlast\ t \mid Suc\ j \Rightarrow seqRun\ t\ x\ j)$›
⟨*proof*⟩

lemma *isInfHidden-seqRun-imp-tickFree* : ‹$isInfHidden\text{-}seqRun\ x\ P\ A\ t \Longrightarrow tF\ t$›
⟨*proof*⟩

lemma *tickFree-seqRun-iff* : ‹$tF\ (seqRun\ t\ x\ i) \longleftrightarrow tF\ t \wedge (\forall\, j{<}i.\ is\text{-}ev\ (x\ j))$›
⟨*proof*⟩

lemma *front-tickFree-seqRun-iff* :
‹$ftF\ (seqRun\ t\ x\ i) \longleftrightarrow (case\ i\ of\ 0 \Rightarrow ftF\ t \mid Suc\ j \Rightarrow tF\ t \wedge (\forall\, k{<}j.\ is\text{-}ev\ (x\ k)))$›
⟨*proof*⟩

lemmas *Hiding-seqRun-projs = F-Hiding-seqRun D-Hiding-seqRun T-Hiding-seqRun*

Chapter 11

The CSP Refinements

11.1 Definitions and first Properties

11.1.1 Definitions

definition *trace-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_T$› *50*)
where ‹$P \sqsubseteq_T Q \equiv \mathcal{T}\ Q \subseteq \mathcal{T}\ P$›

definition *failure-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_F$› *50*)
where ‹$P \sqsubseteq_F Q \equiv \mathcal{F}\ Q \subseteq \mathcal{F}\ P$›

definition *divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒ *bool*›
(**infix** ‹$\sqsubseteq_D$› *50*)
where ‹$P \sqsubseteq_D Q \equiv \mathcal{D}\ Q \subseteq \mathcal{D}\ P$›

abbreviation *failure-divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$
⇒ *bool*› (**infix** ‹$\sqsubseteq_{FD}$› *50*)
where ‹$P \sqsubseteq_{FD} Q \equiv P \leq Q$›

definition *trace-divergence-refine* :: ‹('a, 'r) $process_{ptick}$ ⇒ ('a, 'r) $process_{ptick}$ ⇒
bool› (**infix** ‹$\sqsubseteq_{DT}$› *50*)

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

lemma *failure-divergence-refine-def* : ‹$P \sqsubseteq_{FD} Q \longleftrightarrow P \sqsubseteq_F Q \wedge P \sqsubseteq_D Q$›
⟨*proof*⟩

lemmas *refine-defs* = *failure-divergence-refine-def trace-divergence-refine-def*
failure-refine-def divergence-refine-def trace-refine-def

lemma *failure-divergence-refineI* :
‹⟦⋀s. s ∈ 𝒟 Q ⟹ s ∈ 𝒟 P; ⋀s X. (s, X) ∈ ℱ Q ⟹ (s, X) ∈ ℱ P⟧ ⟹ P ⊑$_{FD}$ Q›
⟨*proof*⟩

lemma *failure-divergence-refine-optimizedI* :
‹⟦⋀s. s ∈ 𝒟 Q ⟹ s ∈ 𝒟 P; ⋀s X. (s, X) ∈ ℱ Q ⟹ 𝒟 Q ⊆ 𝒟 P ⟹ (s, X) ∈ ℱ P⟧ ⟹ P ⊑$_{FD}$ Q›
⟨*proof*⟩

lemma *trace-divergence-refineI* :
‹⟦⋀s. s ∈ 𝒟 Q ⟹ s ∈ 𝒟 P; ⋀s. s ∈ 𝒯 Q ⟹ s ∈ 𝒯 P⟧ ⟹ P ⊑$_{DT}$ Q›
⟨*proof*⟩

lemma *trace-divergence-refine-optimizedI* :
‹⟦⋀s. s ∈ 𝒟 Q ⟹ s ∈ 𝒟 P; ⋀s. s ∈ 𝒯 Q ⟹ 𝒟 Q ⊆ 𝒟 P ⟹ s ∈ 𝒯 P⟧ ⟹ P ⊑$_{DT}$ Q›
⟨*proof*⟩

11.1.2 Idempotency

lemma *idem-F*[*simp*] : ‹P ⊑$_{F}$ P›
and *idem-D*[*simp*] : ‹P ⊑$_{D}$ P›
and *idem-T*[*simp*] : ‹P ⊑$_{T}$ P›
and *idem-FD*[*simp*] : ‹P ⊑$_{FD}$ P›
and *idem-DT*[*simp*] : ‹P ⊑$_{DT}$ P›
⟨*proof*⟩

11.1.3 Some obvious refinements

lemma *BOT-leF* [*simp*] : ‹⊥ ⊑$_{F}$ Q›
and *BOT-leD* [*simp*] : ‹⊥ ⊑$_{D}$ Q›
and *BOT-leT* [*simp*] : ‹⊥ ⊑$_{T}$ Q›
and *BOT-leFD*[*simp*] : ‹⊥ ⊑$_{FD}$ Q›
and *BOT-leDT*[*simp*] : ‹⊥ ⊑$_{DT}$ Q›
⟨*proof*⟩

11.1.4 Antisymmetry

lemma *FD-antisym*: ‹P ⊑$_{FD}$ Q ⟹ Q ⊑$_{FD}$ P ⟹ P = Q› ⟨*proof*⟩

lemma *DT-antisym*: ‹P ⊑$_{DT}$ Q ⟹ Q ⊑$_{DT}$ P ⟹ P = Q›
⟨*proof*⟩

11.1.5 Transitivity

lemma *trans-F* : ‹P ⊑$_{F}$ Q ⟹ Q ⊑$_{F}$ S ⟹ P ⊑$_{F}$ S›
and *trans-D* : ‹P ⊑$_{D}$ Q ⟹ Q ⊑$_{D}$ S ⟹ P ⊑$_{D}$ S›
and *trans-T* : ‹P ⊑$_{T}$ Q ⟹ Q ⊑$_{T}$ S ⟹ P ⊑$_{T}$ S›
and *trans-FD* : ‹P ⊑$_{FD}$ Q ⟹ Q ⊑$_{FD}$ S ⟹ P ⊑$_{FD}$ S›

and *trans-DT* : ‹$P \sqsubseteq_{DT} Q \Longrightarrow Q \sqsubseteq_{DT} S \Longrightarrow P \sqsubseteq_{DT} S$›
⟨*proof*⟩

11.1.6 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*⟩

11.1.7 More obvious refinements

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

11.1.8 Admissibility

lemma *le-F-adm* [*simp*] : ‹*adm* ($\lambda x.\ u\ x \sqsubseteq_F v\ x$)› **if** ‹*cont u*› **and** ‹*monofun v*›
⟨*proof*⟩

lemma *le-T-adm* [*simp*] : ‹*adm* ($\lambda x.\ u\ x \sqsubseteq_T v\ x$)› **if** ‹*cont u*› **and** ‹*monofun v*›
⟨*proof*⟩

lemma *le-D-adm* [*simp*] : ‹*adm* ($\lambda x.\ u\ x \sqsubseteq_D v\ x$)› **if** ‹*cont u*› **and** ‹*monofun v*›
⟨*proof*⟩

declare *le-FD-adm* [*simp*]

thm *le-FD-adm le-FD-adm-cont*

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

lemmas *le-F-adm-cont*[*simp*] = *le-F-adm*[*OF - cont2mono*]
and *le-T-adm-cont*[*simp*] = *le-T-adm*[*OF - cont2mono*]
and *le-D-adm-cont*[*simp*] = *le-D-adm*[*OF - cont2mono*]
and *le-DT-adm-cont*[*simp*] = *le-DT-adm*[*OF - cont2mono*]

11.2 Monotonies

11.2.1 Straight Monotony

Non Deterministic Choice

lemma *mono-Ndet-FD* : $\langle P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{FD} P' \sqcap Q' \rangle$
and *mono-Ndet-DT* : $\langle P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{DT} P' \sqcap Q' \rangle$
and *mono-Ndet-F* : $\langle P \sqsubseteq_{F} P' \Longrightarrow Q \sqsubseteq_{F} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{F} P' \sqcap Q' \rangle$
and *mono-Ndet-D* : $\langle P \sqsubseteq_{D} P' \Longrightarrow Q \sqsubseteq_{D} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{D} P' \sqcap Q' \rangle$
and *mono-Ndet-T* : $\langle P \sqsubseteq_{T} P' \Longrightarrow Q \sqsubseteq_{T} Q' \Longrightarrow P \sqcap Q \sqsubseteq_{T} P' \sqcap Q' \rangle$
⟨*proof*⟩

lemmas *monos-Ndet* = *mono-Ndet mono-Ndet-FD mono-Ndet-DT*
mono-Ndet-F mono-Ndet-D mono-Ndet-T

Global Non Deterministic Choice

lemma *mono-GlobalNdet-FD* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{FD} \sqcap a \in A.\ Q\ a \rangle$
and *mono-GlobalNdet-DT* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{DT} \sqcap a \in A.\ Q\ a \rangle$
and *mono-GlobalNdet-F* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{F} \sqcap a \in A.\ Q\ a \rangle$
and *mono-GlobalNdet-D* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{D} \sqcap a \in A.\ Q\ a \rangle$
and *mono-GlobalNdet-T* : $\langle (\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow (\sqcap a \in A.\ P\ a) \sqsubseteq_{T} \sqcap a \in A.\ Q\ a \rangle$
⟨*proof*⟩

lemmas *monos-GlobalNdet* = *mono-GlobalNdet mono-GlobalNdet-FD mono-GlobalNdet-DT*
mono-GlobalNdet-F mono-GlobalNdet-D mono-GlobalNdet-T

lemma *GlobalNdet-FD-subset* : $\langle A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{FD} (\sqcap a \in A.\ P\ a) \rangle$
and *GlobalNdet-DT-subset* : $\langle A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{DT} (\sqcap a \in A.\ P\ a) \rangle$
and *GlobalNdet-F-subset* : $\langle A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{F} (\sqcap a \in A.\ P\ a) \rangle$
and *GlobalNdet-T-subset* : $\langle A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{T} (\sqcap a \in A.\ P\ a) \rangle$
and *GlobalNdet-D-subset* : $\langle A \subseteq B \Longrightarrow (\sqcap a \in B.\ P\ a) \sqsubseteq_{D} (\sqcap a \in A.\ P\ a) \rangle$
⟨*proof*⟩

lemmas *GlobalNdet-le-subset* =
GlobalNdet-FD-subset GlobalNdet-DT-subset
GlobalNdet-F-subset GlobalNdet-T-subset GlobalNdet-D-subset

Deterministic Choice

lemma *mono-Det-FD* : $\langle P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \,\Box\, Q \sqsubseteq_{FD} P' \,\Box\, Q' \rangle$

⟨*proof*⟩

lemma *mono-Det-D* : ‹$P \sqsubseteq_D P' \Longrightarrow Q \sqsubseteq_D Q' \Longrightarrow P \,\Box\, Q \sqsubseteq_D P' \,\Box\, Q'$›
⟨*proof*⟩

lemma *mono-Det-T* : ‹$P \sqsubseteq_T P' \Longrightarrow Q \sqsubseteq_T Q' \Longrightarrow P \,\Box\, Q \sqsubseteq_T P' \,\Box\, Q'$›
⟨*proof*⟩

corollary *mono-Det-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \,\Box\, Q \sqsubseteq_{DT} P' \,\Box\, Q'$›
⟨*proof*⟩

Deterministic choice monotony doesn't hold for ($\sqsubseteq_F$).

lemmas *monos-Det* = *mono-Det mono-Det-FD mono-Det-DT*
mono-Det-D mono-Det-T

Sliding choice

lemma *mono-Sliding-FD* : ‹$P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \rhd Q \sqsubseteq_{FD} P' \rhd Q'$›
and *mono-Sliding-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \rhd Q \sqsubseteq_{DT} P' \rhd Q'$›
and *mono-Sliding-D* : ‹$P \sqsubseteq_D P' \Longrightarrow Q \sqsubseteq_D Q' \Longrightarrow P \rhd Q \sqsubseteq_D P' \rhd Q'$›
and *mono-Sliding-T* : ‹$P \sqsubseteq_T P' \Longrightarrow Q \sqsubseteq_T Q' \Longrightarrow P \rhd Q \sqsubseteq_T P' \rhd Q'$›
⟨*proof*⟩

Sliding choice monotony doesn't hold for ($\sqsubseteq_F$).

lemma *mono-Sliding-F-right* : ‹$Q \sqsubseteq_F Q' \Longrightarrow P \rhd Q \sqsubseteq_F P \rhd Q'$›
⟨*proof*⟩

lemmas *monos-Sliding* = *mono-Sliding mono-Sliding-FD mono-Sliding-DT*
mono-Sliding-D mono-Sliding-T mono-Sliding-F-right

Synchronization

lemma *mono-Sync-FD* : ‹$[\![P \sqsubseteq_{FD} P';\ Q \sqsubseteq_{FD} Q']\!] \Longrightarrow P \,[\![S]\!]\, Q \sqsubseteq_{FD} P' \,[\![S]\!]\, Q'$›
for $P\ P'\ Q\ Q'$ :: ‹('a, 'r) $process_{ptick}$›
⟨*proof*⟩

lemma *mono-Sync-DT* : ‹$P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \,[\![S]\!]\, Q \sqsubseteq_{DT} P' \,[\![S]\!]\, Q'$›
⟨*proof*⟩

lemmas *mono-Par* = *mono-Sync*[**where** A = *UNIV*]
and *mono-Par-FD* = *mono-Sync-FD*[**where** S = *UNIV*]
and *mono-Par-DT* = *mono-Sync-DT*[**where** S = *UNIV*]
and *mono-Inter* = *mono-Sync*[**where** A = ‹{}›]
and *mono-Inter-FD* = *mono-Sync-FD*[**where** S = ‹{}›]
and *mono-Inter-DT* = *mono-Sync-DT*[**where** S = ‹{}›]

lemmas *monos-Sync = mono-Sync mono-Sync-FD mono-Sync-DT*
and *monos-Par = mono-Par mono-Par-FD mono-Par-DT*
and *monos-Inter = mono-Inter mono-Inter-FD mono-Inter-DT*

Sequential composition

lemma *mono-Seq-FD* : $\langle P \sqsubseteq_{FD} P' \Longrightarrow Q \sqsubseteq_{FD} Q' \Longrightarrow P \,;\, Q \sqsubseteq_{FD} P' \,;\, Q' \rangle$
⟨*proof*⟩

lemma *mono-Seq-DT* : $\langle P \sqsubseteq_{DT} P' \Longrightarrow Q \sqsubseteq_{DT} Q' \Longrightarrow P \,;\, Q \sqsubseteq_{DT} P' \,;\, Q' \rangle$
⟨*proof*⟩

lemma *mono-Seq-F-right* : $\langle Q \sqsubseteq_{F} Q' \Longrightarrow P \,;\, Q \sqsubseteq_{F} P \,;\, Q' \rangle$
⟨*proof*⟩

lemma *mono-Seq-D-right* : $\langle Q \sqsubseteq_{D} Q' \Longrightarrow P \,;\, Q \sqsubseteq_{D} P \,;\, Q' \rangle$
⟨*proof*⟩

lemma *mono-Seq-T-right* : $\langle Q \sqsubseteq_{T} Q' \Longrightarrow P \,;\, Q \sqsubseteq_{T} P \,;\, Q' \rangle$
⟨*proof*⟩

Left Sequence monotony doesn't hold for $(\sqsubseteq_F)$, $(\sqsubseteq_D)$ and $(\sqsubseteq_T)$.

lemmas *monos-Seq = mono-Seq mono-Seq-FD mono-Seq-DT*
mono-Seq-F-right mono-Seq-D-right mono-Seq-T-right

Renaming

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

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

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

lemmas *monos-Renaming = mono-Renaming mono-Renaming-FD mono-Renaming-DT*
mono-Renaming-D

Hiding

lemma *mono-Hiding-leT-imp-leD* : $\langle P \setminus A \sqsubseteq_{D} Q \setminus A \rangle$ **if** $\langle A \neq \{\} \rangle$ **and** $\langle P \sqsubseteq_{T} Q \rangle$
⟨*proof*⟩

lemma *mono-Hiding-F* : $\langle P \setminus A \sqsubseteq_{F} Q \setminus A \rangle$ **if** $\langle P \sqsubseteq_{F} Q \rangle$
⟨*proof*⟩

lemma *mono-Hiding-T* : $\langle P \setminus A \sqsubseteq_T Q \setminus A\rangle$ **if** $\langle P \sqsubseteq_T Q\rangle$
⟨*proof*⟩

lemma *mono-Hiding-FD* : $\langle P \setminus A \sqsubseteq_{FD} Q \setminus A\rangle$ **if** $\langle P \sqsubseteq_{FD} Q\rangle$
⟨*proof*⟩

lemma *mono-Hiding-DT* : $\langle P \setminus A \sqsubseteq_{DT} Q \setminus A\rangle$ **if** $\langle P \sqsubseteq_{DT} Q\rangle$
⟨*proof*⟩

Obviously, Hide monotony doesn't hold for $(\sqsubseteq_D)$.

lemmas *monos-Hiding* = *mono-Hiding mono-Hiding-FD mono-Hiding-DT mono-Hiding-F mono-Hiding-T*

Prefixes

lemma *mono-Mprefix-FD* : $\langle(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD} Q\ a) \Longrightarrow Mprefix\ A\ P \sqsubseteq_{FD} Mprefix\ A\ Q\rangle$
and *mono-Mprefix-DT* : $\langle(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT} Q\ a) \Longrightarrow Mprefix\ A\ P \sqsubseteq_{DT} Mprefix\ A\ Q\rangle$
and *mono-Mprefix-F* : $\langle(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F} Q\ a) \Longrightarrow Mprefix\ A\ P \sqsubseteq_{F} Mprefix\ A\ Q\rangle$
and *mono-Mprefix-T* : $\langle(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T} Q\ a) \Longrightarrow Mprefix\ A\ P \sqsubseteq_{T} Mprefix\ A\ Q\rangle$
and *mono-Mprefix-D* : $\langle(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D} Q\ a) \Longrightarrow Mprefix\ A\ P \sqsubseteq_{D} Mprefix\ A\ Q\rangle$
⟨*proof*⟩

lemmas *monos-Mprefix* = *mono-Mprefix mono-Mprefix-FD mono-Mprefix-DT mono-Mprefix-F mono-Mprefix-T mono-Mprefix-D*

corollary *mono-write0-FD* : $\langle P \sqsubseteq_{FD} Q \Longrightarrow (a \to P) \sqsubseteq_{FD} (a \to Q)\rangle$
and *mono-write0-DT* : $\langle P \sqsubseteq_{DT} Q \Longrightarrow (a \to P) \sqsubseteq_{DT} (a \to Q)\rangle$
and *mono-write0-F* : $\langle P \sqsubseteq_{F} Q \Longrightarrow (a \to P) \sqsubseteq_{F} (a \to Q)\rangle$
and *mono-write0-T* : $\langle P \sqsubseteq_{T} Q \Longrightarrow (a \to P) \sqsubseteq_{T} (a \to Q)\rangle$
and *mono-write0-D* : $\langle P \sqsubseteq_{D} Q \Longrightarrow (a \to P) \sqsubseteq_{D} (a \to Q)\rangle$
⟨*proof*⟩

lemmas *monos-write0* = *mono-write0 mono-write0-FD mono-write0-DT mono-write0-F mono-write0-T mono-write0-D*

corollary *mono-write-FD* : $\langle P \sqsubseteq_{FD} Q \Longrightarrow (c!a \to P) \sqsubseteq_{FD} (c!a \to Q)\rangle$
and *mono-write-DT* : $\langle P \sqsubseteq_{DT} Q \Longrightarrow (c!a \to P) \sqsubseteq_{DT} (c!a \to Q)\rangle$
and *mono-write-F* : $\langle P \sqsubseteq_{F} Q \Longrightarrow (c!a \to P) \sqsubseteq_{F} (c!a \to Q)\rangle$
and *mono-write-T* : $\langle P \sqsubseteq_{T} Q \Longrightarrow (c!a \to P) \sqsubseteq_{T} (c!a \to Q)\rangle$
and *mono-write-D* : $\langle P \sqsubseteq_{D} Q \Longrightarrow (c!a \to P) \sqsubseteq_{D} (c!a \to Q)\rangle$
⟨*proof*⟩

lemmas *monos-write = mono-write mono-write-FD mono-write-DT mono-write-F mono-write-T mono-write-D*

corollary *mono-read-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD}\ Q\ a) \Longrightarrow (c?a{\in}A \to P\ a) \sqsubseteq_{FD} (c?a{\in}A \to Q\ a)$›
and *mono-read-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT}\ Q\ a) \Longrightarrow (c?a{\in}A \to P\ a) \sqsubseteq_{DT} (c?a{\in}A \to Q\ a)$›
and *mono-read-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F}\ Q\ a) \Longrightarrow (c?a{\in}A \to P\ a) \sqsubseteq_{F} (c?a{\in}A \to Q\ a)$›
and *mono-read-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T}\ Q\ a) \Longrightarrow (c?a{\in}A \to P\ a) \sqsubseteq_{T} (c?a{\in}A \to Q\ a)$›
and *mono-read-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D}\ Q\ a) \Longrightarrow (c?a{\in}A \to P\ a) \sqsubseteq_{D} (c?a{\in}A \to Q\ a)$›
⟨*proof*⟩

lemmas *monos-read = mono-read mono-read-FD mono-read-DT mono-read-F mono-read-T mono-read-D*

lemma *mono-Mndetprefix-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD}\ Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{FD}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT}\ Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{DT}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F}\ Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{F}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T}\ Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{T}$ *Mndetprefix A Q*›
and *mono-Mndetprefix-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D}\ Q\ a) \Longrightarrow$ *Mndetprefix A P* $\sqsubseteq_{D}$ *Mndetprefix A Q*›
⟨*proof*⟩

lemmas *monos-Mndetprefix = mono-Mndetprefix mono-Mndetprefix-FD mono-Mndetprefix-DT mono-Mndetprefix-F mono-Mndetprefix-T mono-Mndetprefix-D*

corollary *mono-ndet-write-FD* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{FD}\ Q\ a) \Longrightarrow (c!!a{\in}A \to P\ a) \sqsubseteq_{FD} (c!!a{\in}A \to Q\ a)$›
and *mono-ndet-write-DT* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{DT}\ Q\ a) \Longrightarrow (c!!a{\in}A \to P\ a) \sqsubseteq_{DT} (c!!a{\in}A \to Q\ a)$›
and *mono-ndet-write-F* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{F}\ Q\ a) \Longrightarrow (c!!a{\in}A \to P\ a) \sqsubseteq_{F} (c!!a{\in}A \to Q\ a)$›
and *mono-ndet-write-T* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{T}\ Q\ a) \Longrightarrow (c!!a{\in}A \to P\ a) \sqsubseteq_{T} (c!!a{\in}A \to Q\ a)$›
and *mono-ndet-write-D* : ‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a \sqsubseteq_{D}\ Q\ a) \Longrightarrow (c!!a{\in}A \to P\ a) \sqsubseteq_{D} (c!!a{\in}A \to Q\ a)$›
⟨*proof*⟩

lemmas *monos-ndet-write = mono-ndet-write mono-ndet-write-FD mono-ndet-write-DT*

mono-ndet-write-F mono-ndet-write-T mono-ndet-write-D

lemma *Mndetprefix-FD-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{FD}$ *Mprefix A P*›
and *Mndetprefix-DT-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{DT}$ *Mprefix A P*›
and *Mndetprefix-F-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{F}$ *Mprefix A P*›
and *Mndetprefix-T-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{T}$ *Mprefix A P*›
and *Mndetprefix-D-Mprefix* : ‹*Mndetprefix A P* $\sqsubseteq_{D}$ *Mprefix A P*›
⟨*proof*⟩

lemmas *Mndetprefix-le-Mprefix* =
Mndetprefix-FD-Mprefix Mndetprefix-DT-Mprefix
Mndetprefix-F-Mprefix Mndetprefix-T-Mprefix Mndetprefix-D-Mprefix

corollary *ndet-write-FD-read* : ‹*ndet-write c A P* $\sqsubseteq_{FD}$ *read c A P*›
and *ndet-write-DT-read* : ‹*ndet-write c A P* $\sqsubseteq_{DT}$ *read c A P*›
and *ndet-write-F-read* : ‹*ndet-write c A P* $\sqsubseteq_{F}$ *read c A P*›
and *ndet-write-T-read* : ‹*ndet-write c A P* $\sqsubseteq_{T}$ *read c A P*›
and *ndet-write-D-read* : ‹*ndet-write c A P* $\sqsubseteq_{D}$ *read c A P*›
⟨*proof*⟩

lemmas *ndet-write-le-read* =
ndet-write-FD-read ndet-write-DT-read
ndet-write-F-read ndet-write-T-read ndet-write-D-read

lemma *Mndetprefix-FD-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{FD}$ *Mndetprefix A P*›
and *Mndetprefix-DT-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{DT}$ *Mndetprefix A P*›
and *Mndetprefix-F-subset* : ‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{F}$ *Mndetprefix A P*›
and *Mndetprefix-T-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{T}$ *Mndetprefix A P*›
and *Mndetprefix-D-subset* : ‹$A \subseteq B \Longrightarrow$ *Mndetprefix B P* $\sqsubseteq_{D}$ *Mndetprefix A P*›
⟨*proof*⟩

lemmas *Mndetprefix-le-subset* =
Mndetprefix-FD-subset Mndetprefix-DT-subset
Mndetprefix-F-subset Mndetprefix-T-subset Mndetprefix-D-subset

lemma *ndet-write-FD-subset* : ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{FD} (c!!a{\in}A \to P\ a)$›
and *ndet-write-DT-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{DT} (c!!a{\in}A \to P\ a)$›
and *ndet-write-F-subset* : ‹$A \neq \{\} \Longrightarrow (c!!b{\in}B \to P\ b) \sqsubseteq_{F} (c!!a{\in}A \to P\ a)$›
and *ndet-write-T-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{T} (c!!a{\in}A \to P\ a)$›
and *ndet-write-D-subset* : ‹$(c!!b{\in}B \to P\ b) \sqsubseteq_{D} (c!!a{\in}A \to P\ a)$›
if ‹*inj-on c B*› **and** ‹$A \subseteq B$›

⟨*proof*⟩

lemmas *ndet-write-le-subset* =
ndet-write-FD-subset ndet-write-DT-subset
ndet-write-F-subset ndet-write-T-subset ndet-write-D-subset

lemma *Mndetprefix-FD-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{FD}$ (*a* → *P a*)›
and *Mndetprefix-DT-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{DT}$ (*a* → *P a*)›
and *Mndetprefix-F-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{F}$ (*a* → *P a*)›
and *Mndetprefix-T-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{T}$ (*a* → *P a*)›
and *Mndetprefix-D-write0* : ‹*Mndetprefix A P* $\sqsubseteq_{D}$ (*a* → *P a*)› **if** ‹*a* ∈ *A*›
⟨*proof*⟩

lemmas *Mndetprefix-le-write0* =
Mndetprefix-FD-write0 Mndetprefix-DT-write0
Mndetprefix-F-write0 Mndetprefix-T-write0 Mndetprefix-D-write0

lemma *Mprefix-DT-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{DT}$ *Mprefix A P*›
and *Mprefix-T-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{T}$ *Mprefix A P*›
and *Mprefix-D-subset* : ‹*A* ⊆ *B* ⟹ *Mprefix B P* $\sqsubseteq_{D}$ *Mprefix A P*›
⟨*proof*⟩

lemmas *Mprefix-le-subset* = *Mprefix-DT-subset Mprefix-T-subset Mprefix-D-subset*

Mprefix set monotony doesn't hold for ($\sqsubseteq_F$) and ($\sqsubseteq_{FD}$) where it holds for deterministic choice.

lemma *read-DT-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{DT}$ (*c*?*a*∈*A* → *P a*)›
and *read-T-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{T}$ (*c*?*a*∈*A* → *P a*)›
and *read-D-subset* : ‹(*c*?*b*∈*B* → *P b*) $\sqsubseteq_{D}$ (*c*?*a*∈*A* → *P a*)›
if ‹*inj-on c B*› **and** ‹*A* ⊆ *B*›
⟨*proof*⟩

11.2.2 Monotony Properties

Non Deterministic Choice Operator Laws

lemma *Ndet-FD-self-left* : ‹*P* ⊓ *Q* $\sqsubseteq_{FD}$ *P*›
and *Ndet-DT-self-left* : ‹*P* ⊓ *Q* $\sqsubseteq_{DT}$ *P*›
and *Ndet-F-self-left* : ‹*P* ⊓ *Q* $\sqsubseteq_{F}$ *P*›
and *Ndet-T-self-left* : ‹*P* ⊓ *Q* $\sqsubseteq_{T}$ *P*›
and *Ndet-D-self-left* : ‹*P* ⊓ *Q* $\sqsubseteq_{D}$ *P*›
and *Ndet-FD-self-right* : ‹*P* ⊓ *Q* $\sqsubseteq_{FD}$ *Q*›
and *Ndet-DT-self-right* : ‹*P* ⊓ *Q* $\sqsubseteq_{DT}$ *Q*›
and *Ndet-F-self-right* : ‹*P* ⊓ *Q* $\sqsubseteq_{F}$ *Q*›
and *Ndet-T-self-right* : ‹*P* ⊓ *Q* $\sqsubseteq_{T}$ *Q*›
and *Ndet-D-self-right* : ‹*P* ⊓ *Q* $\sqsubseteq_{D}$ *Q*›
⟨*proof*⟩

lemmas *Ndet-le-self-left = Ndet-FD-self-left Ndet-DT-self-left Ndet-F-self-left Ndet-T-self-left Ndet-D-self-left*
and *Ndet-le-self-right = Ndet-FD-self-right Ndet-DT-self-right Ndet-F-self-right Ndet-T-self-right Ndet-D-self-right*

lemma *Ndet-FD-Det* : $\langle P \sqcap Q \sqsubseteq_{FD} P \mathbin{\square} Q \rangle$
and *Ndet-DT-Det* : $\langle P \sqcap Q \sqsubseteq_{DT} P \mathbin{\square} Q \rangle$
and *Ndet-F-Det* : $\langle P \sqcap Q \sqsubseteq_{F} P \mathbin{\square} Q \rangle$
and *Ndet-T-Det* : $\langle P \sqcap Q \sqsubseteq_{T} P \mathbin{\square} Q \rangle$
and *Ndet-D-Det* : $\langle P \sqcap Q \sqsubseteq_{D} P \mathbin{\square} Q \rangle$
⟨*proof*⟩

lemmas *Ndet-le-Det = Ndet-FD-Det Ndet-DT-Det Ndet-F-Det Ndet-T-Det Ndet-D-Det*

lemma *Ndet-FD-Sliding* : $\langle P \sqcap Q \sqsubseteq_{FD} P \rhd Q \rangle$
and *Ndet-DT-Sliding* : $\langle P \sqcap Q \sqsubseteq_{DT} P \rhd Q \rangle$
and *Ndet-F-Sliding* : $\langle P \sqcap Q \sqsubseteq_{F} P \rhd Q \rangle$
and *Ndet-T-Sliding* : $\langle P \sqcap Q \sqsubseteq_{T} P \rhd Q \rangle$
and *Ndet-D-Sliding* : $\langle P \sqcap Q \sqsubseteq_{D} P \rhd Q \rangle$
⟨*proof*⟩

lemmas *Ndet-le-Sliding = Ndet-FD-Sliding Ndet-DT-Sliding Ndet-F-Sliding Ndet-T-Sliding Ndet-D-Sliding*

lemma *GlobalNdet-refine-FD* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{FD} P\ a \rangle$
⟨*proof*⟩

lemma *GlobalNdet-refine-DT* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{DT} P\ a \rangle$
⟨*proof*⟩

lemma *GlobalNdet-refine-F* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{F} P\ a \rangle$
⟨*proof*⟩

lemma *GlobalNdet-refine-D* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{D} P\ a \rangle$
⟨*proof*⟩

lemma *GlobalNdet-refine-T* : $\langle a \in A \Longrightarrow \sqcap a \in A.\ P\ a \sqsubseteq_{T} P\ a \rangle$
⟨*proof*⟩

lemmas *GlobalNdet-refine-le = GlobalNdet-refine-FD GlobalNdet-refine-DT GlobalNdet-refine-F GlobalNdet-refine-D GlobalNdet-refine-T*

lemma *mono-GlobalNdet-FD-const*:
$\langle A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{FD} Q\ a) \Longrightarrow P \sqsubseteq_{FD} \sqcap a \in A.\ Q\ a \rangle$
⟨*proof*⟩

lemma *mono-GlobalNdet-DT-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{DT}\ Q\ a) \Longrightarrow P \sqsubseteq_{DT} \sqcap a \in A.\ Q\ a$›
⟨*proof*⟩

lemma *mono-GlobalNdet-F-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{F}\ Q\ a) \Longrightarrow P \sqsubseteq_{F} \sqcap a \in A.\ Q\ a$›
⟨*proof*⟩

lemma *mono-GlobalNdet-D-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{D}\ Q\ a) \Longrightarrow P \sqsubseteq_{D} \sqcap a \in A.\ Q\ a$›
⟨*proof*⟩

lemma *mono-GlobalNdet-T-const*:
‹$A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow P \sqsubseteq_{T}\ Q\ a) \Longrightarrow P \sqsubseteq_{T} \sqcap a \in A.\ Q\ a$›
⟨*proof*⟩

lemmas *mono-GlobalNdet-le-const = mono-GlobalNdet-FD-const mono-GlobalNdet-DT-const mono-GlobalNdet-F-const mono-GlobalNdet-D-const mono-GlobalNdet-T-const*

Refinements and Constant Processes

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

lemma *SKIP-FD-iff*: ‹$SKIP\ r \sqsubseteq_{FD}\ P \longleftrightarrow P = SKIP\ r$› **for** P :: ‹$('a, 'r)\ process_{ptick}$›
⟨*proof*⟩

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

Chapter 12

Algebraic Rules of CSP

12.1 The Non-Deterministic Distributivity

CSP operators are distributive over non deterministic choice.

12.1.1 Global Distributivity

lemma *Mndetprefix-distrib-GlobalNdet* :
‹$B \neq \{\} \Longrightarrow (\sqcap a \in A \rightarrow \sqcap b \in B.\ P\ a\ b) = \sqcap b \in B.\ \sqcap a \in A \rightarrow P\ a\ b$›
⟨*proof*⟩

lemma *Det-distrib-GlobalNdet-left* :
‹$P \ \Box\ (\sqcap a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ \mathit{else}\ \sqcap a{\in}A.\ P \ \Box\ Q\ a)$›
⟨*proof*⟩

corollary *Det-distrib-GlobalNdet-right* :
‹$(\sqcap a{\in}A.\ P\ a) \ \Box\ Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ Q\ \mathit{else}\ \sqcap a{\in}A.\ P\ a \ \Box\ Q)$›
⟨*proof*⟩

lemma *Sliding-distrib-GlobalNdet-left* :
‹$P \rhd (\sqcap a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P \sqcap \mathit{STOP}\ \mathit{else}\ \sqcap a{\in}A.\ P \rhd Q\ a)$›
⟨*proof*⟩

lemma *Sliding-distrib-GlobalNdet-right* :
‹$(\sqcap a{\in}A.\ P\ a) \rhd Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ Q\ \mathit{else}\ \sqcap a{\in}A.\ P\ a \rhd Q)$›
⟨*proof*⟩

lemma *Sync-distrib-GlobalNdet-left* :
‹$P \ [\![S]\!]\ (\sqcap\ a{\in}A.\ Q\ a) = (\mathit{if}\ A = \{\}\ \mathit{then}\ P\ [\![S]\!]\ \mathit{STOP}\ \mathit{else} \sqcap\ a{\in}A.\ (P\ [\![S]\!]\ Q\ a))$›
⟨*proof*⟩

corollary *Sync-distrib-GlobalNdet-right* :
‹$(\sqcap\ a{\in}A.\ P\ a)\ [\![S]\!]\ Q = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{STOP}\ [\![S]\!]\ Q\ \mathit{else} \sqcap\ a{\in}A.\ (P\ a\ [\![S]\!]\ Q))$›

⟨*proof*⟩

lemmas *Inter-distrib-GlobalNdet-left* = *Sync-distrib-GlobalNdet-left*[**where** $S = ‹\{\}›$]
and *Par-distrib-GlobalNdet-left* = *Sync-distrib-GlobalNdet-left*[**where** $S = UNIV$]
and *Inter-distrib-GlobalNdet-right* = *Sync-distrib-GlobalNdet-right*[**where** $S = ‹\{\}›$]
and *Par-distrib-GlobalNdet-right* = *Sync-distrib-GlobalNdet-right*[**where** $S = UNIV$]

lemma *Seq-distrib-GlobalNdet-left* :
$‹P \mathbin{;} \sqcap a{\in}A.\ Q\ a = (\mathit{if}\ A = \{\}\ \mathit{then}\ P \mathbin{;} \mathit{STOP}\ \mathit{else} \sqcap a{\in}A.\ (P \mathbin{;} Q\ a))›$
⟨*proof*⟩

lemma *Seq-distrib-GlobalNdet-right* : $‹(\sqcap a{\in}A.\ P\ a) \mathbin{;} Q = \sqcap a{\in}A.\ (P\ a \mathbin{;} Q)›$
⟨*proof*⟩

lemma *Renaming-distrib-GlobalNdet* : $‹\mathit{Renaming}\ (\sqcap a{\in}A.\ P\ a)\ f\ g = \sqcap a{\in}A.\ \mathit{Renaming}\ (P\ a)\ f\ g›$
⟨*proof*⟩

The (\) operator does not distribute the *GlobalNdet* in general. We give a finite version derived from the binary version below.

12.1.2 Binary Distributivity

lemma *Mndetprefix-distrib-Ndet* :
$‹(\sqcap a \in A \to P\ a \sqcap Q\ a) = (\sqcap a \in A \to P\ a) \sqcap (\sqcap a \in A \to Q\ a)›$
⟨*proof*⟩

lemma *Det-distrib-Ndet-left* : $‹P \mathbin{\square} Q \sqcap R = (P \mathbin{\square} Q) \sqcap (P \mathbin{\square} R)›$
⟨*proof*⟩

corollary *Det-distrib-Ndet-right* : $‹P \sqcap Q \mathbin{\square} R = (P \mathbin{\square} R) \sqcap (Q \mathbin{\square} R)›$
⟨*proof*⟩

lemma *Ndet-distrib-Det-left* : $‹P \sqcap (Q \mathbin{\square} R) = P \sqcap Q \mathbin{\square} P \sqcap R›$
⟨*proof*⟩

corollary *Ndet-distrib-Det-right* : $‹(P \mathbin{\square} Q) \sqcap R = P \sqcap R \mathbin{\square} Q \sqcap R›$
⟨*proof*⟩

lemma *Sliding-distrib-Ndet-left* : $‹P \rhd (Q \sqcap R) = (P \rhd Q) \sqcap (P \rhd R)›$
and *Sliding-distrib-Ndet-right* : $‹(P \sqcap Q) \rhd R = (P \rhd R) \sqcap (Q \rhd R)›$

⟨*proof*⟩

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

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

lemma *Seq-distrib-Ndet-left* : ‹P ; $Q \sqcap R = (P$; $Q) \sqcap (P$; $R)$›
⟨*proof*⟩

lemma *Seq-distrib-Ndet-right* : ‹$P \sqcap Q$; $R = (P$; $R) \sqcap (Q$; $R)$›
⟨*proof*⟩

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

lemma *Hiding-distrib-Ndet* : ‹$P \sqcap Q \setminus S = (P \setminus S) \sqcap (Q \setminus S)$›
⟨*proof*⟩

lemma *Hiding-distrib-finite-GlobalNdet* :
‹*finite* $A \Longrightarrow (\sqcap a \in A.\ P\ a) \setminus S = \sqcap a \in A.\ (P\ a \setminus S)$›
⟨*proof*⟩

For the *Mprefix* operator, we obviously do not have a conventional distributivity: ($\sqcap$) becomes ($\Box$).

lemma *Mprefix-distrib-Ndet* :
‹$(\Box a \in A \to P \sqcap Q) = (\Box a \in A \to P) \Box (\Box a \in A \to Q)$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

12.2 The Basic Laws

By "basic" laws we mean the behaviour of $\bot$, *STOP* and *SKIP*, plus the associativity of some concerned operators.

12.2.1 The Laws of $\bot$

From the characterization $(?P = \bot) = ([] \in \mathcal{D}\ ?P)$, we can easily derive the behaviour of $\bot$ wrt. *STOP*, *SKIP*, and the operators.

lemma *BOT-less1* [*simp*]: ‹$(\bot :: ('a, 'r)\ process_{ptick}) \leq X$›
⟨*proof*⟩

lemma *STOP-neq-BOT* [*simp*] : ‹$STOP \neq \bot$›
and *SKIP-neq-BOT* [*simp*] : ‹$SKIP\ r \neq \bot$›
⟨*proof*⟩

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

lemma *Det-BOT* [*simp*] : ‹$P \sqcap \bot = \bot$› **and** *BOT-Det* [*simp*] : ‹$\bot \sqcap P = \bot$›
⟨*proof*⟩

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

lemma *Ndet-BOT* [*simp*] : ‹$P \sqinter \bot = \bot$› **and** *BOT-Ndet* [*simp*] : ‹$\bot \sqinter P = \bot$›
⟨*proof*⟩

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

lemma *Sliding-BOT* [*simp*] : ‹$P \rhd \bot = \bot$› **and** *BOT-Sliding* [*simp*] : ‹$\bot \rhd P = \bot$›
⟨*proof*⟩

lemma *Mprefix-neq-BOT* [*simp*] : ‹$\Box\ a \in A \rightarrow P\ a \neq \bot$›
⟨*proof*⟩

lemma *Mndetprefix-neq-BOT* [*simp*] : ‹$\sqinter\ a \in A \rightarrow P\ a \neq \bot$›
⟨*proof*⟩

lemma *Seq-is-BOT-iff* : ‹$P ; Q = \bot \longleftrightarrow P = \bot \vee (\exists\, r.\ [\checkmark(r)] \in \mathcal{T}\ P \wedge Q = \bot)$›
⟨*proof*⟩

lemma *BOT-Seq* [*simp*] : ‹$\bot ; P = \bot$› ⟨*proof*⟩

lemma *Hiding-BOT* [*simp*] : ‹$\bot \setminus A = \bot$›
⟨*proof*⟩

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

lemma *Sync-BOT* [*simp*] : ‹$P \llbracket S \rrbracket \bot = \bot$› **and** *BOT-Sync* [*simp*] : ‹$\bot \llbracket S \rrbracket P = \bot$›
⟨*proof*⟩

lemma *Renaming-is-BOT-iff* : ‹*Renaming* $P\ f\ g = \bot \longleftrightarrow P = \bot$›
⟨*proof*⟩

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

lemma *GlobalNdet-is-BOT-iff* : ‹$(\sqcap a{\in}A.\ P\ a) = \bot \longleftrightarrow (\exists\, a{\in}A.\ P\ a = \bot)$›
⟨*proof*⟩

lemma *GlobalNdet-BOT* [*simp*] : ‹$a \in A \Longrightarrow P\ a = \bot \Longrightarrow (\sqcap a{\in}A.\ P\ a) = \bot$›
⟨*proof*⟩

12.2.2 The Laws of *STOP*

From the characterization $(?P = STOP) = (\mathcal{T}\ ?P = \{[]\})$, we can easily derive the behaviour of *STOP* wrt. *SKIP* and the operators.

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

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

lemma *Det-STOP* [*simp*] : ‹$P \mathbin{\square} STOP = P$› **and** *STOP-Det* [*simp*] : ‹$STOP \mathbin{\square} P = P$›
⟨*proof*⟩

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

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

lemma *Sliding-STOP* [*simp*] : ‹$P \rhd STOP = P \sqcap STOP$› **and** *STOP-Sliding* [*simp*] : ‹$STOP \rhd P = P$›
⟨*proof*⟩

lemma *Mprefix-is-STOP-iff* : ‹$\square\ a \in A \to P\ a = STOP \longleftrightarrow A = \{\}$›
⟨*proof*⟩

lemma *Mprefix-empty* [*simp*] : ‹$\Box\ a \in \{\} \to P\ a = STOP$›
⟨*proof*⟩

lemma *Mndetprefix-is-STOP-iff* : ‹$\sqcap\ a \in A \to P\ a = STOP \longleftrightarrow A = \{\}$›
⟨*proof*⟩

lemma *Mndetprefix-empty* [*simp*] : ‹$\sqcap\ a \in \{\} \to P\ a = STOP$›
⟨*proof*⟩

lemma *Seq-is-STOP-iff* :
‹$P\ ;\ Q = STOP \longleftrightarrow \mathcal{T}\ P \subseteq insert\ []\ \{[\checkmark(r)]|\ r.\ True\} \wedge (P \neq STOP \longrightarrow Q = STOP)$›
⟨*proof*⟩

lemma *STOP-Seq* [*simp*] : ‹$STOP\ ;\ P = STOP$› ⟨*proof*⟩

lemma *Hiding-STOP* [*simp*] : ‹$STOP \setminus A = STOP$›
⟨*proof*⟩

lemma *STOP-Sync-STOP* [*simp*] : ‹$STOP\ [\![S]\!]\ STOP = STOP$›
⟨*proof*⟩

lemma *Renaming-is-STOP-iff* : ‹$Renaming\ P\ f\ g = STOP \longleftrightarrow P = STOP$›
⟨*proof*⟩

lemma *Renaming-STOP* [*simp*] : ‹$Renaming\ STOP\ f\ g = STOP$›
⟨*proof*⟩

lemma *GlobalNdet-is-STOP-iff* : ‹$(\sqcap a{\in}A.\ P\ a) = STOP \longleftrightarrow (\forall\, a{\in}A.\ P\ a = STOP)$›
⟨*proof*⟩

lemma *GlobalNdet-empty* [*simp*] : ‹$(\sqcap a{\in}\{\}.\ P\ a) = STOP$›
⟨*proof*⟩

lemma *GlobalNdet-id* [*simp*] : ‹$(\sqcap a{\in}A.\ P) = (if\ A = \{\}\ then\ STOP\ else\ P)$›
⟨*proof*⟩

lemma ‹$(\sqcap a{\in}A.\ STOP) = STOP$› ⟨*proof*⟩

More powerful Laws

lemma *Inter-STOP* [*simp*] : ‹$P\ |||\ STOP = P\ ;\ STOP$›

⟨*proof*⟩

corollary *STOP-Inter* [*simp*] : ‹*STOP* ||| *P* = *P* ; *STOP*›
⟨*proof*⟩

lemma *Par-STOP* [*simp*] : ‹*P* || *STOP* = (*if P* = ⊥ *then* ⊥ *else STOP*)›
⟨*proof*⟩

lemma *STOP-Par* [*simp*] : ‹*STOP* || *P* = (*if P* = ⊥ *then* ⊥ *else STOP*)›
⟨*proof*⟩

12.2.3 The Laws of *SKIP*

The definition of SKIPS

definition *SKIPS* :: ‹'*r set* ⇒ ('*a*, '*r*) *process*$_{ptick}$›
where ‹*SKIPS R* ≡ ⊓*r* ∈ *R*. *SKIP r*›

lemma *SKIPS-empty* [*simp*] : ‹*SKIPS* {} = *STOP*›
and *SKIPS-singl-is-SKIP* [*simp*] : ‹*SKIPS* {*r*} = *SKIP r*› ⟨*proof*⟩

lemma *F-SKIPS* :
‹*F* (*SKIPS R*) = (*if R* = {} *then* {(*s*, *X*). *s* = []}
else {([], *X*) |*X*. ∃ *r* ∈ *R*. *tick r* ∉ *X*} ∪
{(*s*, *X*). ∃ *r* ∈ *R*. *s* = [*tick r*]})›
⟨*proof*⟩

lemma *D-SKIPS* : ‹*D* (*SKIPS R*) = {}›
⟨*proof*⟩

lemma *T-SKIPS* : ‹*T* (*SKIPS R*) = *insert* [] {[*tick r*]| *r*. *r* ∈ *R*}›
⟨*proof*⟩

lemmas *SKIPS-projs* = *F-SKIPS D-SKIPS T-SKIPS*

Laws

lemma *SKIP-Sliding* [*simp*] : ‹*SKIP r* ▷ *P* = *P* ⊓ *SKIP r*›
⟨*proof*⟩

lemma *Sliding-SKIP* [*simp*] : ‹*P* ▷ *SKIP r* = *P* □ *SKIP r*›
⟨*proof*⟩

lemma *Mprefix-neq-SKIP* [*simp*] : ‹□ *a* ∈ *A* → *P a* ≠ *SKIP r*›
⟨*proof*⟩

lemma *Mndetprefix-neq-SKIP* [*simp*] : ‹⊓ *a* ∈ *A* → *P a* ≠ *SKIP r*›

⟨*proof*⟩

lemma *SKIP-Seq* [*simp*] : ‹*SKIP r* ; *P* = *P*›
⟨*proof*⟩

lemma *Seq-SKIP* [*simp*] : ‹*P* ; *SKIP* () = *P*›
— This is only true when the type $'r$ of ($'a$, $'r$) $process_{ptick}$ is set to *unit* (morally, we have "only one *tick*").
⟨*proof*⟩

lemma *Hiding-SKIP* [*simp*] : ‹*SKIP r* \ *A* = *SKIP r*›
⟨*proof*⟩

Synchronization

lemma *SKIP-Sync-SKIP* [*simp*] : ‹*SKIP r* ⟦ *S* ⟧ *SKIP s* = (*if r* = *s then SKIP r else STOP*)›
for *S* :: ‹$'a$ *set*› **and** *r* :: $'r$
⟨*proof*⟩

lemma *SKIP-Sync-STOP* [*simp*] : ‹*SKIP r* ⟦*S*⟧ *STOP* = *STOP*›
and *STOP-Sync-SKIP* [*simp*] : ‹*STOP* ⟦*S*⟧ *SKIP r* = *STOP*›
⟨*proof*⟩

lemma *Mprefix-Sync-SKIP* : ‹(□*a* ∈ *A* → *P a*) ⟦*S*⟧ *SKIP res* = □*a* ∈ (*A* − *S*) → (*P a* ⟦*S*⟧ *SKIP res*)›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *SKIP-Sync-Mprefix* : ‹*SKIP res* ⟦*S*⟧ (□*b* ∈ *B* → *P b*) = □*b* ∈ (*B* − *S*) → (*SKIP res* ⟦*S*⟧ *P b*)›
⟨*proof*⟩

lemma *Renaming-SKIP* [*simp*] : ‹*Renaming* (*SKIP r*) *f g* = *SKIP* (*g r*)›
⟨*proof*⟩

Associative Operators

lemma *Det-assoc*: ‹*P* □ (*Q* □ *R*) = *P* □ *Q* □ *R*›
⟨*proof*⟩

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

lemma *Sliding-assoc*: ‹$P \rhd (Q \rhd R) = P \rhd Q \rhd R$›
⟨*proof*⟩

lemma *Seq-assoc* : ‹$P ; (Q ; R) = P ; Q ; R$›
⟨*proof*⟩

Synchronization

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

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

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

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

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

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

lemma *interleave-assoc-1*:
‹⟦*tu setinterleaves* ((*t*, *u*), *A*); *tuv setinterleaves* ((*tu*, *v*), *A*)⟧ ⟹
∃ *uv*. *uv setinterleaves* ((*u*, *v*), *A*) ∧ *tuv setinterleaves* ((*t*, *uv*), *A*)›
⟨*proof*⟩

lemma *interleave-assoc-2*:
assumes ∗:‹*uv setinterleaves* ((*u*, *v*), *A*)› **and**
∗∗:‹*tuv setinterleaves* ((*t*, *uv*), *A*)›
shows ‹∃ *tu*. *tu setinterleaves* ((*t*, *u*), *A*) ∧ *tuv setinterleaves* ((*tu*, *v*), *A*)›
⟨*proof*⟩

theorem *Sync-assoc*: ‹*P* ⟦*S*⟧ (*Q* ⟦*S*⟧ *R*) = *P* ⟦*S*⟧ *Q* ⟦*S*⟧ *R*› **for** *P Q R* :: ‹('*a*, '*r*) *process*$_{ptick}$›
⟨*proof*⟩

12.3 The Step-Laws

The step-laws describe the behaviour of the operators wrt. the multi-prefix choice.

12.3.1 Deterministic Choice

lemma *Mprefix-Det-Mprefix*:
‹(□*a* ∈ *A* → *P a*) □ (□*b* ∈ *B* → *Q b*) =
□*x* ∈ (*A* ∪ *B*) → (*if x* ∈ *A* ∩ *B then P x* ⊓ *Q x else if x* ∈ *A then P x else Q x*)›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

corollary *Mprefix-Un-distrib*:
‹□*a* ∈ (*A* ∪ *B*) → *P a* = (□*a* ∈ *A* → *P a*) □ (□*b* ∈ *B* → *P b*)›
⟨*proof*⟩

12.3.2 Non-Deterministic Choice

lemma *Mprefix-Ndet-Mprefix* :
‹(□*a* ∈ *A* → *P a*) ⊓ (□*b* ∈ *B* → *Q b*) =
(□*a* ∈ (*A* − *B*) → *P a*) ⊓ (□*b* ∈ (*B* − *A*) → *Q b*) □ (□*x* ∈ (*A* ∩ *B*) → *P x* ⊓ *Q x*)›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

12.3.3 Sliding Choice

lemma *Mprefix-Sliding-Mprefix* :

‹(□a∈ A → P a) ▷ (□b ∈ B → Q b) =
(□x ∈ (A ∪ B) → (*if* x ∈ A ∩ B *then* P x ⊓ Q x *else if* x ∈ A *then* P x *else* Q x)) ⊓
(□b ∈ B → (*if* b ∈ A *then* P b ⊓ Q b *else* Q b))›
(**is** ‹*?lhs* = *?rhs*›)
— It is not so much a "step law" as a rewriting.
⟨*proof*⟩

corollary *Mprefix-Sliding-superset-Mprefix* :
‹(□a∈ A → P a) ▷ (□b ∈ B → Q b) =
□b ∈ B → (*if* b ∈ A *then* P b ⊓ Q b *else* Q b)› **if** ‹A ⊆ B›
— This one (with the additional assumption $A \subseteq B$) is a "step law".
⟨*proof*⟩

corollary *Mprefix-Sliding-same-set-Mprefix* :
‹(□a∈ A → P a) ▷ (□a ∈ A → Q a) = □a ∈ A → P a ⊓ Q a›
⟨*proof*⟩

12.3.4 Sequential Composition

lemma *Mprefix-Seq*: ‹□a ∈ A → P a ; Q = □a ∈ A → (P a ; Q)›
⟨*proof*⟩

12.3.5 Hiding

We use a context to hide the intermediate results.

context begin

Two intermediate Results

private lemma *D-Hiding-Mprefix-dir2*:
‹s ∈ 𝒟 (□a ∈ A → P a \ S)› **if** ‹s ∈ 𝒟 (□a ∈ (A − S) → (P a \ S))›
⟨*proof*⟩ **lemma** *F-Hiding-Mprefix-dir2*:
‹(s, X) ∈ ℱ (□a ∈ A → P a \ S)› **if** ‹s ≠ []› **and** ‹(s, X) ∈ ℱ (□a ∈ (A − S) → (P a \ S))›
⟨*proof*⟩

(\) and *Mprefix* for disjoint Sets

theorem *Hiding-Mprefix-disjoint*:
‹□a ∈ A → P a \ S = □a ∈ A → (P a \ S)›
(**is** ‹*?lhs* = *?rhs*›) **if** *disjoint*: ‹A ∩ S = {}›
⟨*proof*⟩

(\) and *Mprefix* for non disjoint Sets

theorem *Hiding-Mprefix-non-disjoint*:
— Rework this proof

‹$\Box a \in A \to P\ a \setminus S = (\Box a \in (A - S) \to (P\ a \setminus S)) \rhd (\sqcap a \in (A \cap S).\ (P\ a \setminus S))$›
(**is** ‹*?lhs = ?rhs*›) **if** *non-disjoint*: ‹$A \cap S \neq \{\}$›
⟨*proof*⟩

end

12.3.6 Synchronization

lemma *Mprefix-Sync-Mprefix-bis* :
‹*Mprefix* $(A \cup A')\ P\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q =$
$(\Box a{\in}A \to (P\ a\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q))$
$\Box\ (\Box b{\in}B \to (Mprefix\ (A \cup A')\ P\ [\![S]\!]\ Q\ b))$
$\Box\ (\Box x{\in}(A' \cap B') \to (P\ x\ [\![S]\!]\ Q\ x))$›
(**is** ‹*?lhs A A' P B B' Q = ?rhs A A' P B B' Q*›)
if *sets-assms*: ‹$A \cap S = \{\}$› ‹$A' \subseteq S$› ‹$B \cap S = \{\}$› ‹$B' \subseteq S$›
for $P\ Q$ :: ‹$'a \Rightarrow ('a, 'r)\ process_{ptick}$›
⟨*proof*⟩

corollary *Mprefix-Sync-Mprefix*:
— This version is easier to use.
‹$\Box a{\in}A \to P\ a\ [\![S]\!]\ \Box b{\in}B \to Q\ b =$
$(\Box a{\in}(A - S) \to (P\ a\ [\![S]\!]\ \Box b{\in}B \to Q\ b))\ \Box$
$(\Box b{\in}(B - S) \to (\Box a{\in}A \to P\ a\ [\![S]\!]\ Q\ b))\ \Box$
$(\Box x{\in}(A \cap B \cap S) \to (P\ x\ [\![S]\!]\ Q\ x))$›
⟨*proof*⟩

Renaming

lemma *Renaming-Mprefix*:
‹*Renaming* $(\Box a \in A \to P\ a)\ f\ g =$
$\Box y \in f\ `\ A \to \sqcap a \in \{x \in A.\ y = f\ x\}.\ Renaming\ (P\ a)\ f\ g$› (**is** ‹*?lhs = ?rhs*›)
⟨*proof*⟩

12.4 Extension of the Step-Laws

12.4.1 Derived step-laws for *Sync*

lemma *Mprefix-Inter-Mprefix* :
‹$\Box a{\in}A \to P\ a\ |||\ \Box b{\in}B \to Q\ b = (\Box a{\in}A \to (P\ a\ |||\ \Box b{\in}B \to Q\ b))\ \Box\ (\Box b{\in}B \to (\Box a \in A \to P\ a\ |||\ Q\ b))$›
⟨*proof*⟩

lemma *Mprefix-Par-Mprefix* : ‹$\Box a{\in}A \to P\ a\ ||\ \Box b{\in}B \to Q\ b = \Box x{\in}(A \cap B) \to (P\ x\ ||\ Q\ x)$›
⟨*proof*⟩

lemma *Mprefix-Sync-Mprefix-subset* :

‹⟦A ⊆ S; B ⊆ S⟧ ⟹ Mprefix A P ⟦S⟧ Mprefix B Q = □x∈(A ∩ B) → (P x ⟦S⟧ Q x)›
⟨proof⟩

lemma *Mprefix-Sync-Mprefix-indep* :
‹A ∩ S = {} ⟹ B ∩ S = {} ⟹
Mprefix A P ⟦S⟧ Mprefix B Q = (□a∈A → (P a ⟦S⟧ Mprefix B Q)) □ (□b∈B → (Mprefix A P ⟦S⟧ Q b))›
⟨proof⟩

lemma *Mprefix-Sync-Mprefix-left* :
‹A ∩ S = {} ⟹ B ⊆ S ⟹ Mprefix A P ⟦S⟧ Mprefix B Q = □a∈A → (P a ⟦S⟧ Mprefix B Q)›
⟨proof⟩

lemma *Mprefix-Sync-Mprefix-right* :
‹A ⊆ S ⟹ B ∩ S = {} ⟹ Mprefix A P ⟦S⟧ Mprefix B Q = □b∈B → (Mprefix A P ⟦S⟧ Q b)›
⟨proof⟩

lemma *Mprefix-Sync-STOP* : ‹(□a ∈ A → P a) ⟦S⟧ STOP = □a ∈ (A − S) → (P a ⟦S⟧ STOP)›
⟨proof⟩

lemma *STOP-Sync-Mprefix* : ‹STOP ⟦S⟧ (□b ∈ B → Q b) = □b ∈ (B − S) → (STOP ⟦S⟧ Q b)›
⟨proof⟩

Mixing deterministic and non deterministic prefix choices **lemma** *Mndetprefix-Sync-Mprefix* :
‹(⊓a ∈ A → P a) ⟦S⟧ (□b ∈ B → Q b) =
(if A = {} then STOP ⟦S⟧ (□b ∈ B → Q b)
else ⊓a∈A. (if a ∈ S then STOP else (a → (P a ⟦S⟧ (□b ∈ B → Q b)))) □
(□b∈(B − S) → ((a → P a) ⟦S⟧ Q b)) □
(if a ∈ B ∩ S then (a → (P a ⟦S⟧ Q a)) else STOP))›
⟨proof⟩

lemma *Mprefix-Sync-Mndetprefix* :
‹(□a ∈ A → P a) ⟦S⟧ (⊓b ∈ B → Q b) =
(if B = {} then (□a ∈ A → P a) ⟦S⟧ STOP
else ⊓b∈B. (if b ∈ S then STOP else (b → ((□a ∈ A → P a) ⟦S⟧ Q b))) □
(□a∈(A − S) → (P a ⟦S⟧ (b → Q b))) □
(if b ∈ A ∩ S then (b → (P b ⟦S⟧ Q b)) else STOP))›
⟨proof⟩

In particular, we can obtain the theorem for *Mndetprefix* synchronized with *STOP*.

lemma *Mndetprefix-Sync-STOP* :

‹$(\sqcap a \in A \to P\ a)\ [\![S]\!]\ STOP =$
(*if* $A \cap S = \{\}$ *then* $\sqcap a \in A \to (P\ a\ [\![S]\!]\ STOP)$
else $(\sqcap a \in (A - S) \to (P\ a\ [\![S]\!]\ STOP)) \sqcap STOP$)›
(**is** ‹*?lhs* = (*if* $A \cap S = \{\}$ *then ?rhs1 else ?rhs2* $\sqcap$ *STOP*)›)
⟨*proof*⟩

corollary *STOP-Sync-Mndetprefix* :
‹$STOP\ [\![S]\!]\ (\sqcap b \in B \to Q\ b)\ =$
(*if* $B \cap S = \{\}$ *then* $\sqcap b \in B \to (STOP\ [\![S]\!]\ Q\ b)$
else $(\sqcap b \in (B - S) \to (STOP\ [\![S]\!]\ Q\ b)) \sqcap STOP$)›
⟨*proof*⟩

corollary *Mndetprefix-Sync-Mprefix-subset* :
‹$(\sqcap a \in A \to P\ a)\ [\![S]\!]\ (\Box b \in B \to Q\ b) =$
(*if* $A \subseteq B$ *then* $\sqcap\ a \in A \to (P\ a\ [\![S]\!]\ Q\ a)$
else $(\sqcap a \in (A \cap B) \to (P\ a\ [\![S]\!]\ Q\ a)) \sqcap STOP$)›
(**is** ‹*?lhs* = (*if* $A \subseteq B$ *then ?rhs1 else ?rhs2*)›) **if** ‹$A \subseteq S$› ‹$B \subseteq S$›
⟨*proof*⟩

corollary *Mprefix-Sync-Mndetprefix-subset* :
‹$A \subseteq S \Longrightarrow B \subseteq S \Longrightarrow$
$\Box a \in A \to P\ a\ [\![S]\!]\ \sqcap b \in B \to Q\ b =$
(*if* $B \subseteq A$ *then* $\sqcap b \in B \to (P\ b\ [\![S]\!]\ Q\ b)$
else $(\sqcap b \in (A \cap B) \to (P\ b\ [\![S]\!]\ Q\ b)) \sqcap STOP$)›
⟨*proof*⟩

corollary *Mndetprefix-Sync-Mprefix-indep* :
‹$(\sqcap a \in A \to P\ a)\ [\![S]\!]\ (\Box b \in B \to Q\ b) =$
(*if* $A = \{\}$ *then* $\Box b \in B \to (STOP\ [\![S]\!]\ Q\ b)$
else $\sqcap a \in A.\ (a \to (P\ a\ [\![S]\!]\ (\Box b \in B \to Q\ b))) \Box$
$(\Box b \in B \to ((a \to P\ a)\ [\![S]\!]\ Q\ b))$)›
if ‹$A \cap S = \{\}$› **and** ‹$B \cap S = \{\}$›
⟨*proof*⟩

corollary *Mprefix-Sync-Mndetprefix-indep* :
‹$A \cap S = \{\} \Longrightarrow B \cap S = \{\} \Longrightarrow$
$(\Box a \in A \to P\ a)\ [\![S]\!]\ (\sqcap b \in B \to Q\ b) =$
(*if* $B = \{\}$ *then* $\Box a \in A \to (P\ a\ [\![S]\!]\ STOP)$
else $\sqcap b \in B.\ (b \to ((\Box a \in A \to P\ a)\ [\![S]\!]\ Q\ b)) \Box$
$(\Box a \in A \to (P\ a\ [\![S]\!]\ (b \to Q\ b)))$)›
⟨*proof*⟩

corollary *Mndetprefix-Sync-Mprefix-left* :

‹(⊓a ∈ A → P a) ⟦S⟧ (□b ∈ B → Q b) =
(if A = {} then STOP ⟦S⟧ (□b ∈ B → Q b)
else ⊓a∈A → (P a ⟦S⟧ (□b ∈ B → Q b)))›
if ‹A ∩ S = {}› **and** ‹B ⊆ S›
⟨*proof*⟩

corollary *Mndetprefix-Sync-Mprefix-right* :
‹(⊓a ∈ A → P a) ⟦S⟧ (□b ∈ B → Q b) =
(if A = {} then STOP ⟦S⟧ (□b ∈ B → Q b)
else □b∈B → ((⊓a∈A → P a) ⟦S⟧ Q b))›
if ‹A ⊆ S› **and** ‹B ∩ S = {}›
⟨*proof*⟩

corollary *Mprefix-Sync-Mndetprefix-left* :
‹A ∩ S = {} ⟹ B ⊆ S ⟹
(□a ∈ A → P a) ⟦S⟧ (⊓b ∈ B → Q b) =
(if B = {} then (□a ∈ A → P a) ⟦S⟧ STOP
else □a∈A → (P a ⟦S⟧ (⊓b ∈ B → Q b)))›
⟨*proof*⟩

corollary *Mprefix-Sync-Mndetprefix-right* :
‹A ⊆ S ⟹ B ∩ S = {} ⟹
(□a ∈ A → P a) ⟦S⟧ (⊓b ∈ B → Q b) =
(if B = {} then (□a ∈ A → P a) ⟦S⟧ STOP
else ⊓b∈B → ((□a ∈ A → P a) ⟦S⟧ Q b))›
⟨*proof*⟩

corollary *Mndetprefix-Par-Mprefix* :
‹⊓a ∈ A → P a || □b ∈ B → Q b =
(if A ⊆ B then ⊓a ∈ A → (P a || Q a) else (⊓a ∈ (A ∩ B) → (P a || Q a)) ⊓ STOP)›
⟨*proof*⟩

corollary *Mprefix-Par-Mndetprefix* :
‹□a ∈ A → P a || ⊓b ∈ B → Q b =
(if B ⊆ A then ⊓b ∈ B → (P b || Q b) else (⊓b ∈ (A ∩ B) → (P b || Q b)) ⊓ STOP)›
⟨*proof*⟩

corollary *Mndetprefix-Inter-Mprefix* :
‹(⊓a ∈ A → P a) ||| (□b ∈ B → Q b) =
(if A = {} then (□b ∈ B → Q b ; STOP)
else ⊓a∈A. (a → (P a ||| (□b ∈ B → Q b))) □
(□b∈B → ((a → P a) ||| Q b)))›
⟨*proof*⟩

corollary *Mprefix-Inter-Mndetprefix* :
‹$(\Box a \in A \rightarrow P\ a) \mid\mid\mid (\sqcap b \in B \rightarrow Q\ b) =$
(*if* $B = \{\}$ *then* $(\Box a \in A \rightarrow P\ a$ **;** $STOP)$
else $\sqcap b \in B.\ (b \rightarrow ((\Box a \in A \rightarrow P\ a) \mid\mid\mid Q\ b)) \Box$
$(\Box a \in A \rightarrow (P\ a \mid\mid\mid (b \rightarrow Q\ b))))$›
⟨*proof*⟩

12.4.2 Non deterministic step-laws

The *Mndetprefix* operator slightly differs from *Mprefix* so we can often adapt the step-laws.

lemma *Mndetprefix-Ndet-Mndetprefix* :
— We assume $A \cap B \neq \{\}$, otherwise this rewriting rule is not interesting.
‹$(\sqcap a \in A \rightarrow P\ a) \sqcap (\sqcap b \in B \rightarrow Q\ b) =$
(*if* $A = B$ *then* $\sqcap b \in B \rightarrow P\ b \sqcap Q\ b$
else *if* $A \subseteq B$ *then* $(\sqcap b \in (B - A) \rightarrow Q\ b) \sqcap (\sqcap x \in A \rightarrow P\ x \sqcap Q\ x)$
else *if* $B \subseteq A$ *then* $(\sqcap a \in (A - B) \rightarrow P\ a) \sqcap (\sqcap x \in B \rightarrow P\ x \sqcap Q\ x)$
else $(\sqcap a \in (A - B) \rightarrow P\ a) \sqcap (\sqcap b \in (B - A) \rightarrow Q\ b) \sqcap (\sqcap x \in (A \cap B) \rightarrow P\ x \sqcap Q\ x))$›
(**is** ‹*?lhs* = (*if* $A = B$ *then* *?rhs1* *else if* $A \subseteq B$ *then* *?rhs2* *else if* $B \subseteq A$ *then* *?rhs3* *else* *?rhs4*)›)
if ‹$A \cap B \neq \{\}$›
⟨*proof*⟩

lemma *Mndetprefix-Det-Mndetprefix* :
‹$(\sqcap a \in A \rightarrow P\ a) \Box (\sqcap b \in B \rightarrow Q\ b) =$
(*if* $A = \{\}$ *then* $\sqcap b \in B \rightarrow Q\ b$ *else if* $B = \{\}$ *then* $\sqcap a \in A \rightarrow P\ a$
else $\sqcap a \in A.\ \sqcap b \in B.$ (*if* $a = b$ *then* $a \rightarrow P\ a \sqcap Q\ a$ *else* $(a \rightarrow P\ a) \Box (b \rightarrow Q\ b)))$›
(**is** ‹*?lhs* = (*if* $A = \{\}$ *then* *?rhs1* *else if* $B = \{\}$ *then* *?rhs2* *else* *?rhs3*)›)
⟨*proof*⟩

lemma *FD-Mndetprefix-Det-Mndetprefix* :
‹$\sqcap x \in (A \cup B) \rightarrow$ (*if* $x \in A \cap B$ *then* $P\ x \sqcap Q\ x$ *else if* $x \in A$ *then* $P\ x$ *else* $Q\ x$)
$\sqsubseteq_{FD}$ $(\sqcap a \in A \rightarrow P\ a) \Box (\sqcap b \in B \rightarrow Q\ b)$›
(**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
⟨*proof*⟩

lemma *FD-Mndetprefix-Sliding-Mndetprefix* :
‹$(\sqcap x \in (A \cup B) \rightarrow$ (*if* $x \in A \cap B$ *then* $P\ x \sqcap Q\ x$ *else if* $x \in A$ *then* $P\ x$ *else* $Q\ x)) \sqcap$
$(\sqcap b \in B \rightarrow$ (*if* $b \in A$ *then* $P\ b \sqcap Q\ b$ *else* $Q\ b)) \sqsubseteq_{FD}$
$(\sqcap a \in A \rightarrow P\ a) \rhd (\sqcap b \in B \rightarrow Q\ b)$› (**is** ‹*?lhs* $\sqsubseteq_{FD}$ *?rhs*›)
⟨*proof*⟩

lemma *Mndetprefix-Sliding-superset-Mndetprefix* :
‹$(\sqcap a{\in}\ A \rightarrow P\ a) \rhd (\sqcap b \in B \rightarrow Q\ b) =$
$\sqcap b \in B \rightarrow (\mathit{if}\ b \in A\ \mathit{then}\ P\ b \sqcap Q\ b\ \mathit{else}\ Q\ b)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹$A \subseteq B$›
⟨*proof*⟩

corollary *Mndetprefix-Sliding-same-set-Mndetprefix* :
‹$(\sqcap a{\in}\ A \rightarrow P\ a) \rhd (\sqcap a \in A \rightarrow Q\ a) = \sqcap a \in A \rightarrow P\ a \sqcap Q\ a$›
⟨*proof*⟩

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

lemma *Mndetprefix-Seq* : ‹$\sqcap a \in A \rightarrow P\ a$ **;** $Q = \sqcap a \in A \rightarrow (P\ a$ **;** $Q)$›
⟨*proof*⟩

For (\), we can not use the distributivity of *GlobalNdet* since it is only working for finite non determinism. But we can still reuse some previous results in the following proof.

theorem *Hiding-Mndetprefix-disjoint* :
‹$\sqcap a \in A \rightarrow P\ a \setminus S = \sqcap a \in A \rightarrow (P\ a \setminus S)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹$A \cap S = \{\}$›
⟨*proof*⟩

theorem *Hiding-Mndetprefix-subset* :
‹$\sqcap a \in A \rightarrow P\ a \setminus S = \sqcap a \in A.\ (P\ a \setminus S)$› (**is** ‹*?lhs* = *?rhs*›) **if** ‹$A \subseteq S$›
⟨*proof*⟩

theorem *Hiding-Mndetprefix-non-disjoint-not-subset* :
‹$\sqcap a \in A \rightarrow P\ a \setminus S = (\sqcap a \in (A - S) \rightarrow (P\ a \setminus S)) \sqcap (\sqcap a \in (A \cap S).\ (P\ a \setminus S))$›
(**is** ‹*?lhs* = *?rhs1* $\sqcap$ *?rhs2*›) **if** ‹$A \cap S \neq \{\}$› **and** ‹$\neg\ A \subseteq S$›
⟨*proof*⟩

12.5 Read and Write Laws

12.5.1 Projections

read

lemma *F-read* :
‹$\mathcal{F}\ (c\mathbf{?}a{\in}A \rightarrow P\ a) = \{([], X)\ |X.\ X \cap ev\ `\ c\ `\ A = \{\}\} \cup$

$\{(ev\ a\ \#\ s,\ X)\ |a\ s\ X.\ a \in c\ `\ A \wedge (s,\ X) \in \mathcal{F}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}\rangle$
⟨*proof*⟩

lemma *F-read-inj-on* :
$\langle inj\text{-}on\ c\ A \Longrightarrow$
$\mathcal{F}\ (c?a{\in}A \rightarrow P\ a) = \{([],\ X)\ |X.\ X \cap ev\ `\ c\ `\ A = \{\}\} \cup$
$\{(ev\ (c\ a)\ \#\ s,\ X)\ |a\ s\ X.\ a \in A \wedge (s,\ X) \in \mathcal{F}\ (P\ a)\}\rangle$
⟨*proof*⟩

lemma *D-read* :
$\langle \mathcal{D}\ (c?a{\in}A \rightarrow P\ a) = \{ev\ a\ \#\ s\ |a\ s.\ a \in c\ `\ A \wedge s \in \mathcal{D}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}\rangle$
⟨*proof*⟩

lemma *D-read-inj-on* :
$\langle inj\text{-}on\ c\ A \Longrightarrow \mathcal{D}\ (c?a{\in}A \rightarrow P\ a) = \{ev\ (c\ a)\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{D}\ (P\ a)\}\rangle$
⟨*proof*⟩

lemma *T-read* :
$\langle \mathcal{T}\ (c?a{\in}A \rightarrow P\ a) = insert\ []\ \{ev\ a\ \#\ s\ |a\ s.\ a \in c\ `\ A \wedge s \in \mathcal{T}\ ((P \circ inv\text{-}into\ A\ c)\ a)\}\rangle$
⟨*proof*⟩

lemma *T-read-inj-on* :
$\langle inj\text{-}on\ c\ A \Longrightarrow \mathcal{T}\ (c?a{\in}A \rightarrow P\ a) = insert\ []\ \{ev\ (c\ a)\ \#\ s\ |a\ s.\ a \in A \wedge s \in \mathcal{T}\ (P\ a)\}\rangle$
⟨*proof*⟩

lemmas *read-projs = F-read D-read T-read*
and *read-inj-on-projs = F-read-inj-on D-read-inj-on T-read-inj-on*

ndet-write

lemma *F-ndet-write* :
$\langle \mathcal{F}\ (c!!a{\in}A \rightarrow P\ a) =$
$(\ \ if\ A = \{\}\ then\ \{(s,\ X).\ s = []\}$
$\ \ else\ \{([],\ X)\ |X.\ \exists\, a{\in}A.\ ev\ (c\ a) \notin X\} \cup$
$\{(ev\ a\ \#\ s,\ X)\ |a\ s\ X.\ a \in c\ `\ A \wedge (s,\ X) \in \mathcal{F}\ ((P \circ inv\text{-}into\ A\ c)\ a)\})\rangle$
⟨*proof*⟩

lemma *F-ndet-write-inj-on* :
$\langle inj\text{-}on\ c\ A \Longrightarrow$
$\mathcal{F}\ (c!!a{\in}A \rightarrow P\ a) =$
$(\ \ if\ A = \{\}\ then\ \{(s,\ X).\ s = []\}$
$\ \ else\ \{([],\ X)\ |X.\ \exists\, a{\in}A.\ ev\ (c\ a) \notin X\} \cup$
$\{(ev\ (c\ a)\ \#\ s,\ X)\ |a\ s\ X.\ a \in A \wedge (s,\ X) \in \mathcal{F}\ (P\ a)\})\rangle$

⟨*proof*⟩

lemma *D-ndet-write* :
‹$\mathcal{D}$ (*c*!!*a*∈*A* → *P a*) = {*ev a* # *s* |*a s*. *a* ∈ *c* ‘ *A* ∧ *s* ∈ $\mathcal{D}$ ((*P* ∘ *inv-into A c*) *a*)}›
⟨*proof*⟩

lemma *D-ndet-write-inj-on* :
‹*inj-on c A* ⟹ $\mathcal{D}$ (*c*!!*a*∈*A* → *P a*) = {*ev* (*c a*) # *s* |*a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{D}$ (*P a*)}›
⟨*proof*⟩

lemma *T-ndet-write* :
‹$\mathcal{T}$ (*c*!!*a*∈*A* → *P a*) = *insert* [] {*ev a* # *s* |*a s*. *a* ∈ *c* ‘ *A* ∧ *s* ∈ $\mathcal{T}$ ((*P* ∘ *inv-into A c*) *a*)}›
⟨*proof*⟩

lemma *T-ndet-write-inj-on* :
‹*inj-on c A* ⟹ $\mathcal{T}$ (*c*!!*a*∈*A* → *P a*) = *insert* [] {*ev* (*c a*) # *s* |*a s*. *a* ∈ *A* ∧ *s* ∈ $\mathcal{T}$ (*P a*)}›
⟨*proof*⟩

lemmas *ndet-write-projs* = *F-ndet-write D-ndet-write T-ndet-write*
and *ndet-write-inj-on-projs* = *F-ndet-write-inj-on D-ndet-write-inj-on T-ndet-write-inj-on*

write **and** (→)

lemma *F-write* :
‹$\mathcal{F}$ (*c*!*a* → *P*) = {([], *X*) |*X*. *ev* (*c a*) ∉ *X*} ∪ {(*ev* (*c a*) # *s*, *X*) |*s X*. (*s*, *X*) ∈ $\mathcal{F}$ *P*}›
⟨*proof*⟩

lemma *F-write0* :
‹$\mathcal{F}$ (*a* → *P*) = {([], *X*) |*X*. *ev a* ∉ *X*} ∪ {(*ev a* # *s*, *X*) |*s X*. (*s*, *X*) ∈ $\mathcal{F}$ *P*}›
⟨*proof*⟩

lemma *D-write* : ‹$\mathcal{D}$ (*c*!*a* → *P*) = {*ev* (*c a*) # *s* |*s*. *s* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemma *D-write0* : ‹$\mathcal{D}$ (*a* → *P*) = {*ev a* # *s* |*s*. *s* ∈ $\mathcal{D}$ *P*}›
⟨*proof*⟩

lemma *T-write* : ‹$\mathcal{T}$ (*c*!*a* → *P*) = *insert* [] {*ev* (*c a*) # *s* |*s*. *s* ∈ $\mathcal{T}$ *P*}›
⟨*proof*⟩

lemma *T-write0* : ‹$\mathcal{T}$ (*a* → *P*) = *insert* [] {*ev a* # *s* |*s*. *s* ∈ $\mathcal{T}$ *P*}›

⟨*proof*⟩

lemmas *write-projs* = *F-write D-write T-write*
and *write0-projs* = *F-write0 D-write0 T-write0*

12.5.2 Equality with Constant Process

STOP

lemma *read-is-STOP-iff* : ‹*c*?*a*∈*A* → *P a* = *STOP* ⟷ *A* = {}›
⟨*proof*⟩

lemma *read-empty* [*simp*] : ‹*c*?*a*∈{} → *P a* = *STOP*› ⟨*proof*⟩

lemma *ndet-write-is-STOP-iff* : ‹*c*!!*a*∈*A* → *P a* = *STOP* ⟷ *A* = {}›
⟨*proof*⟩

lemma *ndet-write-empty* [*simp*] : ‹*c*!!*a*∈{} → *P a* = *STOP*› ⟨*proof*⟩

lemma *write0-neq-STOP* [*simp*] : ‹*a* → *P* ≠ *STOP*› ⟨*proof*⟩

lemma *write-neq-STOP* [*simp*] : ‹*c*!*a* → *P* ≠ *STOP*› ⟨*proof*⟩

SKIP

lemma *read-neq-SKIP* [*simp*] : ‹*c*?*a*∈*A* → *P a* ≠ *SKIP r*› ⟨*proof*⟩

lemma *ndet-write-neq-SKIP* [*simp*] : ‹*c*!!*a*∈*A* → *P a* ≠ *SKIP r*› ⟨*proof*⟩

lemma *write0-neq-SKIP* [*simp*] : ‹*a* → *P* ≠ *SKIP r*› ⟨*proof*⟩

lemma *write-neq-SKIP* [*simp*] : ‹*c*!*a* → *P* ≠ *SKIP r*› ⟨*proof*⟩

⊥

lemma *read-neq-BOT* [*simp*] : ‹*c*?*a*∈*A* → *P a* ≠ ⊥› ⟨*proof*⟩

lemma *ndet-write-neq-BOT* [*simp*] : ‹*c*!!*a*∈*A* → *P a* ≠ ⊥› ⟨*proof*⟩

lemma *write0-neq-BOT* [*simp*] : ‹*a* → *P* ≠ ⊥› ⟨*proof*⟩

lemma *write-neq-BOT* [*simp*] : ‹*c*!*a* → *P* ≠ ⊥› ⟨*proof*⟩

12.5.3 Extensions of Step-Laws

Monotony for equality

lemma *mono-read-eq* :
‹(⋀*a*. *a* ∈ *A* ⟹ *P a* = *Q a*) ⟹ *read c A P* = *read c A Q*›
⟨*proof*⟩

lemma *mono-ndet-write-eq* :
‹$(\bigwedge a.\ a \in A \Longrightarrow P\ a = Q\ a) \Longrightarrow$ *ndet-write c A P* = *ndet-write c A Q*›
⟨*proof*⟩

(□) and (⊓)

lemma *read-Ndet-read* :
‹$(c?a{\in}A \to P\ a) \sqcap (c?b{\in}B \to Q\ b) =$
$(c?a{\in}(A - B) \to P\ a) \sqcap (c?b{\in}(B - A) \to Q\ b) \mathbin{\square} (c?x{\in}(A \cap B) \to P\ x \sqcap Q\ x)$›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*inj-on c* $(A \cup B)$›
⟨*proof*⟩

lemma *read-Det-read* :
‹$(c?a{\in}A \to P\ a) \mathbin{\square} (c?b{\in}B \to Q\ b) =$
$c?a{\in}(A \cup B) \to$ (*if* $a \in A \cap B$ *then* $P\ a \sqcap Q\ a$ *else if* $a \in A$ *then* $P\ a$ *else* $Q\ a$)›
(**is** ‹*?lhs* = *?rhs*›) **if** ‹*inj-on c* $(A \cup B)$›
⟨*proof*⟩

lemma *ndet-write-Det-ndet-write* :
‹$(c!!a{\in}A \to P\ a) \mathbin{\square} (c!!b{\in}B \to Q\ b) =$
(*if* $A = \{\}$ *then* $(c!!b{\in}B \to Q\ b)$
else if $B = \{\}$ *then* $(c!!a{\in}A \to P\ a)$
else $\sqcap a{\in}A.\ \sqcap b{\in}B.$ (*if* $a = b$ *then* $c!a \to P\ a \sqcap Q\ a$ *else* $(c!a \to P\ a) \mathbin{\square} (c!b \to Q\ b)$)))›
if ‹*inj-on c* $(A \cup B)$›
⟨*proof*⟩

lemma *ndet-write-Ndet-ndet-write* :
‹$(c!!a{\in}A \to P\ a) \sqcap (c!!b{\in}B \to Q\ b) =$
(*if* $A = B$ *then* $(c!!b{\in}B \to P\ b \sqcap Q\ b)$
else if $A \subseteq B$
then $(c!!b{\in}(B - A) \to Q\ b) \sqcap (c!!a{\in}A \to P\ a \sqcap Q\ a)$
else if $B \subseteq A$
then $(c!!a{\in}(A - B) \to P\ a) \sqcap (c!!b{\in}B \to P\ b \sqcap Q\ b)$
else $(c!!a{\in}(A - B) \to P\ a) \sqcap (c!!b{\in}(B - A) \to Q\ b) \sqcap (c!!a{\in}(A \cap B) \to P\ a \sqcap Q\ a)$)›
if ‹$A \cap B \neq \{\}$› ‹*inj-on c* $(A \cup B)$›
⟨*proof*⟩

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

lemma *write0-Det-write0-is-write0-Ndet-write0* : ‹$(a \to P) \mathbin{\square} (a \to Q) = (a \to P) \sqcap (a \to Q)$›

⟨*proof*⟩

lemma *write-Ndet-write* : ‹$(c!a \rightarrow P) \sqcap (c!a \rightarrow Q) = c!a \rightarrow P \sqcap Q$›
⟨*proof*⟩

lemma *write-Det-write-is-write-Ndet-write*: ‹$(c!a \rightarrow P) \,\Box\, (c!a \rightarrow Q) = (c!a \rightarrow P) \sqcap (c!a \rightarrow Q)$›
⟨*proof*⟩

lemma *write-Ndet-read* :
‹$(c!a \rightarrow P) \sqcap (c?b{\in}B \rightarrow Q\ b) =$
$(\textit{if } a \in B \textit{ then } STOP \textit{ else } c!a \rightarrow P) \sqcap (c?b{\in}(B - \{a\}) \rightarrow Q\ b) \,\Box\, (\textit{if } a \in B \textit{ then } c!a \rightarrow P \sqcap Q\ a \textit{ else } STOP)$›
if ‹*inj-on c* $(\{a\} \cup B)$›
⟨*proof*⟩

lemma *read-Ndet-write* :
‹*inj-on c* $(A \cup \{b\}) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \sqcap (c!b \rightarrow Q) =$
$(\textit{if } b \in A \textit{ then } STOP \textit{ else } c!b \rightarrow Q) \sqcap (c?a{\in}(A - \{b\}) \rightarrow P\ a) \,\Box\, (\textit{if } b \in A \textit{ then } c!b \rightarrow P\ b \sqcap Q \textit{ else } STOP)$›
⟨*proof*⟩

lemma *write0-Ndet-read* :
‹$(a \rightarrow P) \sqcap (id?b{\in}B \rightarrow Q\ b) =$
$(\textit{if } a \in B \textit{ then } STOP \textit{ else } a \rightarrow P) \sqcap (id?b{\in}(B - \{a\}) \rightarrow Q\ b) \,\Box\, (\textit{if } a \in B \textit{ then } a \rightarrow P \sqcap Q\ a \textit{ else } STOP)$›
⟨*proof*⟩

lemma *read-Ndet-write0* :
‹$(id?a{\in}A \rightarrow P\ a) \sqcap (b \rightarrow Q) =$
$(\textit{if } b \in A \textit{ then } STOP \textit{ else } b \rightarrow Q) \sqcap (id?a{\in}(A - \{b\}) \rightarrow P\ a) \,\Box\, (\textit{if } b \in A \textit{ then } b \rightarrow P\ b \sqcap Q \textit{ else } STOP)$›
⟨*proof*⟩

lemma *write-Det-read* :
‹*inj-on c* $(\textit{insert } a\ B) \Longrightarrow (c!a \rightarrow P) \,\Box\, (c?b{\in}B \rightarrow Q\ b) =$
$c?b{\in}(\textit{insert } a\ B) \rightarrow (\textit{if } b = a \wedge a \in B \textit{ then } P \sqcap Q\ a \textit{ else if } b = a \textit{ then } P \textit{ else } Q\ b)$›
⟨*proof*⟩

lemma *read-Det-write* :
‹*inj-on c* $(\textit{insert } b\ A) \Longrightarrow (c?a{\in}A \rightarrow P\ a) \,\Box\, (c!b \rightarrow Q) =$
$c?a{\in}(\textit{insert } b\ A) \rightarrow (\textit{if } a = b \wedge b \in A \textit{ then } P\ a \sqcap Q \textit{ else if } a = b \textit{ then } Q \textit{ else } P\ a)$›
⟨*proof*⟩

lemma *write0-Det-read* :
‹$(a \to P) \;\square\; (id?b{\in}B \to Q\ b) =$
$id?b{\in}(insert\ a\ B) \to (if\ b = a \wedge a \in B\ then\ P \sqcap Q\ a\ else\ if\ b = a\ then\ P\ else\ Q\ b)$›
⟨*proof*⟩

lemma *read-Det-write0* :
‹$(id?a{\in}A \to P\ a) \;\square\; (b \to Q) =$
$id?a{\in}(insert\ b\ A) \to (if\ a = b \wedge b \in A\ then\ P\ a \sqcap Q\ else\ if\ a = b\ then\ Q\ else\ P\ a)$›
⟨*proof*⟩

Sliding

lemma *write0-Sliding-write0* :
‹$(a \to P) \rhd (b \to Q) =$
$(\square x \in \{a, b\} \to (if\ a = b\ then\ P \sqcap Q\ else\ if\ x = a\ then\ P\ else\ Q)) \sqcap$
$(b \to (if\ a = b\ then\ P \sqcap Q\ else\ Q))$›
⟨*proof*⟩

lemma *write-Sliding-write* :
‹$(c!a \to P) \rhd (d!b \to Q) =$
$(\square x \in \{c\ a, d\ b\} \to (if\ c\ a = d\ b\ then\ P \sqcap Q\ else\ if\ x = c\ a\ then\ P\ else\ Q)) \sqcap$
$(d!b \to (if\ c\ a = d\ b\ then\ P \sqcap Q\ else\ Q))$›
⟨*proof*⟩

lemma *write0-Sliding-write* :
‹$(a \to P) \rhd (d!b \to Q) =$
$(\square x \in \{a, d\ b\} \to (if\ a = d\ b\ then\ P \sqcap Q\ else\ if\ x = a\ then\ P\ else\ Q)) \sqcap$
$(d!b \to (if\ a = d\ b\ then\ P \sqcap Q\ else\ Q))$›
⟨*proof*⟩

lemma *write-Sliding-write0* :
‹$(c!a \to P) \rhd (b \to Q) =$
$(\square x \in \{c\ a, b\} \to (if\ c\ a = b\ then\ P \sqcap Q\ else\ if\ x = c\ a\ then\ P\ else\ Q)) \sqcap$
$(b \to (if\ c\ a = b\ then\ P \sqcap Q\ else\ Q))$›
⟨*proof*⟩

lemma *read-Sliding-superset-read* :
— Not really interesting without the additional assumptions.
‹$A \subseteq B \Longrightarrow inj\text{-}on\ c\ B \Longrightarrow$
$(c?a{\in}A \to P\ a) \rhd (c?b{\in}B \to Q\ b) = c?b{\in}B \to (if\ b \in A\ then\ P\ b \sqcap Q\ b\ else\ Q\ b)$›
⟨*proof*⟩

lemma *read-Sliding-same-set-read* :
‹$inj\text{-}on\ c\ A \Longrightarrow (c?a{\in}A \to P\ a) \rhd (c?a{\in}A \to Q\ a) = c?a{\in}A \to P\ a \sqcap Q\ a$›

⟨*proof*⟩

lemma *ndet-write-Sliding-superset-ndet-write* :
‹$A \subseteq B \Longrightarrow$ *inj-on c B* $\Longrightarrow$
$(c!!a{\in}A \rightarrow P\ a) \rhd (c!!b{\in}B \rightarrow Q\ b) = c!!b{\in}B \rightarrow$ (*if* $b \in A$ *then* $P\ b \sqcap Q\ b$ *else* $Q\ b$)›
⟨*proof*⟩

lemma *ndet-write-Sliding-same-set-ndet-write* :
‹*inj-on c A* $\Longrightarrow (c!!a{\in}A \rightarrow P\ a) \rhd (c!!a{\in}A \rightarrow Q\ a) = c!!a{\in}A \rightarrow P\ a \sqcap Q\ a$›
⟨*proof*⟩

lemma *write-Sliding-superset-read* :
‹$a \in B \Longrightarrow$ *inj-on c B* $\Longrightarrow$
$(c!a \rightarrow P) \rhd (c?b{\in}B \rightarrow Q\ b) = c?b{\in}B \rightarrow$ (*if* $b = a$ *then* $P \sqcap Q\ b$ *else* $Q\ b$)›
⟨*proof*⟩

lemma *write0-Sliding-superset-read* :
‹$a \in B \Longrightarrow (a \rightarrow P) \rhd (id?b{\in}B \rightarrow Q\ b) = id?b{\in}B \rightarrow$ (*if* $b = a$ *then* $P \sqcap Q\ b$ *else* $Q\ b$)›
⟨*proof*⟩

lemma *write-Sliding-superset-ndet-write* :
‹$a \in B \Longrightarrow$ *inj-on c B* $\Longrightarrow$
$(c!a \rightarrow P) \rhd (c!!b{\in}B \rightarrow Q\ b) = c!!b{\in}B \rightarrow$ (*if* $b = a$ *then* $P \sqcap Q\ b$ *else* $Q\ b$)›
⟨*proof*⟩

lemma *write0-Sliding-superset-ndet-write* :
‹$a \in B \Longrightarrow (a \rightarrow P) \rhd (id!!b{\in}B \rightarrow Q\ b) = id!!b{\in}B \rightarrow$ (*if* $b = a$ *then* $P \sqcap Q\ b$ *else* $Q\ b$)›
⟨*proof*⟩

Seq

lemma *read-Seq* : ‹$c?a{\in}A \rightarrow P\ a\ ;\ Q = c?a{\in}A \rightarrow (P\ a\ ;\ Q)$›
⟨*proof*⟩

lemma *write0-Seq* : ‹$a \rightarrow P\ ;\ Q = a \rightarrow (P\ ;\ Q)$›
⟨*proof*⟩

lemma *ndet-write-Seq* : ‹$c!!a{\in}A \rightarrow P\ a\ ;\ Q = c!!a{\in}A \rightarrow (P\ a\ ;\ Q)$›
⟨*proof*⟩

lemma *write-Seq* : ‹$c!a \rightarrow P\ ;\ Q = c!a \rightarrow (P\ ;\ Q)$›
⟨*proof*⟩

Renaming

lemma *Renaming-read* :

‹*Renaming* (*c*?*a*∈*A* → *P a*) *f g* = (*f* ∘ *c*)?*a*∈*A* → *Renaming* (*P a*) *f g*›
if ‹*inj-on c A*› **and** ‹*inj-on f* (*c* ‘ *A*)›
⟨*proof*⟩

lemma *Renaming-write* :
‹*Renaming* (*c*!*a* → *P*) *f g* = (*f* ∘ *c*)!*a* → *Renaming P f g*›
⟨*proof*⟩

lemma *Renaming-write0* :
‹*Renaming* (*a* → *P*) *f g* = *f a* → *Renaming P f g*›
⟨*proof*⟩

lemma *Renaming-ndet-write* :
‹*Renaming* (*c*!!*a*∈*A* → *P a*) *f g* = (*f* ∘ *c*)!!*a*∈*A* → *Renaming* (*P a*) *f g*›
if ‹*inj-on c A*› **and** ‹*inj-on f* (*c* ‘ *A*)›
⟨*proof*⟩

(\)

lemma *Hiding-read-disjoint* :
‹*c* ‘ *A* ∩ *S* = {} ⟹ *c*?*a*∈*A* → *P a* \ *S* = *c*?*a*∈*A* → (*P a* \ *S*)›
⟨*proof*⟩

lemma ‹*A* ⊆ *B* ⟹ *a* ∈ *A* ⟹ (*inv-into A c*) *a* = (*inv-into B c*) *a*›
⟨*proof*⟩

lemma *Hiding-read-non-disjoint* :
‹*c*?*a*∈*A* → *P a* \ *S* = (*c*?*a*∈(*A* − *c* −‘ *S*) → (*P a* \ *S*)) ▷ (⊓*a*∈(*A* ∩ *c* −‘ *S*).
(*P a* \ *S*))›
if ‹*inj-on c A*› **and** ‹*c* ‘ *A* ∩ *S* ≠ {}›
⟨*proof*⟩

lemma *Hiding-read-subset* :
‹*c*?*a*∈*A* → *P a* \ *S* = ⊓*a*∈(*c* ‘ *A* ∩ *S*). (*P* (*inv-into A c a*) \ *S*)›
if ‹*inj-on c A*› **and** ‹*c* ‘ *A* ⊆ *S*›
⟨*proof*⟩

lemma *Hiding-ndet-write-disjoint* :
‹*c* ‘ *A* ∩ *S* = {} ⟹ (*c*!!*a*∈*A* → *P a*) \ *S* = (*c*!!*a*∈*A* → (*P a* \ *S*))›
⟨*proof*⟩

lemma *Hiding-ndet-write-subset* :
‹*c* ‘ *A* ⊆ *S* ⟹ (*c*!!*a*∈*A* → *P a*) \ *S* = ⊓*a*∈*c* ‘ *A*. (*P* (*inv-into A c a*) \ *S*)›
⟨*proof*⟩

lemma *Hiding-ndet-write-subset-bis* :
— With the injectivity...

‹*inj-on c A* ⟹ *c ‘ A* ⊆ *S* ⟹ (*c*!!*a*∈*A* → *P a*) \ *S* = ⊓*a*∈*A*. (*P a* \ *S*)›
⟨*proof*⟩

lemma ‹*A* ⊆ *B* ⟹ *a* ∈ *A* ⟹ (*inv-into A c*) *a* = (*inv-into B c*) *a*›
⟨*proof*⟩

lemma *Hiding-ndet-write-non-disjoint-not-subset* :
‹(*c*!!*a*∈*A* → *P a*) \ *S* =
(*c*!!*a*∈(*A* − *c* −‘ *S*) → (*P a* \ *S*)) ⊓ (⊓*a*∈(*A* ∩ *c* −‘ *S*). (*P a* \ *S*))›
if ‹*inj-on c A*› **and** ‹*c ‘ A* ∩ *S* ≠ {}› **and** ‹¬ *c ‘ A* ⊆ *S*›
⟨*proof*⟩

lemma *Hiding-write0-disjoint* :
‹*a* ∉ *S* ⟹ *a* → *P* \ *S* = *a* → (*P* \ *S*)›
⟨*proof*⟩

lemma *Hiding-write0-non-disjoint* :
‹*a* ∈ *S* ⟹ *a* → *P* \ *S* = *P* \ *S*›
⟨*proof*⟩

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

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

Sync

read **lemma** *read-Sync-read* :
— This is the general case.
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* =
(*c*?*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b*)) □
(*d*?*b*∈(*B* − *d* −‘ *S*) → (*c*?*a*∈*A* → *P a* ⟦*S*⟧ *Q b*)) □
(□*x*∈(*c ‘ A* ∩ *d ‘ B* ∩ *S*) → (*P* (*inv-into A c x*) ⟦*S*⟧ *Q* (*inv-into B d x*)))›
(**is** ‹*?lhs* = *?rhs1* □ *?rhs2* □ *?rhs3*›)
if ‹*c ‘ A* ∩ *S* = {} ∨ *c ‘ A* ⊆ *S* ∨ *inj-on c A*›
‹*d ‘ B* ∩ *S* = {} ∨ *d ‘ B* ⊆ *S* ∨ *inj-on d B*›
— Assumptions may seem strange, but the motivation is that when *A* − *c* −‘ *S* ≠ {} (which is equivalent to ¬ *c ‘ A* ⊆ *S*), we need to ensure that *inv-into* (*A* − *c* −‘ *S*) *c* is equal to *inv-into A c*. This requires *A* − *c* −‘ *S* = *A* (which is equivalent to *c ‘ A* ∩ *S* = {}) or *inj-on c A*. We need obviously a similar assumption for *B*.
⟨*proof*⟩

Enforce read **lemma** *read-Sync-read-forced-read-left* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* =

$(c?a{\in}(A - c -` S) \to (P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b))\ \Box$
$(d?b{\in}(B - d -` S) \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b))\ \Box$
$(c?x{\in}(A \cap c -` (d ` B \cap S)) \to (P\ x\ [\![S]\!]\ Q\ x))\rangle$
(**is** ‹*?lhs* = *?rhs1* □ *?rhs2* □ *?rhs3*›)
if ‹$c ` A \cap S = \{\} \vee$ *inj-on c A*›
‹$d ` B \cap S = \{\} \vee$ *inj-on d B*›
‹$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b$›
⟨*proof*⟩

lemma *read-Sync-read-forced-read-right*:
‹$[\![c ` A \cap S = \{\} \vee$ *inj-on c A*; $d ` B \cap S = \{\} \vee$ *inj-on d B*;
$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b]\!] \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c?a{\in}(A - c -` S) \to (P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b))\ \Box$
$(d?b{\in}(B - d -` S) \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b))\ \Box$
$(d?x{\in}(B \cap d -` (c ` A \cap S)) \to (P\ x\ [\![S]\!]\ Q\ x))\rangle$
⟨*proof*⟩

Special Cases **lemma** *read-Sync-read-subset* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$\Box x{\in}(c ` A \cap d ` B) \to (P\ (\textit{inv-into}\ A\ c\ x)\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ x))\rangle$
if ‹$c ` A \subseteq S\rangle$ ‹$d ` B \subseteq S\rangle$
⟨*proof*⟩

lemma *read-Sync-read-subset-forced-read-left* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = c?x{\in}(A \cap c -` d ` B) \to (P\ x\ [\![S]\!]\ Q\ x)\rangle$
if ‹$c ` A \subseteq S\rangle$ ‹$d ` B \subseteq S\rangle$ ‹*inj-on c A*› ‹*inj-on d B*›
‹$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b\rangle$
⟨*proof*⟩

lemma *read-Sync-read-subset-forced-read-right* :
‹$[\![c ` A \subseteq S$; $d ` B \subseteq S$; *inj-on c A*; *inj-on d B*;
$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow d\ b \in S \Longrightarrow a = b]\!] \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?x{\in}(B \cap d -` c ` A) \to (P\ x\ [\![S]\!]\ Q\ x)\rangle$
⟨*proof*⟩

lemma *read-Sync-read-indep* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c?a{\in}A \to (P\ a\ [\![S]\!]\ (d?b{\in}B \to Q\ b))) \Box (d?b{\in}B \to ((c?a{\in}A \to P\ a)\ [\![S]\!]\ Q\ b))\rangle$
if ‹$c ` A \cap S = \{\}\rangle$ ‹$d ` B \cap S = \{\}\rangle$
⟨*proof*⟩

lemma *read-Sync-read-left* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = c?a{\in}A \to (P\ a\ [\![S]\!]\ (d?b{\in}B \to Q\ b))\rangle$
if ‹$c ` A \cap S = \{\}\rangle$ ‹$d ` B \subseteq S\rangle$
⟨*proof*⟩

lemma *read-Sync-read-right* :
‹c ‘ $A \subseteq S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?b{\in}B \to ((c?a{\in}A \to P\ a)\ [\![S]\!]\ Q\ b)$›
⟨*proof*⟩

corollary *read-Par-read* :
‹$c?a{\in}A \to P\ a \parallel d?b{\in}B \to Q\ b =$
$\Box x{\in}(c$ ‘ $A \cap d$ ‘ $B) \to (P\ (\textit{inv-into}\ A\ c\ x) \parallel Q\ (\textit{inv-into}\ B\ d\ x))$›
⟨*proof*⟩

corollary *read-Par-read-forced-read-left* :
‹$[\![\textit{inj-on}\ c\ A;\ \textit{inj-on}\ d\ B;\ \bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow a = b]\!] \Longrightarrow$
$c?a{\in}A \to P\ a \parallel d?b{\in}B \to Q\ b = c?x{\in}(A \cap c -$‘ d ‘ $B) \to (P\ x \parallel Q\ x)$›
⟨*proof*⟩

corollary *read-Par-read-forced-read-right* :
‹$[\![\textit{inj-on}\ c\ A;\ \textit{inj-on}\ d\ B;\ \bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow a = b]\!] \Longrightarrow$
$c?a{\in}A \to P\ a \parallel d?b{\in}B \to Q\ b = d?x{\in}(B \cap d -$‘ c ‘ $A) \to (P\ x \parallel Q\ x)$›
⟨*proof*⟩

corollary *read-Inter-read* :
‹$[\![\textit{inj-on}\ c\ A;\ \textit{inj-on}\ d\ B;\ \bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = d\ b \Longrightarrow a = b]\!] \Longrightarrow$
$c?a{\in}A \to P\ a \mid\mid\mid d?b{\in}B \to Q\ b =$
$(c?a{\in}A \to (P\ a \mid\mid\mid d?b{\in}B \to Q\ b)) \Box (d?b{\in}B \to (c?a{\in}A \to P\ a \mid\mid\mid Q\ b))$›
⟨*proof*⟩

Same channel Some results can be rewritten when we have the same channel.

lemma *read-Sync-read-forced-read-same-chan* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b =$
$(c?a{\in}(A - c -$‘ $S) \to (P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b)) \Box$
$(c?b{\in}(B - c -$‘ $S) \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)) \Box$
$(c?x{\in}(A \cap B \cap c -$‘ $S) \to (P\ x\ [\![S]\!]\ Q\ x))$›
(**is** ‹*?lhs = ?rhs1* □ *?rhs2* □ *?rhs3*›)
if ‹c ‘ $A \cap S = \{\} \vee$ *inj-on c A*› ‹c ‘ $B \cap S = \{\} \vee$ *inj-on c B*›
‹$\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow c\ a = c\ b \Longrightarrow c\ b \in S \Longrightarrow a = b$›
⟨*proof*⟩

lemma *read-Sync-read-forced-read-same-chan-weaker* :
— Easier with a stronger assumption.
‹*inj-on c* $(A \cup B) \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b =$
$(c?a{\in}(A - c -$‘ $S) \to (P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b)) \Box$
$(c?b{\in}(B - c -$‘ $S) \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)) \Box$
$(c?x{\in}(A \cap B \cap c -$‘ $S) \to (P\ x\ [\![S]\!]\ Q\ x))$›

⟨*proof*⟩

lemma *read-Sync-read-subset-forced-read-same-chan* :
— In the subset case, the assumption *inj-on c* $(A \cup B)$ is equivalent. The result is not weaker anymore.
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b = c?x{\in}(A \cap B) \to (P\ x\ [\![S]\!]\ Q\ x)$›
if ‹$c\ `\ A \subseteq S$› ‹$c\ `\ B \subseteq S$› ‹*inj-on c* $(A \cup B)$›
⟨*proof*⟩

read **and** *ndet-write*. **lemma** *ndet-write-Sync-read* :
‹$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
(*if* $A = \{\}$ *then STOP* $[\![S]\!]\ d?b{\in}B \to Q\ b$
else $\sqcap a{\in}c\ `\ A.$ (*if* $a \in S$ *then STOP else* $a \to (P$ (*inv-into A c a*) $[\![S]\!]\ d?b{\in}B \to Q\ b)) \Box$
$(\Box b{\in}(d\ `\ B - S) \to (a \to P$ (*inv-into A c a*) $[\![S]\!]\ Q$ (*inv-into B d b*))) $\Box$
(*if* $a \in d\ `\ B \cap S$ *then* $a \to (P$ (*inv-into A c a*) $[\![S]\!]\ Q$ (*inv-into B d a*)) *else STOP*))›
⟨*proof*⟩

lemma *read-Sync-ndet-write* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
(*if* $B = \{\}$ *then* $c?a{\in}A \to P\ a\ [\![S]\!]$ *STOP*
else $\sqcap b{\in}d\ `\ B.$ (*if* $b \in S$ *then STOP else* $b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q$ (*inv-into B d b*))) $\Box$
$(\Box a{\in}(c\ `\ A - S) \to (P$ (*inv-into A c a*) $[\![S]\!]\ b \to Q$ (*inv-into B d b*))) $\Box$
(*if* $b \in c\ `\ A \cap S$ *then* $b \to (P$ (*inv-into A c b*) $[\![S]\!]\ Q$ (*inv-into B d b*)) *else STOP*))›
⟨*proof*⟩

lemma *ndet-write-Sync-read-subset* :
‹$c\ `\ A \subseteq S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
(*if* $c\ `\ A \subseteq d\ `\ B$ *then* $\sqcap a{\in}c\ `\ A \to (P$ (*inv-into A c a*) $[\![S]\!]\ Q$ (*inv-into B d a*))
else $(\sqcap a{\in}(c\ `\ A \cap d\ `\ B) \to (P$ (*inv-into A c a*) $[\![S]\!]\ Q$ (*inv-into B d a*))) $\sqcap$ *STOP*)›
⟨*proof*⟩

lemma *read-Sync-ndet-write-subset* :
‹$c\ `\ A \subseteq S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
(*if* $d\ `\ B \subseteq c\ `\ A$ *then* $\sqcap b{\in}d\ `\ B \to (P$ (*inv-into A c b*) $[\![S]\!]\ Q$ (*inv-into B d b*))
else $(\sqcap b{\in}(c\ `\ A \cap d\ `\ B) \to (P$ (*inv-into A c b*) $[\![S]\!]\ Q$ (*inv-into B d b*))) $\sqcap$ *STOP*)›
⟨*proof*⟩
lemma *ndet-write-Sync-read-subset-same-chan*:

‹$c!!a{\in}A \to P\ a\ [\![S]\!]\ c?b{\in}B \to Q\ b =$
($if\ A \subseteq B\ then\ c!!a{\in}A \to (P\ a\ [\![S]\!]\ Q\ a)\ else\ (c!!a{\in}(A \cap B) \to (P\ a\ [\![S]\!]\ Q\ a))$
$\sqcap\ STOP$)›
if ‹c ‘ $A \subseteq S$› ‹c ‘ $B \subseteq S$› ‹$inj\text{-}on\ c\ (A \cup B)$›
⟨*proof*⟩

lemma *read-Sync-ndet-write-subset-same-chan*:
‹c ‘ $A \subseteq S \Longrightarrow c$ ‘ $B \subseteq S \Longrightarrow inj\text{-}on\ c\ (A \cup B) \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ c!!b{\in}B \to Q\ b =$
($if\ B \subseteq A\ then\ c!!b{\in}B \to (P\ b\ [\![S]\!]\ Q\ b)\ else\ (c!!b{\in}(A \cap B) \to (P\ b\ [\![S]\!]\ Q\ b))$
$\sqcap\ STOP$)›
⟨*proof*⟩

lemma *ndet-write-Sync-read-indep* :
‹c ‘ $A \cap S = \{\} \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
($if\ A = \{\}\ then\ d?b{\in}B \to\ (STOP\ [\![S]\!]\ Q\ b)$
$else\ \sqcap a{\in}c$ ‘ $A.\ (a \to (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ d?b{\in}B \to Q\ b)) \Box$
$(d?b{\in}B \to (a \to P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ b)))$›
⟨*proof*⟩

lemma *read-Sync-ndet-write-indep* :
‹c ‘ $A \cap S = \{\} \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
($if\ B = \{\}\ then\ c?a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$
$else\ \sqcap b{\in}d$ ‘ $B.\ (b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ b))) \Box$
$(c?a{\in}A \to (P\ a\ [\![S]\!]\ b \to Q\ (inv\text{-}into\ B\ d\ b))))$›
⟨*proof*⟩

lemma *ndet-write-Sync-read-left* :
‹$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = c!!a{\in}A \to (P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b)$›
(**is** ‹$?lhs = ?rhs$›) **if** ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
⟨*proof*⟩

lemma *read-Sync-ndet-write-left* :
‹$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b = c?a{\in}A \to (P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b)$›
(**is** ‹$?lhs = ?rhs$›) **if** ‹c ‘ $A \cap S = \{\}$› ‹d ‘ $B \subseteq S$›
⟨*proof*⟩

lemma *ndet-write-Sync-read-right* :
‹c ‘ $A \subseteq S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?b{\in}B \to (c!!a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)$›
⟨*proof*⟩

lemma *read-Sync-ndet-write-right* :
‹c ‘ $A \subseteq S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow$

$c?a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b = d!!b{\in}B \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)\rangle$
⟨*proof*⟩

read **and** *write*. **lemma** *write-Sync-read* :
$\langle c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(\mathit{if}\ c\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ c!a \to (P\ [\![S]\!]\ d?b{\in}B \to Q\ b)) \Box$
$(\Box b{\in}(d\ `\ B - S) \to (c!a \to P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ b))) \Box$
$(\mathit{if}\ c\ a \in d\ `\ B \cap S\ \mathit{then}\ c!a \to (P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ (c\ a)))\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write* :
$\langle c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$
$(\mathit{if}\ d\ b \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ d!b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q)) \Box$
$(\Box a{\in}(c\ `\ A - S) \to (P\ (\mathit{inv\text{-}into}\ A\ c\ a)\ [\![S]\!]\ d!b \to Q)) \Box$
$(\mathit{if}\ d\ b \in c\ `\ A \cap S\ \mathit{then}\ d!b \to (P\ (\mathit{inv\text{-}into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *write-Sync-read-subset* :
$\langle c\ a \in S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow$
$c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(\mathit{if}\ c\ a \in d\ `\ B\ \mathit{then}\ c!a \to (P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ (c\ a)))\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write-subset* :
$\langle c\ `\ A \subseteq S \Longrightarrow d\ b \in S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$
$(\mathit{if}\ d\ b \in c\ `\ A\ \mathit{then}\ d!b \to (P\ (\mathit{inv\text{-}into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩
lemma *write-Sync-read-subset-same-chan*:
$\langle c\ a \in S \Longrightarrow c\ `\ B \subseteq S \Longrightarrow \mathit{inj\text{-}on}\ c\ (\mathit{insert}\ a\ B) \Longrightarrow$
$c!a \to P\ [\![S]\!]\ c?b{\in}B \to Q\ b = (\mathit{if}\ a \in B\ \mathit{then}\ c!a \to (P\ [\![S]\!]\ Q\ a)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write-subset-same-chan*:
$\langle c\ `\ A \subseteq S \Longrightarrow c\ b \in S \Longrightarrow \mathit{inj\text{-}on}\ c\ (\mathit{insert}\ b\ A) \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ c!b \to Q = (\mathit{if}\ b \in A\ \mathit{then}\ c!b \to (P\ b\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *write-Sync-read-indep* :
$\langle c\ a \notin S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b =$
$(c!a \to (P\ [\![S]\!]\ d?b{\in}B \to Q\ b)) \Box (d?b{\in}B \to (c!a \to P\ [\![S]\!]\ Q\ b))\rangle$
⟨*proof*⟩

lemma *read-Sync-write-indep* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow d\ b \notin S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$

$(d!b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q)) \Box (c?a{\in}A \to (P\ a\ [\![S]\!]\ d!b \to Q))\rangle$
⟨*proof*⟩

lemma *write-Sync-read-left* :
$\langle c\ a \notin S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow$
$c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b = c!a \to (P\ [\![S]\!]\ d?b{\in}B \to Q\ b)\rangle$
⟨*proof*⟩

lemma *read-Sync-write-left* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow d\ b \in S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q = c?a{\in}A \to (P\ a\ [\![S]\!]\ d!b \to Q)\rangle$
⟨*proof*⟩

lemma *write-Sync-read-right* :
$\langle c\ a \in S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!a \to P\ [\![S]\!]\ d?b{\in}B \to Q\ b = d?b{\in}B \to (c!a \to P\ [\![S]\!]\ Q\ b)\rangle$
⟨*proof*⟩

lemma *read-Sync-write-right* :
$\langle c\ `\ A \subseteq S \Longrightarrow d\ b \notin S \Longrightarrow$
$c?a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q = d!b \to (c?a{\in}A \to P\ a\ [\![S]\!]\ Q)\rangle$
⟨*proof*⟩

ndet-write **and** *ndet-write* **lemma** *ndet-write-Sync-ndet-write* :
$\langle c!!a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
$($ *if* $A = \{\}$ *then* *if* $d\ `\ B \cap S = \{\}$ *then* $d!!b{\in}B \to (STOP\ [\![S]\!]\ Q\ b)$
else $(\sqcap x{\in}d\ `\ (B - d -` S) \to (STOP\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ x))) \sqcap$
$STOP$
else *if* $B = \{\}$ *then* *if* $c\ `\ A \cap S = \{\}$ *then* $c!!a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$
else $(\sqcap x{\in}c\ `\ (A - c -` S) \to (P\ (inv\text{-}into\ A\ c\ x)\ [\![S]\!]\ STOP))$
$\sqcap\ STOP$
else $\sqcap b{\in}d\ `\ B.\ \sqcap a{\in}c\ `\ A.$
$(if\ a \in S\ then\ STOP\ else\ a \to (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ b \to Q\ (inv\text{-}into$
$B\ d\ b))) \Box$
$(if\ b \in S\ then\ STOP\ else\ b \to (a \to P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ (inv\text{-}into$
$B\ d\ b))) \Box$
$(if\ a = b \wedge b \in S\ then\ b \to (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ b))$
$else\ STOP))\rangle$
(**is** ‹*?lhs* = (*if* $A = \{\}$ *then* *if* $d\ `\ B \cap S = \{\}$ *then* *?mv-right* *else* *?mv-right′* $\sqcap$
$STOP$
else *if* $B = \{\}$ *then* *if* $c\ `\ A \cap S = \{\}$ *then* *?mv-left* *else* *?mv-left′* $\sqcap$
$STOP$
else *?huge-mess*)›)
⟨*proof*⟩

lemma *ndet-write-Sync-ndet-write-subset* :

$\langle c!!a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
$($ *if* $\exists\, b.\ c\ `\ A = \{b\} \wedge d\ `\ B = \{b\}$
then $(THE\ b.\ d\ `\ B = \{b\}) \to (P\ (inv\text{-}into\ A\ c\ (THE\ a.\ c\ `\ A = \{a\}))\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ (THE\ b.\ d\ `\ B = \{b\})))$
else $(\sqcap x{\in}(c\ `\ A \cap d\ `\ B) \to (P\ (inv\text{-}into\ A\ c\ x)\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ x))) \sqcap STOP)\rangle$
(**is** $\langle ?lhs = (if\ ?cond\ then\ ?rhs1\ else\ ?rhs2)\rangle$) **if** $\langle c\ `\ A \subseteq S\rangle\ \langle d\ `\ B \subseteq S\rangle$
⟨*proof*⟩

lemma *ndet-write-Sync-ndet-write-indep* :
$\langle c\ `\ A \cap S = \{\} \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
$($ *if* $A = \{\}$ *then* $d!!b{\in}B \to (STOP\ [\![S]\!]\ Q\ b)$
else if $B = \{\}$ *then* $c!!a{\in}A \to (P\ a\ [\![S]\!]\ STOP)$
else $\sqcap b{\in}d\ `\ B.\ \sqcap a{\in}c\ `\ A.$
$((a \to (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ b \to Q\ (inv\text{-}into\ B\ d\ b)))) \Box$
$((b \to (a \to P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ b))))\rangle$
⟨*proof*⟩

lemma *ndet-write-Sync-ndet-write-left* :
$\langle c!!a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b = c!!a{\in}A \to (P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b)\rangle$
(**is** $\langle ?lhs = ?rhs\rangle$) **if** $\langle c\ `\ A \cap S = \{\}\rangle\ \langle d\ `\ B \subseteq S\rangle$
⟨*proof*⟩

lemma *ndet-write-Sync-ndet-write-right* :
$\langle c\ `\ A \subseteq S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!!a{\in}A \to P\ a\ [\![S]\!]\ d!!b{\in}B \to Q\ b = d!!b{\in}B \to (c!!a{\in}A \to P\ a\ [\![S]\!]\ Q\ b)\rangle$
⟨*proof*⟩

ndet-write **and** *write* **lemma** *write-Sync-ndet-write* :
$\langle c!a \to P\ [\![S]\!]\ d!!b{\in}B \to Q\ b =$
$($ *if* $B = \{\}$ *then* $c!a \to P\ [\![S]\!]\ STOP$
else $\sqcap b{\in}d\ `\ B.\ (if\ b \in S\ then\ STOP\ else\ b \to (c!a \to P\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ b))) \Box$
$(if\ c\ a \in S\ then\ STOP\ else\ c!a \to (P\ [\![S]\!]\ b \to Q\ (inv\text{-}into\ B\ d\ b))) \Box$
$(if\ b = c\ a \wedge c\ a \in S\ then\ c!a \to (P\ [\![S]\!]\ Q\ (inv\text{-}into\ B\ d\ (c\ a)))\ else\ STOP))\rangle$
⟨*proof*⟩

lemma *ndet-write-Sync-write* :
$\langle c!!a{\in}A \to P\ a\ [\![S]\!]\ d!b \to Q =$
$($ *if* $A = \{\}$ *then* $STOP\ [\![S]\!]\ d!b \to Q$
else $\sqcap a{\in}c\ `\ A.\ (if\ a \in S\ then\ STOP\ else\ a \to (P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ d!b \to Q)) \Box$
$(if\ d\ b \in S\ then\ STOP\ else\ d!b \to (a \to P\ (inv\text{-}into\ A\ c\ a)\ [\![S]\!]\ Q))$

$\Box$
(*if* $a = d\ b \wedge d\ b \in S$ *then* $d!b \rightarrow (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ Q)$ *else* *STOP*))›
⟨*proof*⟩

lemma *write-Sync-ndet-write-subset* :
‹$c!a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b =$
(*if* $c\ a \notin d\ `\ B$ *then STOP else if* $d\ `\ B = \{c\ a\}$ *then* $c!a \rightarrow (P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ (c\ a)))$
else $(c!a \rightarrow (P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ (c\ a)))) \sqcap STOP$)› **if** ‹$c\ a \in S$› ‹$d\ `\ B \subseteq S$›
⟨*proof*⟩

lemma *ndet-write-Sync-write-subset* :
‹$c\ `\ A \subseteq S \Longrightarrow d\ b \in S \Longrightarrow$
$(c!!a{\in}A \rightarrow P\ a)\ [\![S]\!]\ (d!b \rightarrow Q) =$
(*if* $d\ b \notin c\ `\ A$ *then STOP else if* $c\ `\ A = \{d\ b\}$ *then* $d!b \rightarrow (P\ (\textit{inv-into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)$
else $(d!b \rightarrow (P\ (\textit{inv-into}\ A\ c\ (d\ b))\ [\![S]\!]\ Q)) \sqcap STOP$)›
⟨*proof*⟩

lemma *write-Sync-ndet-write-indep* :
‹$c\ a \notin S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b =$
(*if* $B = \{\}$ *then* $c!a \rightarrow (P\ [\![S]\!]\ STOP)$
else $\sqcap b{\in}d\ `\ B.\ (c!a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q\ (\textit{inv-into}\ B\ d\ b))) \sqcap$
$(b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q\ (\textit{inv-into}\ B\ d\ b))))$›
⟨*proof*⟩

lemma *ndet-write-Sync-write-indep* :
‹$c\ `\ A \cap S = \{\} \Longrightarrow d\ b \notin S \Longrightarrow$
$c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q =$
(*if* $A = \{\}$ *then* $d!b \rightarrow (STOP\ [\![S]\!]\ Q)$
else $\sqcap a{\in}c\ `\ A.\ (a \rightarrow (P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ d!b \rightarrow Q)) \sqcap$
$(d!b \rightarrow (a \rightarrow P\ (\textit{inv-into}\ A\ c\ a)\ [\![S]\!]\ Q)))$›
⟨*proof*⟩

lemma *write-Sync-ndet-write-left* :
‹$c\ a \notin S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b = c!a \rightarrow (P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b)$›
⟨*proof*⟩

lemma *ndet-write-Sync-write-left* :
‹$c\ `\ A \cap S = \{\} \Longrightarrow d\ b \in S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q = c!!a{\in}A \rightarrow (P\ a\ [\![S]\!]\ d!b \rightarrow Q)$›
⟨*proof*⟩

lemma *write-Sync-ndet-write-right* :
‹$c\ a \in S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b = d!!b{\in}B \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q\ b)$›
⟨*proof*⟩

lemma *ndet-write-Sync-write-right* :
‹$c\ `\ A \subseteq S \Longrightarrow d\ b \notin S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)$›
⟨*proof*⟩

write **and** *write* **lemma** *write-Sync-write* :
‹$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q =$
$(\mathit{if}\ d\ b \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q)) \Box$
$(\mathit{if}\ c\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)) \Box$
$(\mathit{if}\ c\ a = d\ b \wedge d\ b \in S\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})$›
⟨*proof*⟩

lemma *write-Inter-write* :
‹$c!a \rightarrow P\ |||\ d!b \rightarrow Q = (c!a \rightarrow (P\ |||\ d!b \rightarrow Q)) \Box (d!b \rightarrow (c!a \rightarrow P\ |||\ Q))$›
⟨*proof*⟩

lemma *write-Par-write* :
‹$c!a \rightarrow P\ ||\ d!b \rightarrow Q = (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ c!a \rightarrow (P\ ||\ Q)\ \mathit{else}\ \mathit{STOP})$›
⟨*proof*⟩

lemma *write-Sync-write-subset* :
‹$c\ a \in S \Longrightarrow d\ b \in S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (\mathit{if}\ c\ a = d\ b\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})$›
⟨*proof*⟩

lemma *write-Sync-write-indep* :
‹$c\ a \notin S \Longrightarrow d\ b \notin S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)) \Box (d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q))$›
⟨*proof*⟩

lemma *write-Sync-write-left* :
‹$c\ a \notin S \Longrightarrow d\ b \in S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = c!a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)$›
⟨*proof*⟩

lemma *write-Sync-write-right* :
‹$c\ a \in S \Longrightarrow d\ b \notin S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q)$›
⟨*proof*⟩

read **and** $(\rightarrow)$**.** **lemma** *write0-Sync-read* :
$\langle a \rightarrow P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b =$
$(\mathit{if}\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ a \rightarrow (P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b)) \ \Box$
$(\Box b{\in}(d \ `\ B - S) \rightarrow (a \rightarrow P \ [\![S]\!] \ Q \ (\mathit{inv\text{-}into}\ B\ d\ b))) \ \Box$
$(\mathit{if}\ a \in d \ `\ B \cap S\ \mathit{then}\ a \rightarrow (P \ [\![S]\!] \ Q \ (\mathit{inv\text{-}into}\ B\ d\ a))\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write0* :
$\langle c?a{\in}A \rightarrow P\ a \ [\![S]\!] \ b \rightarrow Q =$
$(\mathit{if}\ b \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ b \rightarrow (c?a{\in}A \rightarrow P\ a \ [\![S]\!] \ Q)) \ \Box$
$(\Box a{\in}(c \ `\ A - S) \rightarrow (P \ (\mathit{inv\text{-}into}\ A\ c\ a) \ [\![S]\!] \ b \rightarrow Q)) \ \Box$
$(\mathit{if}\ b \in c \ `\ A \cap S\ \mathit{then}\ b \rightarrow (P \ (\mathit{inv\text{-}into}\ A\ c\ b) \ [\![S]\!] \ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *write0-Sync-read-subset* :
$\langle a \in S \Longrightarrow d \ `\ B \subseteq S \Longrightarrow$
$a \rightarrow P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b =$
$(\mathit{if}\ a \in d \ `\ B\ \mathit{then}\ a \rightarrow (P \ [\![S]\!] \ Q \ (\mathit{inv\text{-}into}\ B\ d\ a))\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write0-subset* :
$\langle c \ `\ A \subseteq S \Longrightarrow b \in S \Longrightarrow$
$c?a{\in}A \rightarrow P\ a \ [\![S]\!] \ b \rightarrow Q =$
$(\mathit{if}\ b \in c \ `\ A\ \mathit{then}\ b \rightarrow (P \ (\mathit{inv\text{-}into}\ A\ c\ b) \ [\![S]\!] \ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *write0-Sync-read-subset-same-chan*:
$\langle a \in S \Longrightarrow B \subseteq S \Longrightarrow$
$a \rightarrow P \ [\![S]\!] \ id?b{\in}B \rightarrow Q \ b = (\mathit{if}\ a \in B\ \mathit{then}\ a \rightarrow (P \ [\![S]\!] \ Q\ a)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *read-Sync-write0-subset-same-chan*:
$\langle A \subseteq S \Longrightarrow b \in S \Longrightarrow$
$id?a{\in}A \rightarrow P\ a \ [\![S]\!] \ b \rightarrow Q = (\mathit{if}\ b \in A\ \mathit{then}\ b \rightarrow (P\ b \ [\![S]\!] \ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
⟨*proof*⟩

lemma *write0-Sync-read-indep* :
$\langle a \notin S \Longrightarrow d \ `\ B \cap S = \{\} \Longrightarrow$
$a \rightarrow P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b =$
$(a \rightarrow (P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b)) \ \Box \ (d?b{\in}B \rightarrow (a \rightarrow P \ [\![S]\!] \ Q \ b))\rangle$
⟨*proof*⟩

lemma *read-Sync-write0-indep* :
$\langle c \ `\ A \cap S = \{\} \Longrightarrow b \notin S \Longrightarrow$
$c?a{\in}A \rightarrow P\ a \ [\![S]\!] \ b \rightarrow Q =$
$(b \rightarrow (c?a{\in}A \rightarrow P\ a \ [\![S]\!] \ Q)) \ \Box \ (c?a{\in}A \rightarrow (P\ a \ [\![S]\!] \ b \rightarrow Q))\rangle$
⟨*proof*⟩

lemma *write0-Sync-read-left* :

‹$a \notin S \Longrightarrow d$ ‘ $B \subseteq S \Longrightarrow a \rightarrow P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b = a \rightarrow (P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b)$›
⟨*proof*⟩

lemma *read-Sync-write0-left* :
‹c ‘ $A \cap S = \{\} \Longrightarrow b \in S \Longrightarrow c?a{\in}A \rightarrow P \ a \ [\![S]\!] \ b \rightarrow Q = c?a{\in}A \rightarrow (P \ a \ [\![S]\!] \ b \rightarrow Q)$›
⟨*proof*⟩

lemma *write0-Sync-read-right* :
‹$a \in S \Longrightarrow d$ ‘ $B \cap S = \{\} \Longrightarrow a \rightarrow P \ [\![S]\!] \ d?b{\in}B \rightarrow Q \ b = d?b{\in}B \rightarrow (a \rightarrow P \ [\![S]\!] \ Q \ b)$›
⟨*proof*⟩

lemma *read-Sync-write0-right* :
‹c ‘ $A \subseteq S \Longrightarrow b \notin S \Longrightarrow c?a{\in}A \rightarrow P \ a \ [\![S]\!] \ b \rightarrow Q = b \rightarrow (c?a{\in}A \rightarrow P \ a \ [\![S]\!] \ Q)$›
⟨*proof*⟩

ndet-write **and** $(\rightarrow)$ **lemma** *write0-Sync-ndet-write* :
‹$a \rightarrow P \ [\![S]\!] \ d!!b{\in}B \rightarrow Q \ b =$
(*if* $B = \{\}$ *then* $a \rightarrow P \ [\![S]\!]$ *STOP*
else $\sqcap b{\in}d$ ‘ B. (*if* $b \in S$ *then STOP else* $b \rightarrow (a \rightarrow P \ [\![S]\!] \ Q$ (*inv-into* $B \ d \ b$))) $\Box$
(*if* $a \in S$ *then STOP else* $a \rightarrow (P \ [\![S]\!] \ b \rightarrow Q$ (*inv-into* $B \ d \ b$))) $\Box$
(*if* $b = a \wedge a \in S$ *then* $a \rightarrow (P \ [\![S]\!] \ Q$ (*inv-into* $B \ d \ a$)) *else STOP*))›
⟨*proof*⟩

lemma *ndet-write-Sync-write0* :
‹$c!!a{\in}A \rightarrow P \ a \ [\![S]\!] \ b \rightarrow Q =$
(*if* $A = \{\}$ *then STOP* $[\![S]\!] \ b \rightarrow Q$
else $\sqcap a{\in}c$ ‘ A. (*if* $a \in S$ *then STOP else* $a \rightarrow (P$ (*inv-into* $A \ c \ a$) $[\![S]\!] \ b \rightarrow Q)$) $\Box$
(*if* $b \in S$ *then STOP else* $b \rightarrow (a \rightarrow P$ (*inv-into* $A \ c \ a$) $[\![S]\!] \ Q)$) $\Box$
(*if* $a = b \wedge b \in S$ *then* $b \rightarrow (P$ (*inv-into* $A \ c \ a$) $[\![S]\!] \ Q$) *else STOP*))›
⟨*proof*⟩

lemma *write0-Sync-ndet-write-subset* :
‹$a \in S \Longrightarrow d$ ‘ $B \subseteq S \Longrightarrow$
$a \rightarrow P \ [\![S]\!] \ d!!b{\in}B \rightarrow Q \ b =$
(*if* $a \notin d$ ‘ B *then STOP else if* d ‘ $B = \{a\}$ *then* $a \rightarrow (P \ [\![S]\!] \ Q$ (*inv-into* $B \ d \ a$))
else $(a \rightarrow (P \ [\![S]\!] \ Q$ (*inv-into* $B \ d \ a$))) $\sqcap$ *STOP*)›
⟨*proof*⟩

lemma *ndet-write-Sync-write0-subset* :
‹c ‘ $A \subseteq S \Longrightarrow b \in S \Longrightarrow$
$c!!a{\in}A \rightarrow P \ a \ [\![S]\!] \ b \rightarrow Q =$
(*if* $b \notin c$ ‘ A *then STOP else if* c ‘ $A = \{b\}$ *then* $b \rightarrow (P$ (*inv-into* $A \ c \ b$) $[\![S]\!]$

Q)
  else $(b \rightarrow (P\ (\mathit{inv\text{-}into}\ A\ c\ b)\ [\![S]\!]\ Q)) \sqcap \mathit{STOP})$›
 ⟨*proof*⟩

lemma *write0-Sync-ndet-write-indep* :
 ‹$a \notin S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow$
  $a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b =$
  (*if* $B = \{\}$ *then* $a \rightarrow (P\ [\![S]\!]\ \mathit{STOP})$
   else $\sqcap b{\in}d\ `\ B.\ (a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q\ (\mathit{inv\text{-}into}\ B\ d\ b))) \Box$
                $(b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q\ (\mathit{inv\text{-}into}\ B\ d\ b))))$›
 ⟨*proof*⟩

lemma *ndet-write-Sync-write0-indep* :
 ‹$c\ `\ A \cap S = \{\} \Longrightarrow b \notin S \Longrightarrow$
  $c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ b \rightarrow Q =$
  (*if* $A = \{\}$ *then* $b \rightarrow (\mathit{STOP}\ [\![S]\!]\ Q)$
   else $\sqcap a{\in}c\ `\ A.\ (a \rightarrow (P\ (\mathit{inv\text{-}into}\ A\ c\ a)\ [\![S]\!]\ b \rightarrow Q)) \Box$
                $(b \rightarrow (a \rightarrow P\ (\mathit{inv\text{-}into}\ A\ c\ a)\ [\![S]\!]\ Q)))$›
 ⟨*proof*⟩

lemma *write0-Sync-ndet-write-left* :
 ‹$a \notin S \Longrightarrow d\ `\ B \subseteq S \Longrightarrow a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b = a \rightarrow (P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b)$›
 ⟨*proof*⟩

lemma *ndet-write-Sync-write0-left* :
 ‹$c\ `\ A \cap S = \{\} \Longrightarrow b \in S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ b \rightarrow Q = c!!a{\in}A \rightarrow (P\ a\ [\![S]\!]\ b \rightarrow Q)$›
 ⟨*proof*⟩

lemma *write-Sync-ndet-write0-right* :
 ‹$a \in S \Longrightarrow d\ `\ B \cap S = \{\} \Longrightarrow a \rightarrow P\ [\![S]\!]\ d!!b{\in}B \rightarrow Q\ b = d!!b{\in}B \rightarrow (a \rightarrow P\ [\![S]\!]\ Q\ b)$›
 ⟨*proof*⟩

lemma *ndet-write-Sync-write0-right* :
 ‹$c\ `\ A \subseteq S \Longrightarrow b \notin S \Longrightarrow c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ b \rightarrow Q = b \rightarrow (c!!a{\in}A \rightarrow P\ a\ [\![S]\!]\ Q)$›
 ⟨*proof*⟩

($\rightarrow$) and ($\rightarrow$) **lemma** *write0-Sync-write0* :
 ‹$a \rightarrow P\ [\![S]\!]\ b \rightarrow Q =$
  $(\mathit{if}\ b \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q)) \Box$
  $(\mathit{if}\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q)) \Box$
  $(\mathit{if}\ a = b \wedge b \in S\ \mathit{then}\ a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})$›
 ⟨*proof*⟩

lemma *write0-Sync-write0-bis* :
 ‹$(a \rightarrow P)\ [\![S]\!]\ (b \rightarrow Q) =$

(*if* $a \in S$
then *if* $b \in S$
then *if* $a = b$
then $a \rightarrow (P \llbracket S \rrbracket Q)$
else $STOP$
else $(b \rightarrow ((a \rightarrow P) \llbracket S \rrbracket Q))$
else *if* $b \in S$
then $a \rightarrow (P \llbracket S \rrbracket (b \rightarrow Q))$
else $(a \rightarrow (P \llbracket S \rrbracket (b \rightarrow Q))) \Box (b \rightarrow ((a \rightarrow P) \llbracket S \rrbracket Q)))$›
⟨*proof*⟩

lemma *write0-Inter-write0* :
‹$a \rightarrow P \mid\mid\mid b \rightarrow Q = (a \rightarrow (P \mid\mid\mid b \rightarrow Q)) \Box (b \rightarrow (a \rightarrow P \mid\mid\mid Q))$›
⟨*proof*⟩

lemma *write0-Par-write0* :
‹$a \rightarrow P \parallel b \rightarrow Q = (\textit{if } a = b \textit{ then } a \rightarrow (P \parallel Q) \textit{ else } STOP)$›
⟨*proof*⟩

lemma *write0-Sync-write0-subset* :
‹$a \in S \Longrightarrow b \in S \Longrightarrow a \rightarrow P \llbracket S \rrbracket b \rightarrow Q = (\textit{if } a = b \textit{ then } a \rightarrow (P \llbracket S \rrbracket Q) \textit{ else } STOP)$›
⟨*proof*⟩

lemma *write0-Sync-write0-indep* :
‹$a \notin S \Longrightarrow b \notin S \Longrightarrow a \rightarrow P \llbracket S \rrbracket b \rightarrow Q = (a \rightarrow (P \llbracket S \rrbracket b \rightarrow Q)) \Box (b \rightarrow (a \rightarrow P \llbracket S \rrbracket Q))$›
⟨*proof*⟩

lemma *write0-Sync-write0-left* :
‹$a \notin S \Longrightarrow b \in S \Longrightarrow a \rightarrow P \llbracket S \rrbracket b \rightarrow Q = a \rightarrow (P \llbracket S \rrbracket b \rightarrow Q)$›
⟨*proof*⟩

lemma *write0-Sync-write0-right* :
‹$a \in S \Longrightarrow b \notin S \Longrightarrow a \rightarrow P \llbracket S \rrbracket b \rightarrow Q = b \rightarrow (a \rightarrow P \llbracket S \rrbracket Q)$›
⟨*proof*⟩

write **and** $(\rightarrow)$ **lemma** *write0-Sync-write* :
‹$a \rightarrow P \llbracket S \rrbracket d!b \rightarrow Q =$
$(\textit{if } d\ b \in S \textit{ then } STOP \textit{ else } d!b \rightarrow (a \rightarrow P \llbracket S \rrbracket Q)) \Box$
$(\textit{if } a \in S \textit{ then } STOP \textit{ else } a \rightarrow (P \llbracket S \rrbracket d!b \rightarrow Q)) \Box$
$(\textit{if } a = d\ b \wedge d\ b \in S \textit{ then } a \rightarrow (P \llbracket S \rrbracket Q) \textit{ else } STOP)$›
⟨*proof*⟩

lemma *write-Sync-write0* :
‹$c!a \rightarrow P \llbracket S \rrbracket b \rightarrow Q =$
$(\textit{if } b \in S \textit{ then } STOP \textit{ else } b \rightarrow (c!a \rightarrow P \llbracket S \rrbracket Q)) \Box$

$(\mathit{if}\ c\ a \in S\ \mathit{then}\ \mathit{STOP}\ \mathit{else}\ c!a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q)) \Box$
$(\mathit{if}\ c\ a = b \wedge b \in S\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
$\langle \mathit{proof} \rangle$

lemma *write0-Sync-write-subset* :
$\langle a \in S \Longrightarrow d\ b \in S \Longrightarrow$
$a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (\mathit{if}\ a = d\ b\ \mathit{then}\ a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
$\langle \mathit{proof} \rangle$

lemma *write-Sync-write0-subset* :
$\langle c\ a \in S \Longrightarrow b \in S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = (\mathit{if}\ c\ a = b\ \mathit{then}\ c!a \rightarrow (P\ [\![S]\!]\ Q)\ \mathit{else}\ \mathit{STOP})\rangle$
$\langle \mathit{proof} \rangle$

lemma *write0-Sync-write-indep* :
$\langle a \notin S \Longrightarrow d\ b \notin S \Longrightarrow$
$a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = (a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)) \Box (d!b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q))\rangle$
$\langle \mathit{proof} \rangle$

lemma *write-Sync-write0-indep* :
$\langle c\ a \notin S \Longrightarrow b \notin S \Longrightarrow$
$c!a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = (c!a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q)) \Box (b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q))\rangle$
$\langle \mathit{proof} \rangle$

lemma *write0-Sync-write-left* :
$\langle a \notin S \Longrightarrow d\ b \in S \Longrightarrow a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = a \rightarrow (P\ [\![S]\!]\ d!b \rightarrow Q)\rangle$
$\langle \mathit{proof} \rangle$

lemma *write-Sync-write0-left* :
$\langle c\ a \notin S \Longrightarrow b \in S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = c!a \rightarrow (P\ [\![S]\!]\ b \rightarrow Q)\rangle$
$\langle \mathit{proof} \rangle$

lemma *write0-Sync-write-right* :
$\langle a \in S \Longrightarrow d\ b \notin S \Longrightarrow a \rightarrow P\ [\![S]\!]\ d!b \rightarrow Q = d!b \rightarrow (a \rightarrow P\ [\![S]\!]\ Q)\rangle$
$\langle \mathit{proof} \rangle$

lemma *write-Sync-write0-right* :
$\langle c\ a \in S \Longrightarrow b \notin S \Longrightarrow c!a \rightarrow P\ [\![S]\!]\ b \rightarrow Q = b \rightarrow (c!a \rightarrow P\ [\![S]\!]\ Q)\rangle$
$\langle \mathit{proof} \rangle$

12.5.4 *Sync*, *SKIP* and *STOP*

SKIP

Without injectivity, the result is a trivial corollary of *read ?c ?A ?P ≡ Mprefix (?c ‘ ?A) (?P ∘ inv-into ?A ?c)* and *Mprefix ?A ?P ⟦?S⟧ SKIP ?res = □a∈(?A − ?S) → (?P a ⟦?S⟧ SKIP ?res)*.

lemma *read-Sync-SKIP* :
‹*c?a∈A → P a ⟦S⟧ SKIP r = c?a∈(A − c −‘ S) → (P a ⟦S⟧ SKIP r)*› **if** ‹*inj-on c A*›
⟨*proof*⟩

lemma *SKIP-Sync-read* :
‹*inj-on d B ⟹ SKIP r ⟦S⟧ d?b∈B → Q b = d?b∈(B − d −‘ S) → (SKIP r ⟦S⟧ Q b)*›
⟨*proof*⟩

corollary *write-Sync-SKIP* :
‹*c!a → P ⟦S⟧ SKIP r = (if c a ∈ S then STOP else c!a → (P ⟦S⟧ SKIP r))*›
and *SKIP-Sync-write* :
‹*SKIP r ⟦S⟧ d!b → Q = (if d b ∈ S then STOP else d!b → (SKIP r ⟦S⟧ Q))*›
⟨*proof*⟩

corollary *write0-Sync-SKIP* :
‹*a → P ⟦S⟧ SKIP r = (if a ∈ S then STOP else a → (P ⟦S⟧ SKIP r))*›
and *SKIP-Sync-write0* :
‹*SKIP r ⟦S⟧ b → Q = (if b ∈ S then STOP else b → (SKIP r ⟦S⟧ Q))*›
⟨*proof*⟩

lemma *ndet-write-Sync-SKIP* :
‹*c!!a∈A → P a ⟦S⟧ SKIP r =*
(if c ‘ A ∩ S = {} then c!!a∈A → (P a ⟦S⟧ SKIP r)
else (c!!a∈(A − c −‘ S) → (P a ⟦S⟧ SKIP r)) ⊓ STOP)›
(**is** ‹*?lhs = (if c ‘ A ∩ S = {} then ?rhs1 else ?rhs2 ⊓ STOP)*›) **if** ‹*inj-on c A*›
⟨*proof*⟩

corollary *SKIP-Sync-ndet-write* :
‹*inj-on d B ⟹*
SKIP r ⟦S⟧ d!!b∈B → Q b =
(if d ‘ B ∩ S = {} then d!!b∈B → (SKIP r ⟦S⟧ Q b)
else (d!!b∈(B − d −‘ S) → (SKIP r ⟦S⟧ Q b)) ⊓ STOP)›
⟨*proof*⟩

STOP

Without injectivity, the result is a trivial corollary of *read ?c ?A ?P ≡ Mprefix (?c ‘ ?A) (?P ∘ inv-into ?A ?c)* and *Mprefix ?A ?P ⟦?S⟧ SKIP*

?res = □*a*∈(*?A* − *?S*) → (*?P a* ⟦*?S*⟧ *SKIP ?res*).

lemma *read-Sync-STOP* :
‹*c*?*a*∈*A* → *P a* ⟦*S*⟧ *STOP* = *c*?*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *STOP*)› **if** ‹*inj-on c A*›
⟨*proof*⟩

lemma *STOP-Sync-read* :
‹*inj-on d B* ⟹ *STOP* ⟦*S*⟧ *d*?*b*∈*B* → *Q b* = *d*?*b*∈(*B* − *d* −‘ *S*) → (*STOP* ⟦*S*⟧ *Q b*)›
⟨*proof*⟩

corollary *write-Sync-STOP* :
‹*c*!*a* → *P* ⟦*S*⟧ *STOP* = (*if c a* ∈ *S then STOP else c*!*a* → (*P* ⟦*S*⟧ *STOP*))›
and *STOP-Sync-write* :
‹*STOP* ⟦*S*⟧ *d*!*b* → *Q* = (*if d b* ∈ *S then STOP else d*!*b* → (*STOP* ⟦*S*⟧ *Q*))›
⟨*proof*⟩

corollary *write0-Sync-STOP* :
‹*a* → *P* ⟦*S*⟧ *STOP* = (*if a* ∈ *S then STOP else a* → (*P* ⟦*S*⟧ *STOP*))›
and *STOP-Sync-write0* :
‹*STOP* ⟦*S*⟧ *b* → *Q* = (*if b* ∈ *S then STOP else b* → (*STOP* ⟦*S*⟧ *Q*))›
⟨*proof*⟩

lemma *ndet-write-Sync-STOP* :
‹*c*!!*a*∈*A* → *P a* ⟦*S*⟧ *STOP* =
(*if c* ‘ *A* ∩ *S* = {} *then c*!!*a*∈*A* → (*P a* ⟦*S*⟧ *STOP*)
else (*c*!!*a*∈(*A* − *c* −‘ *S*) → (*P a* ⟦*S*⟧ *STOP*)) ⊓ *STOP*)›
(**is** ‹*?lhs* = (*if c* ‘ *A* ∩ *S* = {} *then ?rhs1 else ?rhs2* ⊓ *STOP*)›) **if** ‹*inj-on c A*›
⟨*proof*⟩

corollary *STOP-Sync-ndet-write* :
‹*inj-on d B* ⟹
STOP ⟦*S*⟧ *d*!!*b*∈*B* → *Q b* =
(*if d* ‘ *B* ∩ *S* = {} *then d*!!*b*∈*B* → (*STOP* ⟦*S*⟧ *Q b*)
else (*d*!!*b*∈(*B* − *d* −‘ *S*) → (*STOP* ⟦*S*⟧ *Q b*)) ⊓ *STOP*)›
⟨*proof*⟩

12.6 Powerful Laws of CSP

12.6.1 Laws for Mndetprefix and Sync

lemma *Mndetprefix-Sync-Det-distr-FD* :
‹(⊓ *a* ∈ *A* → (*P a* ⟦ *C* ⟧ (⊓ *b* ∈ *B* → *Q b*))) □
(⊓ *b* ∈ *B* → ((⊓ *a* ∈ *A* → *P a*) ⟦ *C* ⟧ *Q b*))
⊑*FD* (⊓ *a* ∈ *A* → *P a*) ⟦ *C* ⟧ (⊓ *b* ∈ *B* → *Q b*)›
(**is** ‹*?lhs1* □ *?lhs2* ⊑*FD* *?rhs*›)
if ‹*A* ≠ {}› ‹*B* ≠ {}› ‹*A* ∩ *C* = {}› ‹*B* ∩ *C* = {}›

⟨*proof*⟩

lemmas *Mndetprefix-Sync-Det-distr-F* = *Mndetprefix-Sync-Det-distr-FD*[*THEN leFD-imp-leF*]
and *Mndetprefix-Sync-Det-distr-D* = *Mndetprefix-Sync-Det-distr-FD*[*THEN leFD-imp-leD*]

lemmas *Mndetprefix-Sync-Det-distr-T* = *Mndetprefix-Sync-Det-distr-F*[*THEN leF-imp-leT*]

lemma *Mndetprefix-Sync-Det-distr-DT* :
‹⟦$A \neq \{\}$; $B \neq \{\}$; $A \cap C = \{\}$; $B \cap C = \{\}$⟧ ⟹
($\sqcap\ a \in A \to (P\ a \llbracket\ C\ \rrbracket (\sqcap\ b \in B \to Q\ b))) \Box$
($\sqcap\ b \in B \to ((\sqcap\ a \in A \to P\ a) \llbracket\ C\ \rrbracket\ Q\ b)$)
$\sqsubseteq_{DT}$ ($\sqcap\ a \in A \to P\ a$) ⟦ C ⟧ ($\sqcap\ b \in B \to Q\ b$)›
⟨*proof*⟩

12.6.2 Hiding Operator Laws

theorem *Hiding-Hiding-less-eq-process-Hiding-Un* : ‹$P \setminus A \setminus B \sqsubseteq_{FD} P \setminus (A \cup B)$›
⟨*proof*⟩

theorem *Hiding-Un* : ‹$P \setminus (A \cup B) = P \setminus A \setminus B$›
if ‹*finite A*› **for** P :: ‹('*a*, '*r*) $process_{ptick}$›
⟨*proof*⟩

12.6.3 Sync Operator Laws

Preliminaries

lemma *tickFree-isInfHiddenRun* : ‹*tF s*›
if ‹*isInfHiddenRun f P A*› **and** ‹*s* ∈ *range f*›
⟨*proof*⟩

lemma *Hiding-interleave*:
‹*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*:
‹*s setinterleaves* ((*trace-hide t A*, *trace-hide u A*), *C*)
⟹ ∃ *r*. *s* = *trace-hide r A* ∧ *r setinterleaves* ((*t*, *u*), *C*)› **if** ‹*A* ∩ *C* = {}›
⟨*proof*⟩

lemma *le-trace-hide* : ‹$u \leq$ *trace-hide* $t\ S \Longrightarrow \exists\, u'.\ u =$ *trace-hide* $u'\ S \wedge u' \leq t$›
⟨*proof*⟩

lemma *append-interleave* :
‹*s1 setinterleaves* ((*t1*, *u1*), *S*) ⟹ *s2 setinterleaves* ((*t2*, *u2*), *S*) ⟹
(*s1* @ *s2*) *setinterleaves* ((*t1* @ *t2*, *u1* @ *u2*), *S*)›
⟨*proof*⟩

The Theorem: Sync and Hiding

theorem *Hiding-Sync* : ‹$(P\ [\![S]\!]\ Q) \setminus A = (P \setminus A)\ [\![S]\!]\ (Q \setminus A)$›
if ‹*finite A*› **and** ‹$A \cap S = \{\}$› **for** $P\ Q$ :: ‹$('a, 'r)$ $process_{ptick}$›
— Monster theorem!
⟨*proof*⟩

12.6.4 Renaming Operator Laws

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

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

lemma *Sliding-STOP-Det* [*simp*] : ‹$(P \rhd STOP) \mathbin{\square} Q = P \rhd Q$›
⟨*proof*⟩

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

lemma *Renaming-Sliding*:
‹*Renaming* $(P \rhd Q)\ f\ g =$ *Renaming* $P\ f\ g \rhd$ *Renaming* $Q\ f\ g$›
⟨*proof*⟩

lemma *Renaming-Mprefix-image*:
‹*Renaming* $(\mathbin{\square}\ a \in A \to P\ (f\ a))\ f\ g =$
$\mathbin{\square}\ b \in f\ `\ A \to$ *Renaming* $(P\ b)\ f\ g$› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *Renaming-Mprefix-image-inj-on*:

‹*Renaming* (*Mprefix A P*) *f g* = □ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f g*›
if *inj-on-f*: ‹*inj-on f A*›
⟨*proof*⟩

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

lemma *Renaming-Mndetprefix-image*: ‹*Renaming* (⊓ *a* ∈ *A* → *P* (*f a*)) *f g* = ⊓ *b* ∈ *f* ‘ *A* → *Renaming* (*P b*) *f g*›
⟨*proof*⟩

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

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

lemma *Hiding-distrib-FD-GlobalNdet* :
‹(⊓*a* ∈ *A*. *P a*) \ *S* ⊑$_{FD}$ ⊓*a* ∈ *A*. (*P a* \ *S*)› (**is** ‹*?lhs* ⊑$_{FD}$ *?rhs*›)
⟨*proof*⟩

lemma *Renaming-Seq* :
‹*Renaming* (*P* ; *Q*) *f g* = *Renaming P f g* ; *Renaming Q f g*› (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *Renaming-fix* :
‹*Renaming* (μ *X*. φ *X*) *f g* = (μ *X*. ((λ*P*. *Renaming P f g*) ∘ φ ∘ (λ*P*. *Renaming P* (*inv f*) (*inv g*))) *X*)›
(**is** ‹*Renaming* (μ *X*. φ *X*) *f g* = (μ *X*. *?φ*′ *X*)›) **if** ‹*cont* φ› **and** ‹*bij f*› **and** ‹*bij g*› **for** φ :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ ('*a*, '*r*) *process*$_{ptick}$›
⟨*proof*⟩

Chapter 13

Events and Ticks of a Process

13.1 Definitions

13.1.1 Events of a Process

definition *events-of* :: ‹('a, 'r) $process_{ptick}$ ⇒ 'a *set*› (‹α'(-')›)
— $\alpha(P)$ for "alphabet of P"
where ‹$\alpha(P) \equiv \bigcup t{\in}\mathcal{T}\ P.\ \{a.\ ev\ a \in set\ t\}$›

lemma *events-of-memI* : ‹$t \in \mathcal{T}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow a \in \alpha(P)$›
⟨*proof*⟩

lemma *events-of-memD* : ‹$a \in \alpha(P) \Longrightarrow \exists\, t \in \mathcal{T}\ P.\ ev\ a \in set\ t$›
⟨*proof*⟩

lemma *events-of-memE* :
‹$a \in \alpha(P) \Longrightarrow (\bigwedge t.\ t \in \mathcal{T}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow thesis) \Longrightarrow thesis$›
⟨*proof*⟩

definition *strict-events-of* :: ‹('a, 'r) $process_{ptick}$ ⇒ 'a *set*› (‹$\boldsymbol{\alpha}$'(-')›)
where ‹$\boldsymbol{\alpha}(P) \equiv \bigcup t{\in}\mathcal{T}\ P - \mathcal{D}\ P.\ \{a.\ ev\ a \in set\ t\}$›

lemma *strict-events-of-memI* :
‹$t \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow a \in \boldsymbol{\alpha}(P)$›
⟨*proof*⟩

lemma *strict-events-of-memD* : ‹$a \in \boldsymbol{\alpha}(P) \Longrightarrow \exists\, t \in \mathcal{T}\ P.\ t \notin \mathcal{D}\ P \wedge ev\ a \in set\ t$›
⟨*proof*⟩

lemma *strict-events-of-memE* :
‹$a \in \boldsymbol{\alpha}(P) \Longrightarrow (\bigwedge t.\ t \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow ev\ a \in set\ t \Longrightarrow thesis) \Longrightarrow$

thesis›
⟨*proof*⟩

lemma *events-of-is-strict-events-of-or-UNIV* :
‹$\alpha(P) = (\mathit{if}\ \mathcal{D}\ P = \{\}\ \mathit{then}\ \boldsymbol{\alpha}(P)\ \mathit{else}\ \mathit{UNIV})$›
⟨*proof*⟩

lemma *strict-events-of-subset-events-of* : ‹$\boldsymbol{\alpha}(P) \subseteq \alpha(P)$›
⟨*proof*⟩

13.1.2 Ticks of a Process

definition *ticks-of* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ '*r set*› (‹✓*s*'(-')›)
where ‹$\checkmark s(P) \equiv \{r.\ \exists\, t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P\}$›

lemma *ticks-of-memI* : ‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow r \in \checkmark s(P)$›
⟨*proof*⟩

lemma *ticks-of-memD* : ‹$r \in \checkmark s(P) \Longrightarrow \exists\, t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P$›
⟨*proof*⟩

lemma *ticks-of-memE* :
‹$r \in \checkmark s(P) \Longrightarrow (\bigwedge t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$›
⟨*proof*⟩

definition *strict-ticks-of* :: ‹('*a*, '*r*) *process*$_{ptick}$ ⇒ '*r set*› (‹✓**s**'(-')›)
where ‹$\checkmark\mathbf{s}(P) \equiv \{r.\ \exists\, s.\ s\ @\ [\checkmark(r)] \in \mathcal{T}\ P - \mathcal{D}\ P\}$›

lemma *strict-ticks-of-memI* :
‹$t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow t\ @\ [\checkmark(r)] \notin \mathcal{D}\ P \Longrightarrow r \in \checkmark\mathbf{s}(P)$›
⟨*proof*⟩

lemma *strict-ticks-of-memD* :
‹$r \in \checkmark\mathbf{s}(P) \Longrightarrow \exists\, t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \wedge t \notin \mathcal{D}\ P$›
⟨*proof*⟩

lemma *strict-ticks-of-memE* :
‹$r \in \checkmark\mathbf{s}(P) \Longrightarrow (\bigwedge t.\ t\ @\ [\checkmark(r)] \in \mathcal{T}\ P \Longrightarrow t \notin \mathcal{D}\ P \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$›
⟨*proof*⟩

lemma *ticks-of-is-strict-ticks-of-or-UNIV* :
‹$\checkmark s(P) = (\mathit{if}\ \mathcal{D}\ P = \{\}\ \mathit{then}\ \checkmark\mathbf{s}(P)\ \mathit{else}\ \mathit{UNIV})$›
⟨*proof*⟩

lemma *strict-ticks-of-subset-ticks-of* : ‹$\checkmark\mathbf{s}(P) \subseteq \checkmark s(P)$›
⟨*proof*⟩

13.2 Laws

13.2.1 Preliminaries

lemma *inj-on-map-map-event*$_{ptick}$*-T-tickFree* :
‹*inj-on* (*map* (*map-event*$_{ptick}$ *f g*)) {*t* ∈ $\mathcal{T}$ *P*. *tF t*}› **if** ‹*inj-on f* α(*P*)›
⟨*proof*⟩

lemma *inj-on-map-map-event*$_{ptick}$*-T* :
‹*inj-on* (*map* (*map-event*$_{ptick}$ *f g*)) ($\mathcal{T}$ *P*)› **if** ‹*inj-on f* α(*P*)› ‹*inj-on g* ✓*s*(*P*)›
⟨*proof*⟩

lemma *inj-on-map-map-event*$_{ptick}$*-T-diff-D-tickFree* :
‹*inj-on* (*map* (*map-event*$_{ptick}$ *f g*)) {*t* ∈ $\mathcal{T}$ *P* − $\mathcal{D}$ *P*. *tF t*}› **if** ‹*inj-on f* $\boldsymbol{\alpha}$(*P*)›
⟨*proof*⟩

lemma *inj-on-map-map-event*$_{ptick}$*-T-diff-D* :
‹*inj-on* (*map* (*map-event*$_{ptick}$ *f g*)) ($\mathcal{T}$ *P* − $\mathcal{D}$ *P*)› **if** ‹*inj-on f* $\boldsymbol{\alpha}$(*P*)› **and** ‹*inj-on g* ✓**s**(*P*)›
⟨*proof*⟩

13.2.2 Events of a Process

lemma *events-of-BOT* [*simp*] : ‹$\alpha(\bot)$ = *UNIV*›
and *events-of-SKIP* [*simp*] : ‹α(*SKIP r*) = {}›
and *events-of-STOP* [*simp*] : ‹α(*STOP*) = {}›
⟨*proof*⟩

lemma *anti-mono-events-of-T*: ‹*P* $\sqsubseteq_T$ *Q* ⟹ $\alpha(Q) \subseteq \alpha(P)$›
⟨*proof*⟩

lemma *anti-mono-events-of-F*: ‹*P* $\sqsubseteq_F$ *Q* ⟹ $\alpha(Q) \subseteq \alpha(P)$›
⟨*proof*⟩

lemma *anti-mono-events-of-FD*: ‹*P* $\sqsubseteq_{FD}$ *Q* ⟹ $\alpha(Q) \subseteq \alpha(P)$›
⟨*proof*⟩

lemma *anti-mono-events-of-DT*: ‹*P* $\sqsubseteq_{DT}$ *Q* ⟹ $\alpha(Q) \subseteq \alpha(P)$›
⟨*proof*⟩

lemma *anti-mono-events-of* : ‹*P* $\sqsubseteq$ *Q* ⟹ $\alpha(Q) \subseteq \alpha(P)$›
⟨*proof*⟩

lemma *events-of-GlobalNdet*: ‹$\alpha(\sqcap a \in A.\ P\ a) = (\bigcup a \in A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-write0* : ‹$\alpha(a \rightarrow P) = \mathit{insert}\ a\ \alpha(P)$›
⟨*proof*⟩

lemma *events-of-Mndetprefix*: ‹$\alpha(\sqcap a{\in}A \rightarrow P\ a) = A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-Mprefix*: ‹$\alpha(\Box a{\in}A \rightarrow P\ a) = A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-read* :
‹$\alpha(c\mathbf{?}a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}c\ `\ A.\ \alpha(P\ (\mathit{inv\text{-}into}\ A\ c\ a)))$›
⟨*proof*⟩

lemma *events-of-inj-on-read* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \alpha(c\mathbf{?}a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-ndet-write* :
‹$\alpha(c\mathbf{!!}a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}c\ `\ A.\ \alpha(P\ (\mathit{inv\text{-}into}\ A\ c\ a)))$›
⟨*proof*⟩

lemma *events-of-inj-on-ndet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \alpha(c\mathbf{!!}a{\in}A \rightarrow P\ a) = c\ `\ A \cup (\bigcup a{\in}A.\ \alpha(P\ a))$›
⟨*proof*⟩

lemma *events-of-write* : ‹$\alpha(c\mathbf{!}a \rightarrow P) = \mathit{insert}\ (c\ a)\ \alpha(P)$›
⟨*proof*⟩

lemma *events-of-Ndet*: ‹$\alpha(P \sqcap Q) = \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Det*: ‹$\alpha(P\ \Box\ Q) = \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Sliding*: ‹$\alpha(P \rhd Q) = \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Renaming*:
‹$\alpha(\mathit{Renaming}\ P\ f\ g) = (\mathit{if}\ \mathcal{D}\ P = \{\}\ \mathit{then}\ f\ `\ \alpha(P)\ \mathit{else}\ \mathit{UNIV})$›
⟨*proof*⟩

lemma *events-of-Seq* : ‹$\alpha(P\ ;\ Q) = \alpha(P) \cup (\mathit{if}\ \checkmark\mathbf{s}(P) = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ \alpha(Q))$›

(**is** ‹- = *?A*›)
⟨*proof*⟩

lemma *events-of-Sync-subset* : ‹$\alpha(P \ [\![S]\!] \ Q) \subseteq \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Inter*: ‹$\alpha((P :: ('a, 'r)\ process_{ptick}) \ ||| \ Q) = \alpha(P) \cup \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Par-div* :
‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P \neq \{\} \Longrightarrow \alpha(P \ ||\ Q) = UNIV$›
and *events-of-Par-subset* :
‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P = \{\} \Longrightarrow \alpha(P \ ||\ Q) \subseteq \alpha(P) \cap \alpha(Q)$›
⟨*proof*⟩

lemma *events-of-Hiding*:
‹$\alpha(P \setminus B) = (if\ \mathcal{D}\ (P \setminus B) = \{\}\ then\ \alpha(P) - B\ else\ UNIV)$›
⟨*proof*⟩

13.2.3 Strict Events of a Process

lemma *strict-events-of-BOT* [*simp*] : ‹$\boldsymbol{\alpha}(\bot) = \{\}$›
and *strict-events-of-SKIP* [*simp*] : ‹$\boldsymbol{\alpha}(SKIP\ r) = \{\}$›
and *strict-events-of-STOP* [*simp*] : ‹$\boldsymbol{\alpha}(STOP) = \{\}$›
⟨*proof*⟩

lemma *strict-events-of-GlobalNdet-subset* : ‹$\boldsymbol{\alpha}(\sqcap a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-Mprefix*:
‹$\boldsymbol{\alpha}(\Box a{\in}A \rightarrow P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-Mndetprefix*:
‹$\boldsymbol{\alpha}(\sqcap a{\in}A \rightarrow P\ a) = \{a{\in}A.\ P\ a \neq \bot\} \cup (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-write0* : ‹$\boldsymbol{\alpha}(a \rightarrow P) = (if\ P = \bot\ then\ \{\}\ else\ insert\ a\ \boldsymbol{\alpha}(P))$›
⟨*proof*⟩

lemma *strict-events-of-read* :
‹$\boldsymbol{\alpha}(c?a{\in}A \rightarrow P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\} \cup$
$(\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \boldsymbol{\alpha}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
⟨*proof*⟩

lemma *strict-events-of-inj-on-read* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \boldsymbol{\alpha}(c?a{\in}A \rightarrow P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ a \neq \bot\} \cup$
$(\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-ndet-write* :
‹$\boldsymbol{\alpha}(c!!a{\in}A \rightarrow P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\} \cup$
$(\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \boldsymbol{\alpha}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
⟨*proof*⟩

lemma *strict-events-of-inj-on-ndet-write* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \boldsymbol{\alpha}(c!!a{\in}A \rightarrow P\ a) = \{c\ a\ |a.\ a \in A \wedge P\ a \neq \bot\} \cup$
$(\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \boldsymbol{\alpha}(P\ a))$›
⟨*proof*⟩

lemma *strict-events-of-write* : ‹$\boldsymbol{\alpha}(c!a \rightarrow P) = (\mathit{if}\ P = \bot\ \mathit{then}\ \{\}\ \mathit{else}\ \mathit{insert}\ (c\ a)\ \boldsymbol{\alpha}(P))$›
⟨*proof*⟩

lemma *strict-events-of-Ndet-subset* : ‹$\boldsymbol{\alpha}(P \sqcap Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
⟨*proof*⟩

lemma *strict-events-of-Det-subset* : ‹$\boldsymbol{\alpha}(P \,\Box\, Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
⟨*proof*⟩

lemma *strict-events-of-Sliding-subset* : ‹$\boldsymbol{\alpha}(P \rhd Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
⟨*proof*⟩

lemma *strict-events-of-Renaming-subset* : ‹$\boldsymbol{\alpha}(\mathit{Renaming}\ P\ f\ g) \subseteq f\ `\ \boldsymbol{\alpha}(P)$›
⟨*proof*⟩

lemma *strict-events-of-inj-on-Renaming* :
‹$\boldsymbol{\alpha}(\mathit{Renaming}\ P\ f\ g) = f\ `\ \boldsymbol{\alpha}(P)$› **if** ‹$\mathit{inj\text{-}on}\ f\ \alpha(P)$›
⟨*proof*⟩

lemma *strict-events-of-Seq-subseteq* :

‹$\boldsymbol{\alpha}(P \mathbin{;} Q) \subseteq \boldsymbol{\alpha}(P) \cup (\mathit{if}\ \checkmark\mathbf{s}(P) = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ \boldsymbol{\alpha}(Q))$›
⟨*proof*⟩

lemma *strict-events-of-Sync-subset* : ‹$\boldsymbol{\alpha}(P \ [\![S]\!]\ Q) \subseteq \boldsymbol{\alpha}(P) \cup \boldsymbol{\alpha}(Q)$›
⟨*proof*⟩

13.3 Ticks of a Process

lemma *ticks-of-BOT* [*simp*] : ‹$\checkmark s(\bot) = \mathit{UNIV}$›
and *ticks-of-SKIP* [*simp*] : ‹$\checkmark s(\mathit{SKIP}\ r) = \{r\}$›
and *ticks-of-STOP* [*simp*] : ‹$\checkmark s(\mathit{STOP}) = \{\}$›
⟨*proof*⟩

lemma *anti-mono-ticks-of-T*: ‹$P \sqsubseteq_T Q \Longrightarrow \checkmark s(Q) \subseteq \checkmark s(P)$›
⟨*proof*⟩

lemma *anti-mono-ticks-of-F*: ‹$P \sqsubseteq_F Q \Longrightarrow \checkmark s(Q) \subseteq \checkmark s(P)$›
⟨*proof*⟩

lemma *anti-mono-ticks-of-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow \checkmark s(Q) \subseteq \checkmark s(P)$›
⟨*proof*⟩

lemma *anti-mono-ticks-of-DT*: ‹$P \sqsubseteq_{DT} Q \Longrightarrow \checkmark s(Q) \subseteq \checkmark s(P)$›
⟨*proof*⟩

lemma *anti-mono-ticks-of* : ‹$P \sqsubseteq Q \Longrightarrow \checkmark s(Q) \subseteq \checkmark s(P)$›
⟨*proof*⟩

lemma *ticks-of-GlobalNdet*: ‹$\checkmark s(\sqcap a \in A.\ P\ a) = (\bigcup a{\in}A.\ \checkmark s(P\ a))$›
⟨*proof*⟩

lemma *ticks-of-Mprefix*: ‹$\checkmark s(\Box a{\in}A \rightarrow P\ a) = (\bigcup a{\in}A.\ \checkmark s(P\ a))$›
⟨*proof*⟩

lemma *ticks-of-write0* : ‹$\checkmark s(a \rightarrow P) = \checkmark s(P)$›
⟨*proof*⟩

lemma *ticks-of-Mndetprefix*: ‹$\checkmark s(\sqcap a{\in}A \rightarrow P\ a) = (\bigcup a{\in}A.\ \checkmark s(P\ a))$›
⟨*proof*⟩

lemma *ticks-of-read* :
‹$\checkmark s(c\mathbf{?}a{\in}A \rightarrow P\ a) = (\bigcup a{\in}c\ `\ A.\ \checkmark s(P\ (\mathit{inv\text{-}into}\ A\ c\ a)))$›
⟨*proof*⟩

lemma *ticks-of-inj-on-read* :
‹*inj-on c A* ⟹ ✓*s*(*c*?*a*∈*A* → *P a*) = (⋃ *a*∈*A*. ✓*s*(*P a*))›
⟨*proof*⟩

lemma *ticks-of-ndet-write* :
‹✓*s*(*c*!!*a*∈*A* → *P a*) = (⋃ *a*∈*c* ‘ *A*. ✓*s*(*P* (*inv-into A c a*)))›
⟨*proof*⟩

lemma *ticks-of-inj-on-ndet-write* :
‹*inj-on c A* ⟹ ✓*s*(*c*!!*a*∈*A* → *P a*) = (⋃ *a*∈*A*. ✓*s*(*P a*))›
⟨*proof*⟩

lemma *ticks-of-write* : ‹✓*s*(*c*!*a* → *P*) = ✓*s*(*P*)›
⟨*proof*⟩

lemma *ticks-of-Ndet*: ‹✓*s*(*P* ⊓ *Q*) = ✓*s*(*P*) ∪ ✓*s*(*Q*)›
⟨*proof*⟩

lemma *ticks-of-Det*: ‹✓*s*(*P* □ *Q*) = ✓*s*(*P*) ∪ ✓*s*(*Q*)›
⟨*proof*⟩

lemma *ticks-of-Sliding*: ‹✓*s*(*P* ▷ *Q*) = ✓*s*(*P*) ∪ ✓*s*(*Q*)›
⟨*proof*⟩

lemma *ticks-of-Renaming*:
‹✓*s*(*Renaming P f g*) = (*if* 𝒟 *P* = {} *then g* ‘ ✓*s*(*P*) *else UNIV*)›
⟨*proof*⟩

lemma *ticks-of-Seq* :
‹✓*s*(*P* ; *Q*) = (*if* 𝒟 *P* = {} *then if* ✓*s*(*P*) = {} *then* {} *else* ✓*s*(*Q*) *else UNIV*)›
(**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *ticks-of-Sync-subset* : ‹✓*s*(*P* ⟦*S*⟧ *Q*) ⊆ ✓*s*(*P*) ∪ ✓*s*(*Q*)›
⟨*proof*⟩

lemma *ticks-of-no-div-Sync-subset* :
‹✓*s*(*P* ⟦*S*⟧ *Q*) ⊆ ✓*s*(*P*) ∩ ✓*s*(*Q*)› **if** ‹𝒟 (*P* ⟦*S*⟧ *Q*) = {}›
⟨*proof*⟩

lemma *ticks-of-Par-div* :
‹𝒟 *P* ∩ 𝒯 *Q* ∪ 𝒟 *Q* ∩ 𝒯 *P* ≠ {} ⟹ ✓*s*(*P* || *Q*) = *UNIV*›
and *ticks-of-no-div-Par-subset* :

‹$\mathcal{D}\ P \cap \mathcal{T}\ Q \cup \mathcal{D}\ Q \cap \mathcal{T}\ P = \{\} \Longrightarrow \checkmark s(P \mid\mid Q) \subseteq \checkmark s(P) \cap \checkmark s(Q)$›
⟨*proof*⟩

lemma *ticks-of-Hiding*:
‹$\checkmark s(P \setminus B) = (\mathit{if}\ \mathcal{D}\ (P \setminus B) = \{\}\ \mathit{then}\ \checkmark s(P)\ \mathit{else}\ \mathit{UNIV})$›
⟨*proof*⟩

lemma *tickFree-traces-iff-empty-ticks-of* : ‹$(\forall\, t \in \mathcal{T}\ P.\ tF\ t) \longleftrightarrow \checkmark s(P) = \{\}$›
⟨*proof*⟩

13.3.1 Strict Events of a Process

lemma *strict-ticks-of-BOT* [*simp*] : ‹$\checkmark\mathbf{s}(\bot) = \{\}$›
and *strict-ticks-of-SKIP* [*simp*] : ‹$\checkmark\mathbf{s}(\mathit{SKIP}\ r) = \{r\}$›
and *strict-ticks-of-STOP* [*simp*] : ‹$\checkmark\mathbf{s}(\mathit{STOP}) = \{\}$›
⟨*proof*⟩

lemma *strict-ticks-of-GlobalNdet-subset* : ‹$\checkmark\mathbf{s}(\sqcap a \in A.\ P\ a) \subseteq (\bigcup a{\in}A.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-Mprefix*:
‹$\checkmark\mathbf{s}(\Box a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-Mndetprefix*:
‹$\checkmark\mathbf{s}(\sqcap a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a{\in}A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-write0* : ‹$\checkmark\mathbf{s}(a \rightarrow P) = (\mathit{if}\ P = \bot\ \mathit{then}\ \{\}\ \mathit{else}\ \checkmark\mathbf{s}(P))$›
⟨*proof*⟩

lemma *strict-ticks-of-read* :
‹$\checkmark\mathbf{s}(c?a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a)) \neq \bot\}.\ \checkmark\mathbf{s}(P\ (\mathit{inv\text{-}into}\ A\ c\ (c\ a))))$›
⟨*proof*⟩

lemma *strict-ticks-of-inj-on-read* :
‹$\mathit{inj\text{-}on}\ c\ A \Longrightarrow \checkmark\mathbf{s}(c?a{\in}A \rightarrow P\ a) = (\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}.\ \checkmark\mathbf{s}(P\ a))$›
⟨*proof*⟩

lemma *strict-ticks-of-ndet-write* :

‹✓**s**(c**!!**$a{\in}A \to P\ a$) = ($\bigcup a{\in}\{a \in A.\ P\ (\textit{inv-into}\ A\ c\ (c\ a)) \neq \bot\}$. ✓**s**($P$ (*inv-into* A c (c a))))›
⟨*proof*⟩

lemma *strict-ticks-of-inj-on-ndet-write* :
‹*inj-on* c $A \Longrightarrow$ ✓**s**(c**!!**$a{\in}A \to P\ a$) = ($\bigcup a{\in}\{a \in A.\ P\ a \neq \bot\}$. ✓**s**($P$ a))›
⟨*proof*⟩

lemma *strict-ticks-of-write* : ‹✓**s**(c**!**$a \to P$) = (*if* $P = \bot$ *then* {} *else* ✓**s**(P))›
⟨*proof*⟩

lemma *strict-ticks-of-Ndet-subset* : ‹✓**s**($P \sqcap Q$) ⊆ ✓**s**(P) ∪ ✓**s**(Q)›
⟨*proof*⟩

lemma *strict-ticks-of-Det-subset* : ‹✓**s**($P \ \Box\ Q$) ⊆ ✓**s**(P) ∪ ✓**s**(Q)›
⟨*proof*⟩

lemma *strict-ticks-of-Sliding-subset* : ‹✓**s**($P \rhd Q$) ⊆ ✓**s**(P) ∪ ✓**s**(Q)›
⟨*proof*⟩

lemma *strict-ticks-of-Renaming-subset* : ‹✓**s**(*Renaming* P f g) ⊆ g ‘ ✓**s**(P)›
⟨*proof*⟩

lemma *strict-ticks-of-inj-on-Renaming* :
‹✓**s**(*Renaming* P f g) = g ‘ ✓**s**(P)› **if** ‹*inj-on* f $\alpha(P)$›
⟨*proof*⟩

lemma *strict-ticks-of-Seq-subset* : ‹✓**s**(P **;** Q) ⊆ (*if* ✓**s**(P) = {} *then* {} *else* ✓**s**(Q))›
⟨*proof*⟩

lemma *strict-ticks-of-Sync-subset* : ‹✓**s**(P ⟦S⟧ Q) ⊆ ✓**s**(P) ∩ ✓**s**(Q)›
⟨*proof*⟩

Chapter 14

CSP Assertions

14.1 Reference Processes

definition *DF* :: ‹'*a set* ⇒ ('*a*, '*r*) *process*$_{ptick}$›
where ‹*DF A* ≡ μ *X*. □*a*∈*A* → *X*›

lemma *DF-unfold* : ‹*DF A* = □*a*∈*A* → *DF A*›
⟨*proof*⟩

definition *DF*$_{SKIPS}$ :: ‹'*a set* ⇒ '*r set* ⇒ ('*a*,'*r*) *process*$_{ptick}$›
where ‹*DF*$_{SKIPS}$ *A R* ≡ μ *X*. (□*a* ∈ *A* → *X*) □ *SKIPS R*›

lemma *DF*$_{SKIPS}$*-unfold* : ‹*DF*$_{SKIPS}$ *A R* = (□*a* ∈ *A* → *DF*$_{SKIPS}$ *A R*) □ *SKIPS R*›
⟨*proof*⟩

definition *RUN* :: ‹'*a set* ⇒ ('*a*,'*r*) *process*$_{ptick}$›
where ‹*RUN A* ≡ μ *X*. □*x*∈*A* → *X*›

lemma *RUN-unfold* : ‹*RUN A* = □*a*∈*A* → *RUN A*›
⟨*proof*⟩

definition *CHAOS* :: ‹'*a set* ⇒ ('*a*,'*r*) *process*$_{ptick}$›
where ‹*CHAOS A* ≡ μ *X*. *STOP* □ (□*a* ∈ *A* → *X*)›

lemma *CHAOS-unfold* : ‹*CHAOS A* = *STOP* □ (□*a*∈*A* → *CHAOS A*)›
⟨*proof*⟩

definition *CHAOS*$_{SKIPS}$ :: ‹'*a set* ⇒ '*r set* ⇒ ('*a*,'*r*) *process*$_{ptick}$›
where ‹*CHAOS*$_{SKIPS}$ *A R* ≡ μ *X*. *SKIPS R* □ *STOP* □ (□*a*∈*A* → *X*)›

lemma *CHAOS$_{SKIPS}$-unfold*: ‹*CHAOS$_{SKIPS}$ A R = SKIPS R ⊓ STOP ⊓ (□a ∈ A → CHAOS$_{SKIPS}$ A R)*›
⟨*proof*⟩

14.2 Assertions

definition *deadlock-free* :: ‹('a,'r) *process$_{ptick}$* ⇒ *bool*›
where ‹*deadlock-free P ≡ DF UNIV ⊑$_{FD}$ P*›

definition *deadlock-free$_{SKIPS}$* :: ‹('a, 'r) *process$_{ptick}$* ⇒ *bool*›
where ‹*deadlock-free$_{SKIPS}$ P ≡ DF$_{SKIPS}$ UNIV UNIV ⊑$_{F}$ P*›

definition *lifelock-free* :: ‹('a,'r) *process$_{ptick}$* ⇒ *bool*›
where ‹*lifelock-free P ≡ CHAOS UNIV ⊑$_{FD}$ P*›

definition *lifelock-free$_{SKIPS}$* :: ‹('a,'r) *process$_{ptick}$* ⇒ *bool*›
where ‹*lifelock-free$_{SKIPS}$ P ≡ CHAOS$_{SKIPS}$ UNIV UNIV ⊑$_{FD}$ P*›

definition *non-terminating* :: ‹('a,'r) *process$_{ptick}$* ⇒ *bool*›
where ‹*non-terminating P ≡ RUN UNIV ⊑$_{T}$ P*›

14.3 Properties

lemma *DF$_{SKIPS}$-FD-DF* : ‹*DF$_{SKIPS}$ A R ⊑$_{FD}$ DF A*›
⟨*proof*⟩

lemma *SKIPS-FD-SKIPS-iff* :
‹*SKIPS S ⊑$_{FD}$ SKIPS R ⟷ (if R = {} then S = {} else R ⊆ S)*›
⟨*proof*⟩

lemma *SKIPS-F-SKIPS-iff* :
‹*SKIPS S ⊑$_{F}$ SKIPS R ⟷ (if R = {} then S = {} else R ⊆ S)*›
⟨*proof*⟩

lemma *SKIPS-T-SKIPS-iff* : ‹*SKIPS S ⊑$_{T}$ SKIPS R ⟷ (R ⊆ S)*›
⟨*proof*⟩

lemma *DF-subset* : ‹*DF B ⊑$_{FD}$ DF A*› **if** ‹*A ≠ {}*› ‹*A ⊆ B*› **for** *A B* :: ‹*'a set*›
⟨*proof*⟩

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

lemma *deadlock-free-Ndet-iff* :
‹*deadlock-free* $(P \sqcap Q) \longleftrightarrow$ *deadlock-free* $P \wedge$ *deadlock-free* Q›
⟨*proof*⟩

lemma DF_{SKIPS}*-subset* : ‹$DF_{SKIPS}\ B\ S \sqsubseteq_{FD} DF_{SKIPS}\ A\ R$›
if ‹$A \neq \{\}$› ‹$A \subseteq B$› ‹$R \neq \{\}$› ‹$R \subseteq S$›
⟨*proof*⟩

lemma DF_{SKIPS}*-Univ-freeness* : ‹$[\![A \neq \{\}; R \neq \{\}; DF_{SKIPS}\ A\ R \sqsubseteq_{FD} P]\!] \Longrightarrow$ *deadlock-free*$_{SKIPS}$ P›
⟨*proof*⟩

lemma *deadlock-free*$_{SKIPS}$*-Ndet-iff* :
‹*deadlock-free*$_{SKIPS}$ $(P \sqcap Q) \longleftrightarrow$ *deadlock-free*$_{SKIPS}$ $P \wedge$ *deadlock-free*$_{SKIPS}$ Q›
⟨*proof*⟩

lemma *div-free-DF*$_{SKIPS}$: ‹$\mathcal{D}\ (DF_{SKIPS}\ A\ R) = \{\}$›
⟨*proof*⟩

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

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

14.4 Events and Ticks of Reference Processes

lemma *events-of-SKIPS* : ‹$\alpha(SKIPS\ R) = \{\}$›
and *ticks-of-SKIPS* : ‹$\checkmark s(SKIPS\ R) = R$›
⟨*proof*⟩

lemma *no-ticks-imp-tickFree-T* : ‹$\checkmark s(P) = \{\} \Longrightarrow s \in \mathcal{T}\ P \Longrightarrow tF\ s$›
⟨*proof*⟩

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

lemma *ticks-DF* : ‹$\checkmark s(DF\ A) = \{\}$›

⟨*proof*⟩

lemma *events-of-DF*$_{SKIPS}$: ‹$\alpha(DF_{SKIPS}\ A\ R) = A$›
⟨*proof*⟩

lemma *ticks-DF*$_{SKIPS}$: ‹$\checkmark s(DF_{SKIPS}\ A\ R) = R$›
⟨*proof*⟩

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

lemma *ticks-RUN* : ‹$\checkmark s(RUN\ A) = \{\}$›
⟨*proof*⟩

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

lemma *ticks-CHAOS* : ‹$\checkmark s(CHAOS\ A) = \{\}$›
⟨*proof*⟩

lemma *events-of-CHAOS*$_{SKIPS}$: ‹$\alpha(CHAOS_{SKIPS}\ A\ R) = A$›
⟨*proof*⟩

lemma *ticks-CHAOS*$_{SKIPS}$: ‹$\checkmark s(CHAOS_{SKIPS}\ A\ R) = R$›
⟨*proof*⟩

lemma *RUN-subset-DT*: ‹$RUN\ B \sqsubseteq_{DT} RUN\ A$›
if ‹$A \subseteq B$› **for** $A\ B$ :: ‹$'a\ set$›
⟨*proof*⟩

lemma *CHAOS-subset-FD* : ‹$CHAOS\ B \sqsubseteq_{FD} CHAOS\ A$›
if ‹$A \subseteq B$› **for** $A\ B$ :: ‹$'a\ set$›
⟨*proof*⟩

lemma *CHAOS*$_{SKIPS}$*-subset-FD* : ‹$CHAOS_{SKIPS}\ B\ S \sqsubseteq_{FD} CHAOS_{SKIPS}\ A\ R$›
if ‹$A \subseteq B$› ‹$R \neq \{\}$› ‹$R \subseteq S$›
⟨*proof*⟩

14.5 Relations between refinements on reference processes

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

lemma *CHAOS$_{SKIPS}$-superset-events-of-ticks-of-leF* :
‹$CHAOS_{SKIPS}\ A\ R \sqsubseteq_F P$› **if** ‹$\alpha(P) \subseteq A$› **and** ‹$\checkmark s(P) \subseteq R$›
⟨*proof*⟩

corollary *CHAOS$_{SKIPS}$-events-of-ticks-of-leF*: ‹$CHAOS_{SKIPS}\ \alpha(P)\ \checkmark s(P) \sqsubseteq_F P$›
and *CHAOS$_{SKIPS}$-UNIV-UNIV-leF*: ‹$CHAOS_{SKIPS}\ UNIV\ UNIV \sqsubseteq_F P$›
⟨*proof*⟩

lemma *DF$_{SKIPS}$-F-DF* : ‹$DF_{SKIPS}\ A\ R \sqsubseteq_F DF\ A$›
⟨*proof*⟩

lemma *DF-F-RUN* : ‹$DF\ A \sqsubseteq_F RUN\ A$› **for** A :: ‹$'a\ set$›
⟨*proof*⟩

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

corollary *CHAOS$_{SKIPS}$-F-CHAOS* : ‹$CHAOS_{SKIPS}\ A\ R \sqsubseteq_F CHAOS\ A$›
and *CHAOS$_{SKIPS}$-F-DF$_{SKIPS}$* : ‹$CHAOS_{SKIPS}\ A\ R \sqsubseteq_F DF_{SKIPS}\ A\ R$›
⟨*proof*⟩

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

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

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

corollary *DF-FD-RUN* : ‹*DF A* $\sqsubseteq_{FD}$ *RUN A*›
and *CHAOS-FD-DF* : ‹*CHAOS A* $\sqsubseteq_{FD}$ *DF A*›
and $CHAOS_{SKIPS}$*-FD-CHAOS* : ‹$CHAOS_{SKIPS}$ *A R* $\sqsubseteq_{FD}$ *CHAOS A*›
and $CHAOS_{SKIPS}$*-FD-*DF_{SKIPS} : ‹$CHAOS_{SKIPS}$ *A R* $\sqsubseteq_{FD}$ DF_{SKIPS} *A R*›
⟨*proof*⟩

lemma *traces-CHAOS-subset* : ‹$\mathcal{T}$ (*CHAOS A*) ⊆ {*s*. *set s* ⊆ *ev* ‘ *A*}›
⟨*proof*⟩

lemma *traces-RUN-superset* : ‹{*s*. *set s* ⊆ *ev* ‘ *A*} ⊆ $\mathcal{T}$ (*RUN A*)›
⟨*proof*⟩

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

corollary *RUN-all-tickfree-traces2* : ‹*s* ∈ $\mathcal{T}$ (*RUN UNIV*)›
and *DF-all-tickfree-traces2* : ‹*s* ∈ $\mathcal{T}$ (*DF UNIV*)›
and *CHAOS-all-tickfree-trace2* : ‹*s* ∈ $\mathcal{T}$ (*CHAOS UNIV*)› **if** ‹*tF s*›
⟨*proof*⟩

lemma $traces\text{-}CHAOS_{SKIPS}$*-subset* :
‹$\mathcal{T}$ ($CHAOS_{SKIPS}$ *A R*) ⊆ {*s*. *ftF s* ∧ *set s* ⊆ *ev* ‘ *A* ∪ *tick* ‘ *R*}›
⟨*proof*⟩

lemma $traces\text{-}DF_{SKIPS}$*-superset* :
‹{*s*. *ftF s* ∧ *set s* ⊆ *ev* ‘ *A* ∪ *tick* ‘ *R*} ⊆ $\mathcal{T}$ (DF_{SKIPS} *A R*)›
⟨*proof*⟩

corollary DF_{SKIPS}*-all-front-tickfree-traces1*:
‹$\mathcal{T}$ (DF_{SKIPS} *A R*) = {*s*. *ftF s* ∧ *set s* ⊆ *ev* ‘ *A* ∪ *tick* ‘ *R*}›
and $CHAOS_{SKIPS}$*-all-front-tickfree-traces1*:
‹$\mathcal{T}$ ($CHAOS_{SKIPS}$ *A R*) = {*s*. *ftF s* ∧ *set s* ⊆ *ev* ‘ *A* ∪ *tick* ‘ *R*}›
⟨*proof*⟩

corollary DF_{SKIPS}*-all-front-tickfree-traces2* : ‹*s* ∈ $\mathcal{T}$ (DF_{SKIPS} *UNIV UNIV*)›
and $CHAOS_{SKIPS}$*-all-front-tickfree-traces2*: ‹*s* ∈ $\mathcal{T}$ ($CHAOS_{SKIPS}$ *UNIV UNIV*)›
if ‹*ftF s*›
⟨*proof*⟩

corollary *DF$_{SKIPS}$-UNIV-UNIV-leT* : ‹*DF$_{SKIPS}$ UNIV UNIV* $\sqsubseteq_T$ *P*›
and *CHAOS$_{SKIPS}$-UNIV-UNIV-leT* : ‹*CHAOS$_{SKIPS}$ UNIV UNIV* $\sqsubseteq_T$ *P*›
⟨*proof*⟩

lemma *deadlock-free-implies-lifelock-free*: ‹*deadlock-free P* $\Longrightarrow$ *lifelock-free P*›
⟨*proof*⟩

lemma *deadlock-free-implies-non-terminating*:
‹*tF s*› **if** ‹*deadlock-free P*› ‹*s* $\in \mathcal{T}$ *P*›
⟨*proof*⟩

lemma *deadlock-free$_{SKIPS}$-is-right*:
‹*deadlock-free$_{SKIPS}$* (*P* :: ('*a*, '*r*) *process$_{ptick}$*) $\longleftrightarrow$
($\forall$ *s*$\in\mathcal{T}$ *P*. *tF s* $\longrightarrow$ (*s*, *UNIV* :: ('*a*, '*r*) *event$_{ptick}$ set*) $\notin \mathcal{F}$ *P*)›
⟨*proof*⟩

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

corollary *deadlock-free$_{SKIPS}$-FD*: ‹*deadlock-free$_{SKIPS}$ P* $\longleftrightarrow$ *DF$_{SKIPS}$ UNIV UNIV* $\sqsubseteq_{FD}$ *P*›
⟨*proof*⟩

lemma *all-events-ticks-refusal*:
‹(*s*, *tick* ‘ ✓*s*(*P*) $\cup$ *ev* ‘ α(*P*)) $\in \mathcal{F}$ *P* $\Longrightarrow$ (*s*, *UNIV*) $\in \mathcal{F}$ *P*›
⟨*proof*⟩

corollary *deadlock-free$_{SKIPS}$-is-right-wrt-events*:
‹*deadlock-free$_{SKIPS}$ P* $\longleftrightarrow$
($\forall$ *s*$\in\mathcal{T}$ *P*. *tF s* $\longrightarrow$ (*s*, *tick* ‘ ✓*s*(*P*) $\cup$ *ev* ‘ α(*P*)) $\notin \mathcal{F}$ *P*)›
⟨*proof*⟩

lemma *deadlock-free-imp-deadlock-free$_{SKIPS}$*: ‹*deadlock-free P* $\Longrightarrow$ *deadlock-free$_{SKIPS}$ P*›
⟨*proof*⟩

14.6 *deadlock-free* and *deadlock-free$_{SKIPS}$* with *SKIP* and *STOP*

lemma *non-deadlock-free$_{SKIPS}$-STOP*: ‹¬ *deadlock-free$_{SKIPS}$ STOP*›
⟨*proof*⟩

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

lemma *deadlock-free$_{SKIPS}$-SKIPS* : ‹*deadlock-free$_{SKIPS}$* (*SKIPS R*) ⟷ *R* ≠ {}›
⟨*proof*⟩

lemma *deadlock-free$_{SKIPS}$-SKIP*: ‹*deadlock-free$_{SKIPS}$* (*SKIP r*)›
⟨*proof*⟩

lemma *non-deadlock-free-SKIPS* : ‹¬ *deadlock-free* (*SKIPS R*)›
⟨*proof*⟩

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

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* ⟷ (∀ *s*∈$\mathcal{T}$ *P*. *tF 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* ⊑$_{FD}$ *P*›
⟨*proof*⟩

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

lemma *divergence-refine-div-free* :
‹*CHAOS$_{SKIPS}$ UNIV UNIV* ⊑$_D$ *P* ⟷ $\mathcal{D}$ *P* = {}›

‹*CHAOS UNIV* $\sqsubseteq_D$ *P* $\longleftrightarrow \mathcal{D}$ *P* = {}›
‹*RUN UNIV* $\sqsubseteq_D$ *P* $\longleftrightarrow \mathcal{D}$ *P* = {}›
‹*DF*$_{SKIPS}$ *UNIV UNIV* $\sqsubseteq_D$ *P* $\longleftrightarrow \mathcal{D}$ *P* = {}›
‹*DF UNIV* $\sqsubseteq_D$ *P* $\longleftrightarrow \mathcal{D}$ *P* = {}›
⟨*proof*⟩

lemma *lifelock-free*$_{SKIPS}$*-iff-div-free* : ‹*lifelock-free*$_{SKIPS}$ *P* $\longleftrightarrow \mathcal{D}$ *P* = {}›
⟨*proof*⟩

lemma *lifelock-free-imp-lifelock-free*$_{SKIPS}$: ‹*lifelock-free P* $\Longrightarrow$ *lifelock-free*$_{SKIPS}$ *P*›
⟨*proof*⟩

corollary *deadlock-free*$_{SKIPS}$*-imp-lifelock-free*$_{SKIPS}$: ‹*deadlock-free*$_{SKIPS}$ *P* $\Longrightarrow$ *lifelock-free*$_{SKIPS}$ *P*›
⟨*proof*⟩

lemma *non-terminating-Seq* : ‹*P* **;** *Q* = *P*› **if** ‹*non-terminating P*›
⟨*proof*⟩

lemma *non-terminating-Sync* :
‹*non-terminating P* $\Longrightarrow$ *lifelock-free*$_{SKIPS}$ *Q* $\Longrightarrow$ *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* = ‹{}›]

Chapter 15

Advanced Induction Schemata

15.1 k-fixpoint-induction

lemma *nat-k-induct* [*case-names base step*]:
‹*P n*› **if** ‹∀ *i*<*k*. *P i*› **and** ‹∀ n_0. (∀ *i*<*k*. *P* (n_0 + *i*)) ⟶ *P* (n_0 + *k*)› **for** *k n* :: *nat*
⟨*proof*⟩

thm *fix-ind fix-ind2*

lemma *fix-ind-k* [*case-names admissibility base-k-steps step*]:
assumes *adm* : ‹*adm P*›
and *base-k-steps* : ‹∀ *i*<*k*. *P* (*iterate i*·*f*·⊥)›
and *step* : ‹⋀*X*. (∀ *i*<*k*. *P* (*iterate i*·*f*·*X*)) ⟹ *P* (*iterate k*·*f*·*X*)›
shows ‹*P* (*fix*·*f*)›
⟨*proof*⟩

lemma *nat-k-skip-induct* [*case-names lower-bound base-k step*]:
‹*P n*› **if** ‹1 ≤ *k*› **and** ‹∀ *i*<*k*. *P i*› **and** ‹∀ n_0. *P* n_0 ⟶ *P* (n_0 + *k*)› **for** *k n* :: *nat*
⟨*proof*⟩

lemma *fix-ind-k-skip* [*case-names lower-bound admissibility base-k-steps step*]:
assumes *k-1* : ‹1 ≤ *k*›
and *adm* : ‹*adm P*›
and *base-k-steps* : ‹∀ *i*<*k*. *P* (*iterate i*·*f*·⊥)›
and *step* : ‹⋀*X*. *P X* ⟹ *P* (*iterate k*·*f*·*X*)›
shows ‹*P* (*fix*·*f*)›
⟨*proof*⟩

15.2 Parallel fixpoint-induction

lemma *parallel-fix-ind-inc*[*case-names admissibility base-fst base-snd step*]:
assumes *adm*: ‹*adm* (λ*X*. *P* (*fst X*) (*snd X*))›
and *base-fst*: ‹⋀*Y*. *P* ⊥ *Y*› **and** *base-snd*: ‹⋀*X*. *P X* ⊥›
and *step*: ‹⋀*X Y*. *P X Y* ⟹ *P* (*G*·*X*) *Y* ⟹ *P X* (*H*·*Y*) ⟹ *P* (*G*·*X*) (*H*·*Y*)›
shows ‹*P* (*fix*·*G*) (*fix*·*H*)›
⟨*proof*⟩

Chapter 16

The Main Entry Point

This is the theory `HOL-CSP` should be imported from.

Chapter 17

Conclusion

17.1 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.

17.2 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.

It is also worth noting that the restructuring of the theory, as well as the proofs (declarative Isar style), has substantially increased the possibility to parallelize the proof checking process and makes the entire theory more maintainable.

Finally, having the entire theory formalized makes extensions such as parametized ticks possible since the effect of changes of basic definitions can be traced consequently. This is an important aspect that extensions of this kind are not ad hoc and do not endanger global consistency.

17.3 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 $(\setminus)$ 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 Mprefix ?A ?P ⟦?S⟧ Mprefix ?B ?Q* = $(\Box a{\in}(?A - ?S) \to (?P\ a\ [\![?S]\!]\ \mathit{Mprefix}\ ?B\ ?Q)) \Box (\Box b{\in}(?B - ?S) \to (\mathit{Mprefix}\ ?A\ ?P\ [\![?S]\!]\ ?Q\ b)) \Box (\Box x{\in}(?A \cap ?B \cap ?S) \to (?P\ x\ [\![?S]\!]\ ?Q\ x))$ 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 $?P\ [\![?S]\!]\ (?Q\ [\![?S]\!]\ ?R) = ?P\ [\![?S]\!]\ ?Q\ [\![?S]\!]\ ?R$. (In HOL-CSP 1.0, this had only be shown for special cases like $?P \parallel (?Q \parallel ?R) = ?P \parallel ?Q \parallel ?R$).

6. the generalized non deterministic prefix choice operator — relevant for proofs of deadlock-freeness — has been added to the theory *Mndet-prefix* ≡ *map-fun id* (*map-fun id process-of-process*$_0$) ($\lambda A\ P$. *if* $A = \{\}$ *then process*$_0$*-of-process STOP else* $(\bigcup_{a\in A} \mathcal{F}\ (a \to P\ a), \bigcup_{a\in A} \mathcal{D}\ (a \to P\ a))$); it is proven monotone and continuous ($\bigwedge a.\ a \in ?A \Longrightarrow \mathit{cont}\ (?f\ a)) \Longrightarrow \mathit{cont}\ (\lambda b.\ \sqcap a{\in}?A \to ?f\ a\ b)$ in the general case (in contrast to the global choice without prefix, see $[\![\mathit{finite}\ ?A; \bigwedge a.\ a \in ?A \Longrightarrow \mathit{cont}\ (?P\ a)]\!] \Longrightarrow \mathit{cont}\ (\lambda y.\ \sqcap z{\in}?A.\ ?P\ z\ y)$). This is relevant for the definition of the deadlock reference processes $DF\ ?A \equiv \mu\ x.\ \sqcap a{\in}?A \to x$ and $DF_{SKIPS}\ ?A\ ?R \equiv \mu\ x.\ (\sqcap a{\in}?A \to x) \sqcap \mathit{SKIPS}\ ?R$.

7. since Isabelle-2025, new support for read *read c A P* and non deterministic write *ndet-write c A P* has been added. Also, sliding choice has been added and new algebraic laws involving this operator (see $?A \cap ?S \neq \{\} \Longrightarrow \mathit{Mprefix}\ ?A\ ?P \setminus ?S = (\Box a{\in}(?A - ?S) \to (?P\ a \setminus ?S)) \rhd (\sqcap a{\in}(?A \cap ?S).\ (?P\ a \setminus ?S))$ have been proven.

Chapter 18

Annex: Refinement Example with Buffer over infinite Alphabet

18.1 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*

18.2 The Standard Proof

18.2.1 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 *simplification-lemmas* [*simp*] :
‹*range left* ∩ *SYN* = {}›
‹*range right* ∩ *SYN* = {}›
‹*ack* ∈ *SYN*›
‹*range mid* ⊆ *SYN*›
‹*mid x* ∈ *SYN*›
‹*right x* ∉ *SYN*›
‹*left x* ∉ *SYN*›
‹*inj mid*›
⟨*proof*⟩

lemma *finite* (*SYN*:: *'a channel set*) ⟹ *finite* {(*t*::*'a*). *True*}
⟨*proof*⟩

18.2.2 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* → *right*!*x* → *COPY*
⟨*proof*⟩

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

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

18.2.3 Various Samples of Refinement Proofs

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

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

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

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

An example for a very explicit structured proof. Slow-motion for presentations. Note that the proof makes no assumption over the structure of the

content of the channel; it is truly polymorphic wrt. the channel type $'\alpha$.

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

An example for a highly automated proof.

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

lemma *impl-refines-spec* : $COPY \sqsubseteq_{FD} SYSTEM$
⟨*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*⟩

18.2.4 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*⟩

18.3 An Alternative Approach: Using the fixrec-Package

18.3.1 Channels and Synchronisation Sets

As before.

18.3.2 Process Definitions via fixrec-Package

fixrec
$COPY' :: 'a\ channel\ process$
and
$SEND' :: 'a\ channel\ process$

and
$REC' :: 'a\ channel\ process$
where
$COPY'\text{-}rec[simp\ del]:\ \ COPY' = left?x \rightarrow right!x \rightarrow COPY'$
$|\ \ SEND'\text{-}rec[simp\ del]:\ \ SEND' = left?x \rightarrow mid!x \rightarrow ack \rightarrow SEND'$
$|\ \ REC'\text{-}rec[simp\ del]:\ \ REC' = mid?x \rightarrow right!x \rightarrow ack \rightarrow REC'$

thm *COPY′-rec*
definition $SYSTEM' :: 'a\ channel\ process$
where $\ \ \langle SYSTEM' \equiv ((SEND' \llbracket SYN \rrbracket REC') \setminus SYN)\rangle$

18.3.3 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′′* : $(COPY'::'a\ channel\ process) \sqsubseteq_{FD} SYSTEM'$
$\langle proof\rangle$

lemma *spec-refines-impl′* :
assumes *fin*: *finite* $(SYN::'a\ channel\ set)$
shows $SYSTEM' \sqsubseteq_{FD} (COPY'::'a\ channel\ process)$
$\langle proof\rangle$

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

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.