

HOL-CSP Version 2.0

Benoit Ballenghien

Safouan Taha
Lina Ye

Burkhart Wolff

June 17, 2024

Contents

1	Context	7
1.1	Preface	7
1.2	Introduction	8
1.3	An Outline of Failure-Divergence Semantics	9
1.3.1	Non-Determinism	10
1.3.2	Infinite Chatter	10
1.3.3	The Global Architecture of HOL-CSP 2.0	11
2	The Notion of Processes	15
2.1	Pre-Requisite: Basic Traces and tick-Freeness	15
2.2	Basic Types, Traces, Failures and Divergences	18
2.3	The Process Type Invariant	19
2.4	The Abstraction to the process-Type	22
2.5	Some Consequences of the Process Characterization	25
2.6	Process Approximation is a Partial Ordering, a Cpo, and a Pcpo	27
2.7	Process Refinement is a Partial Ordering	31
2.8	Process Refinement is Admissible	33
2.9	The Conditional Statement is Continuous	33
3	The CSP Operators	35
3.1	The Undefined Process	35
3.2	The SKIP Process	36
3.3	The STOP Process	36
3.4	Deterministic Choice Operator Definition	37
3.4.1	The Det Operator Definition	37
3.4.2	The Projections	37
3.4.3	Basic Laws	38
3.4.4	The Continuity-Rule	38
3.5	Nondeterministic Choice Operator Definition	39
3.5.1	The Ndet Operator Definition	39
3.5.2	The Projections	39
3.5.3	Basic Laws	39

3.5.4	The Continuity Rule	39
3.6	The Sequence Operator	40
3.6.1	Definition	40
3.6.2	The Projections	40
3.6.3	Continuity Rule	41
3.7	The Renaming Operator	43
3.7.1	Some preliminaries	43
3.7.2	The Renaming Operator Definition	44
3.7.3	The Projections	44
3.7.4	Continuity Rule	45
3.7.5	Some nice properties	47
4	Concurrent CSP Operators	49
4.1	The Hiding Operator	49
4.1.1	Preliminaries : primitives and lemmas	49
4.1.2	The Hiding Operator Definition	50
4.1.3	Projections	50
4.1.4	Continuity Rule	50
4.2	The Synchronizing Operator	51
4.2.1	Basic Concepts	51
4.2.2	Consequences	52
4.2.3	The Sync Operator Definition	55
4.2.4	The Projections	55
4.2.5	Syntax for Interleave and Parallel Operator	56
4.2.6	Continuity Rule	56
4.3	The Multi-Prefix Operator Definition	58
4.3.1	The Definition and some Consequences	58
4.3.2	Projections in Prefix	59
4.3.3	Basic Properties	59
4.3.4	Proof of Continuity Rule	60
4.3.5	High-level Syntax for Read and Write	61
4.3.6	CSP _M -Style Syntax for Communication Primitives	62
4.4	Multiple non deterministic operator	63
4.4.1	Finite case Continuity	64
4.4.2	General case Continuity	64
5	Advanced Induction Schemata	65
5.1	k-fixpoint-induction	65
5.2	Parallel fixpoint-induction	66
6	Refinements	67
6.1	Idempotency	67
6.2	Some obvious refinements	68
6.3	Antisymmetry	68

6.4	Transitivity	68
6.5	Relations between refinements	68
6.6	More obvious refinements	68
6.7	Admissibility	69
7	The "Laws" of CSP	71
7.1	General Laws	71
7.2	Deterministic Choice Operator Laws	71
7.3	NonDeterministic Choice Operator Laws	72
7.3.1	Multi-Operators laws	72
7.4	Sequence Operator Laws	72
7.4.1	Preliminaries	72
7.4.2	Laws	73
7.4.3	Multi-Operators laws	73
7.5	The Multi-Prefix Operator Laws	74
7.5.1	Multi-Operators laws	74
7.5.2	Derivative Operators laws	74
7.6	The Hiding Operator Laws	75
7.6.1	Preliminaries	75
7.6.2	Laws	75
7.6.3	Multi-Operators laws	76
7.7	The Sync Operator Laws	76
7.7.1	Preliminaries	76
7.7.2	Laws	79
7.7.3	Multi-Operators laws	79
7.7.4	Derivative Operators laws	81
7.8	Multiple Non Deterministic Operator Laws	82
7.9	Infra-structure for Communication Primitives	82
7.10	Renaming operator laws	83
8	The refinements laws	87
8.1	Monotonicity	87
8.2	Monotonicity	88
8.3	CSP Assertions	93
8.4	Some deadlock freeness laws	93
8.5	Preliminaries	94
8.6	Deadlock Free	94
8.7	Run	94
8.8	Reference processes and their unfolding rules	94
8.9	Process events and reference processes events	94
8.10	Relations between refinements on reference processes	95
8.11	<i>deadlock-free</i> and <i>deadlock-free-v2</i> with <i>SKIP</i> and <i>STOP</i>	98
8.12	Non-terminating Runs	98
8.13	Lifelock Freeness	99

8.14	New laws	99
9	Conclusion	101
9.1	Related Work	101
9.2	Lessons learned	101
9.3	A Summary on New Results	102
10	Annex: Refinement Example with Buffer over infinite Al-	
	phabet	105
10.1	Defining the Copy-Buffer Example	105
10.2	The Standard Proof	106
10.2.1	Channels and Synchronization Sets	106
10.2.2	Definitions by Recursors	106
10.2.3	A Refinement Proof	106
10.2.4	Deadlock Freeness Proof	107
10.3	An Alternative Approach: Using the fixrec-Package	107
10.3.1	Channels and Synchronisation Sets	107
10.3.2	Process Definitions via fixrec-Package	107
10.3.3	Another Refinement Proof on fixrec-infrastructure	108

Chapter 1

Context

1.1 Preface

This is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe’s Book ”Theory and Practice of Concurrency” [12] and the semantic details in a joint Paper of Roscoe and Brooks ”An improved failures model for communicating processes” [2].

The original version of this formalization, called HOL-CSP 1.0 [13], revealed minor, but omnipresent foundational errors in key concepts like the process invariant. A correction was proposed slightly before the apparition of Roscoe’s book (all three authors were in e-mail contact at that time).

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

The effort is motivated by the following assets of CSP:

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

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

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

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

1.2 Introduction

DRAWN FROM THE PAPER [13]

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

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

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

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

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

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

1.3 An Outline of Failure-Divergence Semantics

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

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

one, neither to distinguish deadlock from infinite internal chatter,

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

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

1.3.1 Non-Determinism

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

$$(a \rightarrow Stop) \sqcap b \rightarrow Stop \tag{1}$$

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 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

$$(\mu x. a \rightarrow x) \setminus a \quad (3)$$

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

1.3.3 The Global Architecture of HOL-CSP 2.0

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

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

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

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

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

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

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

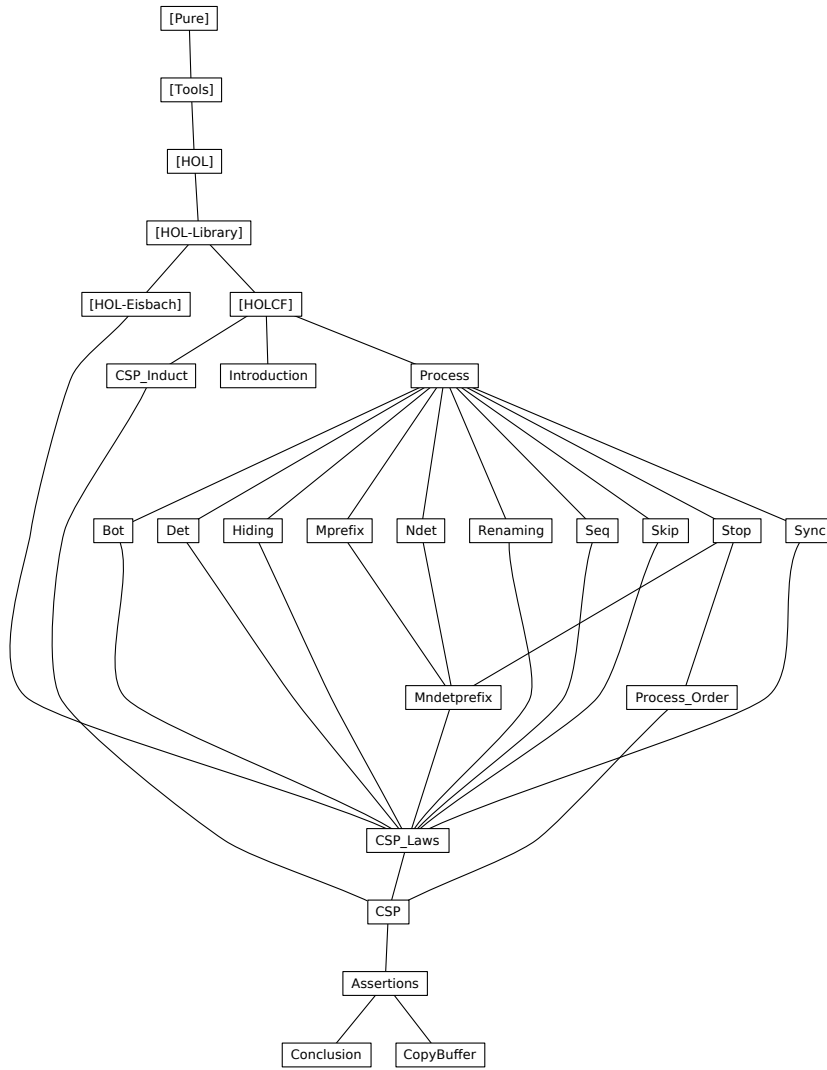


Figure 1.1: The HOL-CSP 2.0 Theory Graph

Chapter 2

The Notion of Processes

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

```
theory Process
  imports HOLCF
begin
```

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

```
default-sort type
```

2.1 Pre-Requisite: Basic Traces and tick-Freeness

The denotational semantics of CSP assumes a distinguishable special event, called `tick` and written \checkmark , that is required to occur only in the end of traces in order to signalize successful termination of a process. (In the original text of Hoare, this treatment was more liberal and lead to foundational problems: the process invariant could not be established for the sequential composition operator of CSP; see [13] for details.)

```
datatype 'a event = ev 'a | tick
```

```
type-synonym 'a trace = ('a event) list
```

We chose as standard ordering on traces the prefix ordering.

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

```
definition le-list-def : (s::'a list) ≤ t ↔ (∃ r. s @ r = t)
```

```
definition less-list-def: (s::'a list) < t ↔ s ≤ t ∧ s ≠ t
```

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

instance
 ⟨proof⟩

end

Some facts on the prefix ordering.

lemma $\text{nil-le}[simp]: [] \leq s$
 ⟨proof⟩

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

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

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

lemma $\text{less-self}[simp]: t < t@[a]$
 ⟨proof⟩

lemma $\text{le-length-mono}: s \leq t \implies \text{length } s \leq \text{length } t$
 ⟨proof⟩

lemma $\text{less-length-mono}: s < t \implies \text{length } s < \text{length } t$
 ⟨proof⟩

lemma $\text{less-cons}: s < t \implies a \# s < a \# t$
 ⟨proof⟩

lemma $\text{less-append}: s < t \implies a @ s < a @ t$
 ⟨proof⟩

lemma $\text{less-tail}: s \neq [] \implies s < t \implies \text{tl } s < \text{tl } t$
 ⟨proof⟩

lemma $\text{list-nonMt-append}: s \neq [] \implies \exists a t. s = t @ [a]$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

lemma *tickFree-tl* : *tickFree* *s* ⇒ *tickFree*(*tl* *s*)
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *nonTickFree-n-frontTickFree*: $\llbracket \neg \text{tickFree } s; \text{front-tickFree } s \rrbracket \implies \exists t. s = t @ [\text{tick}]$
 ⟨proof⟩

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

lemma *front-tickFree-append*: $\llbracket \text{tickFree } s; \text{front-tickFree } t \rrbracket \implies \text{front-tickFree } (s @ t)$
 ⟨proof⟩

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

lemma *tickFree-butlast*: $\langle \text{tickFree } s \longleftrightarrow \text{tickFree } (\text{butlast } s) \wedge (s \neq [] \longrightarrow \text{last } s \neq \text{tick}) \rangle$
 ⟨proof⟩

lemma *front-tickFree-butlast*: $\langle \text{front-tickFree } s \longleftrightarrow \text{tickFree } (\text{butlast } s) \rangle$
 ⟨proof⟩

2.2 Basic Types, Traces, Failures and Divergences

type-synonym $'\alpha \text{ refusal} = ('\alpha \text{ event}) \text{ set}$

type-synonym $'\alpha \text{ failure} = '\alpha \text{ trace} \times '\alpha \text{ refusal}$

type-synonym $'\alpha \text{ divergence} = '\alpha \text{ trace set}$

type-synonym $'\alpha \text{ process}_0 = '\alpha \text{ failure set} \times '\alpha \text{ divergence}$

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

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

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

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

2.3 The Process Type Invariant

definition *is-process* :: ' α process₀ \Rightarrow bool where

$$\begin{aligned}
\text{is-process } P = & \\
& (([],\{\}) \in \text{FAILURES } P \wedge \\
& (\forall s X. (s,X) \in \text{FAILURES } P \longrightarrow \text{front-tickFree } s) \wedge \\
& (\forall s t. (s@t,\{\}) \in \text{FAILURES } P \longrightarrow (s,\{\}) \in \text{FAILURES } P) \wedge \\
& (\forall s X Y. (s,Y) \in \text{FAILURES } P \ \& \ X \leq Y \longrightarrow (s,X) \in \text{FAILURES } P) \wedge \\
& (\forall s X Y. (s,X) \in \text{FAILURES } P \wedge \\
& (\forall c. \quad c \in Y \longrightarrow ((s@[c],\{\}) \notin \text{FAILURES } P)) \longrightarrow \\
& \quad (s,X \cup Y) \in \text{FAILURES } P) \wedge \\
& (\forall s X. (s@[tick],\{\}) : \text{FAILURES } P \longrightarrow (s,X - \{tick\}) \in \text{FAILURES } P) \wedge \\
& (\forall s t. s \in \text{DIVERGENCES } P \wedge \text{tickFree } s \wedge \text{front-tickFree } t \\
& \quad \longrightarrow s@t \in \text{DIVERGENCES } P) \wedge \\
& (\forall s X. s \in \text{DIVERGENCES } P \longrightarrow (s,X) \in \text{FAILURES } P) \wedge \\
& (\forall s. s @ [tick] : \text{DIVERGENCES } P \longrightarrow s \in \text{DIVERGENCES } P))
\end{aligned}$$

lemma *is-process-spec*:

$$\begin{aligned}
\text{is-process } P = & \\
& (([],\{\}) \in \text{FAILURES } P \wedge \\
& (\forall s X. (s,X) \in \text{FAILURES } P \longrightarrow \text{front-tickFree } s) \wedge \\
& (\forall s t. (s @ t,\{\}) \notin \text{FAILURES } P \vee (s,\{\}) \in \text{FAILURES } P) \wedge \\
& (\forall s X Y. (s,Y) \notin \text{FAILURES } P \vee \neg(X \subseteq Y) \mid (s,X) \in \text{FAILURES } P) \wedge \\
& (\forall s X Y. (s,X) \in \text{FAILURES } P \wedge \\
& (\forall c. c \in Y \longrightarrow ((s@[c],\{\}) \notin \text{FAILURES } P)) \longrightarrow (s,X \cup Y) \in \text{FAILURES } \\
& P) \wedge \\
& (\forall s X. (s@[tick],\{\}) \in \text{FAILURES } P \longrightarrow (s,X - \{tick\}) \in \text{FAILURES } P) \\
& \wedge \\
& (\forall s t. s \notin \text{DIVERGENCES } P \vee \neg \text{tickFree } s \vee \neg \text{front-tickFree } t \\
& \quad \vee s @ t \in \text{DIVERGENCES } P) \wedge \\
& (\forall s X. s \notin \text{DIVERGENCES } P \vee (s,X) \in \text{FAILURES } P) \wedge \\
& (\forall s. s @ [tick] \notin \text{DIVERGENCES } P \vee s \in \text{DIVERGENCES } P)) \\
\langle \text{proof} \rangle
\end{aligned}$$

lemma *Process-eqI* :

assumes *A*: $\text{FAILURES } P = \text{FAILURES } Q$

assumes *B*: $\text{DIVERGENCES } P = \text{DIVERGENCES } Q$

shows $(P::'\alpha \text{ process}_0) = Q$

$\langle \text{proof} \rangle$

lemma *process-eq-spec*:

$$\begin{aligned}
((P::'\alpha \text{ process}_0) = Q) = & (\text{FAILURES } P = \text{FAILURES } Q \wedge \text{DIVERGENCES } P \\
= \text{DIVERGENCES } Q) & \\
\langle \text{proof} \rangle &
\end{aligned}$$

lemma *process-surj-pair*: $(\text{FAILURES } P, \text{DIVERGENCES } P) = P$

$\langle \text{proof} \rangle$

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

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

lemma *is-process2*: $is-process P \implies \forall s X. (s, X) \in FAILURES P \longrightarrow front-tickFree_s$
 ⟨proof⟩

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

lemma *is-process3-S-pref*: $\llbracket is-process P; (t, \{\}) \in FAILURES P; s \leq t \rrbracket \implies (s, \{\}) \in FAILURES P$
 ⟨proof⟩

lemma *is-process4*: $is-process P \implies \forall s X Y. (s, Y) \notin FAILURES P \vee \neg X \subseteq Y \vee (s, X) \in FAILURES P$
 ⟨proof⟩

lemma *is-process4-S*: $\llbracket is-process P; (s, Y) \in FAILURES P; X \subseteq Y \rrbracket \implies (s, X) \in FAILURES P$
 ⟨proof⟩

lemma *is-process4-S1*: $\llbracket is-process P; x \in FAILURES P; X \subseteq snd x \rrbracket \implies (fst x, X) \in FAILURES P$
 ⟨proof⟩

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

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

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

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

$(s, X - \{tick\}) \in FAILURES P$
 $\langle proof \rangle$

lemma *is-process6-S*: $\llbracket is-process P ; (s@[tick], \{\}) \in FAILURES P \rrbracket \implies (s, X - \{tick\}) \in FAILURES P$
 $\langle proof \rangle$

lemma *is-process7*:
 $is-process P \implies \forall s t. s \notin DIVERGENCES P \vee \neg tickFree s \vee \neg front-tickFree t \vee s @ t \in DIVERGENCES P$
 $\langle proof \rangle$

lemma *is-process7-S*:
 $\llbracket is-process P ; s : DIVERGENCES P ; tickFree s ; front-tickFree t \rrbracket \implies s @ t \in DIVERGENCES P$
 $\langle proof \rangle$

lemma *is-process8*: $is-process P \implies \forall s X. s \notin DIVERGENCES P \vee (s, X) \in FAILURES P$
 $\langle proof \rangle$

lemma *is-process8-S*: $\llbracket is-process P ; s \in DIVERGENCES P \rrbracket \implies (s, X) \in FAILURES P$
 $\langle proof \rangle$

lemma *is-process9*: $is-process P \implies \forall s. s@[tick] \notin DIVERGENCES P \vee s \in DIVERGENCES P$
 $\langle proof \rangle$

lemma *is-process9-S*: $\llbracket is-process P ; s@[tick] \in DIVERGENCES P \rrbracket \implies s \in DIVERGENCES P$
 $\langle proof \rangle$

lemma *Failures-implies-Traces*: $\llbracket is-process P ; (s, X) \in FAILURES P \rrbracket \implies s \in TRACES P$
 $\langle proof \rangle$

lemma *is-process5-sing*:
 $\llbracket is-process P ; (s, \{x\}) \notin FAILURES P ; (s, \{\}) \in FAILURES P \rrbracket \implies (s @ [x], \{\}) \in FAILURES P$
 $\langle proof \rangle$

lemma *is-process5-singT*:
 $\llbracket is-process P ; (s, \{x\}) \notin FAILURES P ; (s, \{\}) \in FAILURES P \rrbracket \implies s @ [x] \in TRACES P$
 $\langle proof \rangle$

lemma *front-trace-is-tickfree*:

assumes A : *is-process* P **and** B : $(t @ [tick], X) \in FAILURES P$
shows $tickFree t$
 $\langle proof \rangle$

lemma *trace-with-Tick-implies-tickFree-front* : $\llbracket is-process P; t @ [tick] \in TRACES P \rrbracket \implies tickFree t$
 $\langle proof \rangle$

2.4 The Abstraction to the process-Type

typedef
 $'\alpha process = \{p :: '\alpha process_0 . is-process p\}$
 $\langle proof \rangle$

setup-lifting *type-definition-process*

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

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

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

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

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

lemma *is-process-Rep* : *is-process* (*Rep-process* P)
 $\langle proof \rangle$

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

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

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

<proof>

lemma *is-processT*:

$([], \{\}) \in \mathcal{F} P \wedge$
 $(\forall s X. (s, X) \in \mathcal{F} P \longrightarrow \text{front-tickFree } s) \wedge$
 $(\forall s t. (s @ t, \{\}) \in \mathcal{F} P \longrightarrow (s, \{\}) \in \mathcal{F} P) \wedge$
 $(\forall s X Y. (s, Y) \in \mathcal{F} P \wedge (X \subseteq Y) \longrightarrow (s, X) \in \mathcal{F} P) \wedge$
 $(\forall s X Y. (s, X) \in \mathcal{F} P \wedge (\forall c. c \in Y \longrightarrow ((s @ [c], \{\}) \notin \mathcal{F} P)) \longrightarrow (s, X \cup Y) \in \mathcal{F} P) \wedge$
 $(\forall s X. (s @ [\text{tick}], \{\}) \in \mathcal{F} P \longrightarrow (s, X - \{\text{tick}\}) \in \mathcal{F} P) \wedge$
 $(\forall s t. s \in \mathcal{D} P \wedge \text{tickFree } s \wedge \text{front-tickFree } t \longrightarrow s @ t \in \mathcal{D} P) \wedge$
 $(\forall s X. s \in \mathcal{D} P \longrightarrow (s, X) \in \mathcal{F} P) \wedge$
 $(\forall s. s @ [\text{tick}] \in \mathcal{D} P \longrightarrow s \in \mathcal{D} P)$
<proof>

lemma *process-charn*:

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

<proof>

split of `is_processT`:

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

<proof>

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

<proof>

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

<proof>

lemma *is-proT2*:

assumes $A : (s, X) \in \mathcal{F} P$ **and** $B : s \neq []$

shows $\text{tick} \notin \text{set } (tl (\text{rev } s))$

<proof>

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

<proof>

lemma *is-processT3-S-pref* :

$\llbracket (t, \{\}) \in \mathcal{F} P; s \leq t \rrbracket \implies (s, \{\}) \in \mathcal{F} P$
<proof>

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

<proof>

lemma *is-processT4-S1* : $\llbracket x \in \mathcal{F} P; X \subseteq \text{snd } x \rrbracket \implies (\text{fst } x, X) \in \mathcal{F} P$

<proof>

lemma *is-processT5*:

$\forall s X Y. (s, X) \in \mathcal{F} P \wedge (\forall c. c \in Y \longrightarrow (s @ [c], \{\}) \notin \mathcal{F} P) \longrightarrow (s, X \cup Y) \in \mathcal{F} P$

<proof>

lemma *is-processT5-S1*:

$(s, X) \in \mathcal{F} P \implies (s, X \cup Y) \notin \mathcal{F} P \implies \exists c. c \in Y \wedge (s @ [c], \{\}) \in \mathcal{F} P$

<proof>

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

<proof>

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

<proof>

lemma *is-processT5-S3*:

assumes *A*: $(s, \{\}) \in \mathcal{F} P$

and *B*: $(s @ [c], \{\}) \notin \mathcal{F} P$

shows $(s, \{c\}) \in \mathcal{F} P$

<proof>

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

<proof>

lemma *is-processT5-S5*:

$(s, X) \in \mathcal{F} P \implies \forall c. c \in Y \longrightarrow (s, X \cup \{c\}) \notin \mathcal{F} P \implies \forall c. c \in Y \longrightarrow (s @ [c], \{\}) \in \mathcal{F} P$

<proof>

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

<proof>

2.5. SOME CONSEQUENCES OF THE PROCESS CHARACTERIZATION 25

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

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

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

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

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

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

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

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

2.5 Some Consequences of the Process Characterization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

definition *min_elems* :: ('s::ord) set \Rightarrow 's set
 where $\text{min_elems } X = \{s \in X. \forall t. t \in X \longrightarrow \neg (t < s)\}$

lemma *Nil-min_elems* : $[] \in A \implies [] \in \text{min_elems } A$
 ⟨proof⟩

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

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

lemma *min_elems-Collect-ftF-is-Nil* : $\text{min_elems } (\text{Collect front-tickFree}) = \{[]\}$
 ⟨proof⟩

lemma *min-elems5* :
assumes $A: (x::'a \text{ list}) \in A$
shows $\exists s \leq x. s \in \text{min-elems } A$
<proof>

lemma *min-elems4*: $A \neq \{\}$ $\implies \exists s. (s :: 'a \text{ trace}) \in \text{min-elems } A$
<proof>

lemma *min-elems-charn*: $t \in A \implies \exists t' r. t = (t' @ r) \wedge t' \in \text{min-elems } A$
<proof>

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

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

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

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

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

instantiation
process :: (type) below
begin

declares approximation ordering \sqsubseteq also written \ll .

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

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*: $P \sqsubseteq Q \implies \mathcal{D} Q \subseteq \mathcal{D} P$
<proof>

2.6. PROCESS APPROXIMATION IS A PARTIAL ORDERING, A CPO, AND A PCPO29

lemma *le-approx2*: $\llbracket P \sqsubseteq Q; s \notin \mathcal{D} P \rrbracket \Longrightarrow (s, X) \in \mathcal{F} Q = ((s, X) \in \mathcal{F} P)$
<proof>

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

lemma *le-approx2T*: $\llbracket P \sqsubseteq Q; s \notin \mathcal{D} P \rrbracket \Longrightarrow s \in \mathcal{T} Q = (s \in \mathcal{T} P)$
<proof>

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

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

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

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

instance

process :: (type) po
<proof>

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

Some facts from the theory of complete partial orders:

- `Porder.chainE` : $\text{chain } ?Y \Longrightarrow ?Y ?i \sqsubseteq ?Y (\text{Suc } ?i)$
- `Porder.chain_mono` : $\llbracket \text{chain } ?Y; ?i \leq ?j \rrbracket \Longrightarrow ?Y ?i \sqsubseteq ?Y ?j$
- `Porder.is_ubD` : $\llbracket ?S <| ?u; ?x \in ?S \rrbracket \Longrightarrow ?x \sqsubseteq ?u$
- `Porder.ub_rangeI` :
 $(\bigwedge i. ?S i \sqsubseteq ?x) \Longrightarrow \text{range } ?S <| ?x$
- `Porder.ub_imageD` : $\llbracket ?f ' ?S <| ?u; ?x \in ?S \rrbracket \Longrightarrow ?f ?x \sqsubseteq ?u$
- `Porder.is_ub_upward` : $\llbracket ?S <| ?x; ?x \sqsubseteq ?y \rrbracket \Longrightarrow ?S <| ?y$
- `Porder.is_lubD1` : $?S <<| ?x \Longrightarrow ?S <| ?x$
- `Porder.is_lubI` : $\llbracket ?S <| ?x; \bigwedge u. ?S <| u \Longrightarrow ?x \sqsubseteq u \rrbracket \Longrightarrow ?S <<| ?x$
- `Porder.is_lub_maximal` : $\llbracket ?S <| ?x; ?x \in ?S \rrbracket \Longrightarrow ?S <<| ?x$

- $\text{Porder.is_lub_lub} : ?M \ll\mid ?x \implies ?M \ll\mid \text{lub } ?M$
- $\text{Porder.is_lub_range_shift}$:
 $\text{chain } ?S \implies \text{range } (\lambda i. ?S (i + ?j)) \ll\mid ?x = \text{range } ?S \ll\mid ?x$
- $\text{Porder.is_lub_rangeD1} : \text{range } ?S \ll\mid ?x \implies ?S ?i \sqsubseteq ?x$
- $\text{Porder.lub_eqI} : ?M \ll\mid ?l \implies \text{lub } ?M = ?l$
- $\text{Porder.is_lub_unique} : \llbracket ?S \ll\mid ?x; ?S \ll\mid ?y \rrbracket \implies ?x = ?y$

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

lemma $\text{min-elems3} : s @ [c] \in \mathcal{D} P \implies s @ [c] \notin \text{min-elems } (\mathcal{D} P) \implies s \in \mathcal{D} P$
 $\langle \text{proof} \rangle$

lemma $\text{min-elems1} : s \notin \mathcal{D} P \implies s @ [c] \in \mathcal{D} P \implies s @ [c] \in \text{min-elems } (\mathcal{D} P)$
 $\langle \text{proof} \rangle$

lemma $\text{min-elems2} : s \notin \mathcal{D} P \implies s @ [c] \in \mathcal{D} P \implies P \sqsubseteq S \implies Q \sqsubseteq S \implies (s @ [c], \{\}) \in \mathcal{F} Q$
 $\langle \text{proof} \rangle$

lemma $\text{min-elems6} : s \notin \mathcal{D} P \implies s @ [c] \in \mathcal{D} P \implies P \sqsubseteq S \implies (s @ [c], \{\}) \in \mathcal{F} S$
 $\langle \text{proof} \rangle$

lemma $\text{ND-F-dir2} : s \notin \mathcal{D} P \implies (s, \{\}) \in \mathcal{F} P \implies P \sqsubseteq S \implies Q \sqsubseteq S \implies (s, \{\}) \in \mathcal{F} Q$
 $\langle \text{proof} \rangle$

lemma ND-F-dir2' : $s \notin \mathcal{D} P \implies s \in \mathcal{T} P \implies P \sqsubseteq S \implies Q \sqsubseteq S \implies s \in \mathcal{T} Q$
 $\langle \text{proof} \rangle$

lemma $\text{chain-lemma} : \llbracket \text{chain } S \rrbracket \implies S i \sqsubseteq S k \vee S k \sqsubseteq S i$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemmas $\text{Rep-Abs-LUB} = \text{Abs-process-inverse}[\text{simplified Rep-process}, \text{simplified}, \text{OF is-process-REP-LUB},$

simplified]

lemma *F-LUB*: $chain\ S \implies \mathcal{F}(lim\text{-}proc(range\ S)) = \bigcap (\mathcal{F} \text{ ' } range\ S)$
 $\langle proof \rangle$

lemma *D-LUB*: $chain\ S \implies \mathcal{D}(lim\text{-}proc(range\ S)) = \bigcap (\mathcal{D} \text{ ' } range\ S)$
 $\langle proof \rangle$

lemma *T-LUB*: $chain\ S \implies \mathcal{T}(lim\text{-}proc(range\ S)) = \bigcap (\mathcal{T} \text{ ' } range\ S)$
 $\langle proof \rangle$

schematic-goal *D-LUB-2*: $chain\ S \implies t \in \mathcal{D}(lim\text{-}proc(range\ S)) = ?X$
 $\langle proof \rangle$

schematic-goal *T-LUB-2*: $chain\ S \implies (t \in \mathcal{T}(lim\text{-}proc(range\ S))) = ?X$
 $\langle proof \rangle$

2.7 Process Refinement is a Partial Ordering

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

instantiation

process :: (type) ord

begin

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

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

instance $\langle proof \rangle$

end

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

lemma *le-approx-implies-le-ref*: $(P :: 'a\ process) \sqsubseteq Q \implies P \leq Q$
 $\langle proof \rangle$

lemma *le-ref1*: $P \leq Q \implies \mathcal{D}\ Q \subseteq \mathcal{D}\ P$
 $\langle proof \rangle$

lemma *le-ref2*: $P \leq Q \implies \mathcal{F}\ Q \subseteq \mathcal{F}\ P$
 $\langle proof \rangle$

lemma *le-ref2T* : $P \leq Q \implies \mathcal{T}\ Q \subseteq \mathcal{T}\ P$
 $\langle proof \rangle$

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

lemma *lim-proc-is-ub*: $\text{chain } S \implies \text{range } S <| \text{lim-proc } (\text{range } S)$
 ⟨proof⟩

lemma *lim-proc-is-lub1*: $\text{chain } S \implies \forall u. (\text{range } S <| u \longrightarrow \mathcal{D} u \subseteq \mathcal{D} (\text{lim-proc } (\text{range } S)))$
 ⟨proof⟩

lemma *lim-proc-is-lub2*:
 $\text{chain } S \implies \forall u. \text{range } S <| u \longrightarrow (\forall s. s \notin \mathcal{D} (\text{lim-proc } (\text{range } S))$
 $\longrightarrow \text{Ra } (\text{lim-proc } (\text{range } S)) s = \text{Ra } u s)$
 ⟨proof⟩

lemma *lim-proc-is-lub3a*: $\text{front-tickFree } s \implies s \notin \mathcal{D} P \longrightarrow (\forall t. t \in \mathcal{D} P \longrightarrow \neg t < s @ [c])$
 ⟨proof⟩

lemma *lim-proc-is-lub3b*:
assumes $1 : \forall x. x \in X \longrightarrow (\forall xa. xa \in \mathcal{D} x \wedge (\forall t. t \in \mathcal{D} x \longrightarrow \neg t < xa) \longrightarrow xa \in \mathcal{T} u)$
and $2 : xa \in X$
and $3 : \forall xa. xa \in X \longrightarrow x \in \mathcal{D} xa$
and $4 : \forall t. (\forall x. x \in X \longrightarrow t \in \mathcal{D} x) \longrightarrow \neg t < x$
shows $x \in \mathcal{T} u$
 ⟨proof⟩

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

lemma *limproc-is-lub*: $\text{chain } S \implies \text{range } S <<| \text{lim-proc } (\text{range } S)$
 ⟨proof⟩

lemma *limproc-is-thelub*: $\text{chain } S \implies \text{Lub } S = \text{lim-proc } (\text{range } S)$
 ⟨proof⟩

instance
process :: (type) cpo

<proof>

instance

process :: (type) *pcpo*
<proof>

2.8 Process Refinement is Admissible

lemma *le-adm[simp]*: $\text{cont } (u::('a::\text{cpo}) \Rightarrow 'b \text{ process}) \Longrightarrow \text{monofun } v \Longrightarrow \text{adm}(\lambda x. u x \leq v x)$
<proof>

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

2.9 The Conditional Statement is Continuous

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

lemma *if-then-else-cont[simp]*:
assumes *: $(\bigwedge x. P x \Longrightarrow \text{cont } ((f::'c \Rightarrow ('a::\text{cpo}) \Rightarrow 'b \text{ process}) x))$
and **: $(\bigwedge x. \neg P x \Longrightarrow \text{cont } (g x))$
shows $\bigwedge x. \text{cont}(\lambda y. \text{if } P x \text{ then } f x y \text{ else } g x y)$
<proof>

end

Chapter 3

The CSP Operators

3.1 The Undefined Process

```
theory    Bot
imports  Process
begin
```

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

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

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

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

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

```
lemma BOT-is-UU[simp]:  $BOT = \perp$ 
  ⟨proof⟩
```

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

```
lemma D-UU:  $\mathcal{D} \perp = \{d. \text{front-tickFree } d\}$ 
  ⟨proof⟩
```

```
lemma T-UU:  $\mathcal{T} \perp = \{s. \text{front-tickFree } s\}$ 
  ⟨proof⟩
```

lemma *BOT-iff-D*: $\langle P = \perp \longleftrightarrow [] \in \mathcal{D} P \rangle$
<proof>

end

3.2 The SKIP Process

theory *Skip*
imports *Process*
begin

lift-definition *SKIP* :: $\langle 'a \text{ process} \rangle$
is $\langle \{(s, X). s = [] \wedge tick \notin X\} \cup \{(s, X). s = [tick]\}, \{\} \rangle$
<proof>

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

lemma *D-SKIP*: $\mathcal{D} SKIP = \{\}$
<proof>

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

end

3.3 The STOP Process

theory *Stop*
imports *Process*
begin

lift-definition *STOP* :: $\langle 'a \text{ process} \rangle$
is $\langle \{(s, X). s = []\}, \{\} \rangle$
<proof>

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

lemma *D-STOP*: $\mathcal{D} STOP = \{\}$
<proof>

lemma *T-STOP*: $\mathcal{T} \text{ STOP} = \{\square\}$
<proof>

lemma *STOP-iff-T*: $\langle P = \text{STOP} \longleftrightarrow \mathcal{T} P = \{\square\} \rangle$
<proof>

end

3.4 Deterministic Choice Operator Definition

theory *Det*
imports *Process*
begin

3.4.1 The Det Operator Definition

lift-definition

Det :: [*'α process, 'α process*] \Rightarrow *'α process* (**infixl** [+] 79)
is $\lambda P Q. (\{(s, X). s = \square \wedge (s, X) \in \mathcal{F} P \cap \mathcal{F} Q\}$
 $\cup \{(s, X). s \neq \square \wedge (s, X) \in \mathcal{F} P \cup \mathcal{F} Q\}$
 $\cup \{(s, X). s = \square \wedge s \in \mathcal{D} P \cup \mathcal{D} Q\}$
 $\cup \{(s, X). s = \square \wedge \text{tick} \notin X \wedge [\text{tick}] \in \mathcal{T} P \cup \mathcal{T} Q\},$
 $\mathcal{D} P \cup \mathcal{D} Q)$

<proof>

notation

Det (**infixl** \square 79)

term $\langle (A \square B) \square D' = C \rangle$

3.4.2 The Projections

lemma *F-Det* :

$\mathcal{F} (P \square Q) = \{(s, X). s = \square \wedge (s, X) \in \mathcal{F} P \cap \mathcal{F} Q\}$
 $\cup \{(s, X). s \neq \square \wedge (s, X) \in \mathcal{F} P \cup \mathcal{F} Q\}$
 $\cup \{(s, X). s = \square \wedge s \in \mathcal{D} P \cup \mathcal{D} Q\}$
 $\cup \{(s, X). s = \square \wedge \text{tick} \notin X \wedge [\text{tick}] \in \mathcal{T} P \cup \mathcal{T} Q\}$

<proof>

lemma *D-Det*: $\mathcal{D} (P \square Q) = \mathcal{D} P \cup \mathcal{D} Q$
<proof>

lemma *T-Det*: $\mathcal{T} (P \square Q) = \mathcal{T} P \cup \mathcal{T} Q$
<proof>

3.4.3 Basic Laws

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

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

3.4.4 The Continuity-Rule

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

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

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

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

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

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

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

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

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

and $g: cont\ g$
shows $cont\ (\lambda x. f\ x\ \sqcap\ g\ x)$
 $\langle proof \rangle$
end

3.5 Nondeterministic Choice Operator Definition

theory *Ndet*
imports *Process*
begin

3.5.1 The Ndet Operator Definition

lift-definition

$Ndet :: [\alpha\ process, \alpha\ process] \Rightarrow \alpha\ process$ (**infixl** $|-|$ 80)
is $\lambda P\ Q. (\mathcal{F}\ P \cup \mathcal{F}\ Q, \mathcal{D}\ P \cup \mathcal{D}\ Q)$
 $\langle proof \rangle$

notation

$Ndet$ (**infixl** \sqcap 80)

3.5.2 The Projections

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

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

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

3.5.3 Basic Laws

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

lemma *Ndet-commute*: $(P \sqcap Q) = (Q \sqcap P)$
 $\langle proof \rangle$

3.5.4 The Continuity Rule

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

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

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

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

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

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

lemma *Ndet-cont[simp]*:
assumes $f: \text{cont } f$
and $g: \text{cont } g$
shows $\text{cont } (\lambda x. f x \sqcap g x)$
 ⟨proof⟩

end

3.6 The Sequence Operator

theory *Seq*
imports *Process*
begin

3.6.1 Definition

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

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

3.6.2 The Projections

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

$\langle proof \rangle \quad \{(t1, X) \mid t1 \ X. \ t1 \in \mathcal{D} \ P\}$

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

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

3.6.3 Continuity Rule

lemma *mono-Seq-D11*:
 $P \sqsubseteq Q \implies \mathcal{D} \ (Q ; S) \subseteq \mathcal{D} \ (P ; S)$
 $\langle proof \rangle$

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

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

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

lemma *mono-Seq[simp]* : $P \sqsubseteq Q \implies (P ; S) \sqsubseteq (Q ; S)$
 $\langle proof \rangle$

lemma *mono-Seq-D21*:
 $P \sqsubseteq Q \implies \mathcal{D} \ (S ; Q) \subseteq \mathcal{D} \ (S ; P)$
 $\langle proof \rangle$

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

lemma *mono-Seq-D23* :

assumes *ordered*: $P \sqsubseteq Q$

shows $\text{min-elems } (\mathcal{D} (S ; P)) \subseteq \mathcal{T} (S ; Q)$

<proof>

lemma *mono-Seq-sym[simp]* : $P \sqsubseteq Q \implies (S ; P) \sqsubseteq (S ; Q)$

<proof>

lemma *chain-Seq1*: $\text{chain } Y \implies \text{chain } (\lambda i. Y i ; S)$

<proof>

lemma *chain-Seq2*: $\text{chain } Y \implies \text{chain } (\lambda i. S ; Y i)$

<proof>

lemma *limproc-Seq-D1*: $\text{chain } Y \implies \mathcal{D} (\text{lim-proc } (\text{range } Y) ; S) = \mathcal{D} (\text{lim-proc } (\text{range } (\lambda i. Y i ; S)))$

<proof>

lemma *limproc-Seq-F1*: $\text{chain } Y \implies \mathcal{F} (\text{lim-proc } (\text{range } Y) ; S) = \mathcal{F} (\text{lim-proc } (\text{range } (\lambda i. Y i ; S)))$

<proof>

lemma *cont-left-D-Seq* : $\text{chain } Y \implies ((\bigsqcup i. Y i) ; S) = (\bigsqcup i. (Y i ; S))$

<proof>

lemma *limproc-Seq-D2*: $\text{chain } Y \implies \mathcal{D} (S ; \text{lim-proc } (\text{range } Y)) = \mathcal{D} (\text{lim-proc } (\text{range } (\lambda i. S ; Y i)))$

<proof>

lemma *limproc-Seq-F2*:

$\text{chain } Y \implies \mathcal{F} (S ; \text{lim-proc } (\text{range } Y)) = \mathcal{F} (\text{lim-proc } (\text{range } (\lambda i. S ; Y i)))$

<proof>

lemma *cont-right-D-Seq* : $\text{chain } Y \implies (S ; (\bigsqcup i. Y i)) = (\bigsqcup i. (S ; Y i))$

<proof>

lemma *Seq-cont[simp]*:

assumes *f:cont* f

and *g:cont* g

shows $\text{cont } (\lambda x. f x ; g x)$

<proof>

end

3.7 The Renaming Operator

```
theory Renaming
  imports Process
begin
```

3.7.1 Some preliminaries

definition *EvExt* where $\langle \text{EvExt } f \ x \equiv \text{case-event } (ev \ o \ f) \ \text{tick } x \rangle$

term $\langle f \ -' \ B \rangle$

definition *finitary* :: $\langle ('a \Rightarrow 'b) \Rightarrow \text{bool} \rangle$
where $\langle \text{finitary } f \equiv \forall x. \text{finite}(f \ -' \ \{x\}) \rangle$

We start with some simple results.

lemma $\langle f \ -' \ \{\} = \{\} \rangle$ *<proof>*

lemma $\langle X \subseteq Y \Longrightarrow f \ -' \ X \subseteq f \ -' \ Y \rangle$ *<proof>*

lemma $\langle f \ -' \ (X \cup Y) = f \ -' \ X \cup f \ -' \ Y \rangle$ *<proof>*

lemma *EvExt-id*: $\langle \text{EvExt } id = id \rangle$
<proof>

lemma *EvExt-eq-tick*: $\langle \text{EvExt } f \ a = \text{tick} \longleftrightarrow a = \text{tick} \rangle$
<proof>

lemma *tick-eq-EvExt*: $\langle \text{tick} = \text{EvExt } f \ a \longleftrightarrow a = \text{tick} \rangle$
<proof>

lemma *EvExt-ev1*: $\langle \text{EvExt } f \ b = ev \ a \longleftrightarrow (\exists c. b = ev \ c \wedge \text{EvExt } f \ (ev \ c) = ev \ a) \rangle$
<proof>

lemma *EvExt-tF*: $\langle \text{tickFree } (\text{map } (\text{EvExt } f) \ s) \longleftrightarrow \text{tickFree } s \rangle$
<proof>

lemma *inj-EvExt*: $\langle \text{inj } \text{EvExt} \rangle$
<proof>

lemma *EvExt-ftF*: $\langle \text{front-tickFree } (\text{map } (\text{EvExt } f) \ s) \longleftrightarrow \text{front-tickFree } s \rangle$
<proof>

lemma *map-EvExt-tick*: $\langle [\text{tick}] = \text{map } (\text{EvExt } f) \ t \longleftrightarrow t = [\text{tick}] \rangle$

$\langle proof \rangle$

lemma *antedec-EvExt-diff-tick*: $\langle EvExt f - ' (X - \{tick\}) = EvExt f - ' X - \{tick\} \rangle$
 $\langle proof \rangle$

lemma *ev-elem-antedec1*: $\langle ev a \in EvExt f - ' A \longleftrightarrow ev (f a) \in A \rangle$
and *tick-elem-antedec1*: $\langle tick \in EvExt f - ' A \longleftrightarrow tick \in A \rangle$
 $\langle proof \rangle$

lemma *hd-map-EvExt*: $\langle t \neq [] \implies hd t = ev a \implies hd (map (EvExt f) t) = ev (f a) \rangle$
and *tl-map-EvExt*: $\langle t \neq [] \implies tl (map (EvExt f) t) = map (EvExt f) (tl t) \rangle$
 $\langle proof \rangle$

3.7.2 The Renaming Operator Definition

lift-definition *Renaming* :: $\langle 'a \text{ process}, 'a \Rightarrow 'b \Rightarrow 'b \text{ process} \rangle$
is $\langle \lambda P f. (\{(s, R). \exists s1. (s1, (EvExt f) - ' R) \in \mathcal{F} P \wedge s = map (EvExt f) s1\} \cup \{(s, R). \exists s1 s2. tickFree s1 \wedge front-tickFree s2 \wedge s = (map (EvExt f) s1) @ s2 \wedge s1 \in \mathcal{D} P\}, \{t. \exists s1 s2. tickFree s1 \wedge front-tickFree s2 \wedge t = (map (EvExt f) s1) @ s2 \wedge s1 \in \mathcal{D} P\}) \rangle$
 $\langle proof \rangle$

Some syntactic sugar

syntax

-Renaming :: $\langle 'a \text{ process} \Rightarrow updbinds \Rightarrow 'a \text{ process} \rangle (\langle -[-] \rangle [100, 100])$

translations

-Renaming $P \text{ updates} \equiv CONST \text{ Renaming } P (-Update (CONST id) \text{ updates})$

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

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

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

lemma $\langle P[x := y, x := z] = P[x := z] \rangle \langle proof \rangle$

lemma $\langle a \neq c \implies P[a := b, c := d] = P[c := d, a := b] \rangle \langle proof \rangle$

3.7.3 The Projections

lemma *F-Renaming*: $\langle \mathcal{F} (Renaming P f) =$

$$\{(s, R). \exists s1. (s1, EvExt f -' R) \in \mathcal{F} P \wedge s = \text{map } (EvExt f) s1\} \cup$$

$$\{(s, R). \exists s1 s2. \text{tickFree } s1 \wedge \text{front-tickFree } s2 \wedge s = \text{map } (EvExt f) s1 @ s2 \wedge$$

$$s1 \in \mathcal{D} P\}$$

<proof>

lemma *D-Renaming*: $\langle \mathcal{D} (\text{Renaming } P f) = \{t. \exists s1 s2. \text{tickFree } s1 \wedge \text{front-tickFree } s2 \wedge$

$$t = \text{map } (EvExt f) s1 @ s2 \wedge s1 \in \mathcal{D} P\} \rangle$$

<proof>

lemma *T-Renaming*: $\langle \mathcal{T} (\text{Renaming } P f) =$

$$\{t. \exists t1. t1 \in \mathcal{T} P \wedge t = \text{map } (EvExt f) t1\} \cup$$

$$\{t. \exists t1 t2. \text{tickFree } t1 \wedge \text{front-tickFree } t2 \wedge t = \text{map } (EvExt f) t1 @ t2 \wedge t1 \in \mathcal{D} P\} \rangle$$

<proof>

3.7.4 Continuity Rule

Monotonicity of Renaming.

lemma *mono-Renaming[simp]*: $\langle P \sqsubseteq Q \implies (\text{Renaming } P f) \sqsubseteq (\text{Renaming } Q f) \rangle$

<proof>

lemma $\langle \{ev\ c \mid c. f\ c = a\} = ev\ \{c. f\ c = a\} \rangle$ *<proof>*

lemma *vimage-EvExt-subset-vimage*: $\langle EvExt\ f\ -' (ev\ -' A) \subseteq \text{insert tick } (ev\ -' (f\ -' A)) \rangle$

and *vimage-subset-vimage-EvExt*: $\langle (ev\ -' (f\ -' A)) \subseteq (EvExt\ f\ -' (ev\ -' A)) - \{tick\} \rangle$

<proof>

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

lemma *inj-imp-finitary[simp]*: $\langle \text{inj } f \implies \text{finitary } f \rangle$ *<proof>*

lemma *finitary-comp-iff*: $\langle \text{finitary } g \implies \text{finitary } (g\ o\ f) \longleftrightarrow \text{finitary } f \rangle$

<proof>

lemma *finitary-updated-iff[simp]*: $\langle \text{finitary } (f\ (a := b)) \longleftrightarrow \text{finitary } f \rangle$

<proof>

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

lemma $\langle \text{finitary } f \longleftrightarrow \text{finitary } (f\ (a := b, c := d, e := g, h := i)) \rangle$ *<proof>*

lemma *Cont-RenH2*: $\langle \text{finitary } (EvExt f) \longleftrightarrow \text{finitary } f \rangle$
 $\langle \text{proof} \rangle$

lemma *Cont-RenH3*:
 $\langle \{s1 @ t1 \mid s1 t1. (\exists b. s1 = [b] \ \& \ f \ b = a) \wedge \text{list} = \text{map } f \ t1\} =$
 $(\bigcup b \in f - \{a\}. (\lambda t. b \# t) \cdot \{t. \text{list} = \text{map } f \ t\}) \rangle$
 $\langle \text{proof} \rangle$

lemma *Cont-RenH4*: $\langle \text{finitary } f \longleftrightarrow (\forall s. \text{finite } \{t. s = \text{map } f \ t\}) \rangle$
 $\langle \text{proof} \rangle$

lemma *Cont-RenH5*: $\langle \text{finitary } f \implies \text{finite } (\bigcup t \in \{t. t \leq s\}. \{s. t = \text{map } (EvExt f) \ s\}) \rangle$
 $\langle \text{proof} \rangle$

lemma *Cont-RenH6*: $\langle \{t. \exists t2. \text{tickFree } t \wedge \text{front-tickFree } t2 \wedge x = \text{map } (EvExt f) \ t @ t2\}$
 $\subseteq \{y. \exists xa. xa \in \{t. t \leq x\} \wedge y \in \{s. xa = \text{map } (EvExt f) \ s\}\} \rangle$
 $\langle \text{proof} \rangle$

lemma *Cont-RenH7*: $\langle \text{finite } \{t. \exists t2. \text{tickFree } t \wedge \text{front-tickFree } t2 \wedge s = \text{map } (EvExt f) \ t @ t2\} \rangle$
if $\langle \text{finitary } f \rangle$
 $\langle \text{proof} \rangle$

lemma *chain-Renaming*: $\langle \text{chain } Y \implies \text{chain } (\lambda i. \text{Renaming } (Y \ i) \ f) \rangle$
 $\langle \text{proof} \rangle$

A key lemma for the continuity.

lemma *Inter-nonempty-finite-chained-sets*:
 $\langle \forall i. S \ i \neq \{\} \implies \text{finite } (S \ 0) \implies \forall i. S \ (Suc \ i) \subseteq S \ i \implies (\bigcap i. S \ i) \neq \{\} \rangle$
 $\langle \text{proof} \rangle$

Finally, continuity !

lemma *Cont-Renaming-prem*:

$\langle (\bigsqcup i. \text{Renaming } (Y i) f) = (\text{Renaming } (\bigsqcup i. Y i) f) \rangle$ **if** *finitary*: $\langle \text{finitary } f \rangle$
and *chain*: $\langle \text{chain } Y \rangle$
 $\langle \text{proof} \rangle$

lemma *Renaming-cont[simp]* : $\langle \text{cont } P \implies \text{finitary } f \implies \text{cont } (\lambda x. (\text{Renaming } (P x) f)) \rangle$
 $\langle \text{proof} \rangle$

lemma *RenamingF-cont* : $\langle \text{cont } P \implies \text{cont } (\lambda x. (P x)[[a := b]]) \rangle$
 $\langle \text{proof} \rangle$

lemma $\langle \text{cont } P \implies \text{cont } (\lambda x. (P x)[[a := b, c := d, e := f, g := a]]) \rangle$
 $\langle \text{proof} \rangle$

3.7.5 Some nice properties

lemma *EvExt-comp*: $\langle \text{EvExt } (g \circ f) = \text{EvExt } g \circ \text{EvExt } f \rangle$
 $\langle \text{proof} \rangle$

lemma *Renaming-comp* : $\langle \text{Renaming } P (g \circ f) = \text{Renaming } (\text{Renaming } P f) g \rangle$
 $\langle \text{proof} \rangle$

lemma *Renaming-id*: $\langle \text{Renaming } P \text{ id} = P \rangle$
 $\langle \text{proof} \rangle$

lemma *Renaming-inv*: $\langle \text{Renaming } (\text{Renaming } P f) (\text{inv } f) = P \rangle$ **if** $\langle \text{inj } f \rangle$
 $\langle \text{proof} \rangle$

end

Chapter 4

Concurrent CSP Operators

4.1 The Hiding Operator

theory *Hiding*
imports *Process*
begin

4.1.1 Preliminaries : primitives and lemmas

abbreviation *trace-hide* $t A \equiv \text{filter } (\lambda x. x \notin A) t$

lemma *Hiding-tickFree* : $\text{tickFree } s \longleftrightarrow \text{tickFree } (\text{trace-hide } s (ev' A))$
<proof>

lemma *Hiding-fronttickFree* : $\text{front-tickFree } s \implies \text{front-tickFree } (\text{trace-hide } s (ev' A))$
<proof>

lemma *trace-hide-union[simp]* : $\text{trace-hide } t (ev' (A \cup B)) = \text{trace-hide } (\text{trace-hide } t (ev' A)) (ev' B)$
<proof>

abbreviation *isInfHiddenRun* $f P A \equiv \text{strict-mono } f \wedge (\forall i. f i \in \mathcal{T} P) \wedge$
 $(\forall i. \text{trace-hide } (f i) (ev' A) = \text{trace-hide } (f (0::nat))$
 $(ev' A))$

lemma *isInfHiddenRun-1*: $\text{isInfHiddenRun } f P A \longleftrightarrow \text{strict-mono } f \wedge (\forall i. f i \in \mathcal{T} P) \wedge$
 $(\forall i. \exists t. f i = f 0 @ t \wedge \text{set } t \subseteq (ev' A))$
 $(\text{is } ?A \longleftrightarrow ?B)$
<proof>

abbreviation *div-hide* $P A \equiv \{s. \exists t u. \text{front-tickFree } u \wedge$
 $\text{tickFree } t \wedge s = \text{trace-hide } t (ev' A) @ u \wedge$
 $(t \in \mathcal{D} P \vee (\exists f. \text{isInfHiddenRun } f P A \wedge t \in$

$range\ f))\}$

lemma *inf-hidden*:

assumes $as1:\forall t. trace\text{-}hide\ t\ (ev\ 'A) = trace\text{-}hide\ s\ (ev\ 'A) \longrightarrow (t, ev\ 'A) \notin \mathcal{F}\ P$

and $as2:s \in \mathcal{T}\ P$

shows $\exists f. isInfHiddenRun\ f\ P\ A \wedge s \in range\ f$

$\langle proof \rangle$

lemma *trace-hide-append*: $s @ t = trace\text{-}hide\ ta\ (ev\ 'A) \implies \exists ss\ tt. ta = ss @ tt \wedge$

$$s = trace\text{-}hide\ ss\ (ev\ 'A) \wedge$$

$$t = trace\text{-}hide\ tt\ (ev\ 'A)$$

$\langle proof \rangle$

4.1.2 The Hiding Operator Definition

lift-definition *Hiding* :: $['\alpha\ process, '\alpha\ set] \Rightarrow '\alpha\ process$ ($\langle - \rangle \rightarrow [57,56] 57$)

is

$$\langle \lambda P\ A. (\{(s,X). \exists t. s = trace\text{-}hide\ t\ (ev\ 'A) \wedge (t, X \cup (ev\ 'A)) \in \mathcal{F}\ P\} \cup$$

$$\{(s,X). s \in div\text{-}hide\ P\ A\}, div\text{-}hide\ P\ A) \rangle$$

$\langle proof \rangle$

4.1.3 Projections

lemma *F-Hiding*: $\langle \mathcal{F}\ (P \setminus A) = \{(s,X). \exists t. s = trace\text{-}hide\ t\ (ev\ 'A) \wedge (t, X \cup (ev\ 'A)) \in \mathcal{F}\ P\} \cup$

$$\{(s,X). s \in div\text{-}hide\ P\ A\} \rangle$$

$\langle proof \rangle$

lemma *D-Hiding*: $\langle \mathcal{D}\ (P \setminus A) = div\text{-}hide\ P\ A \rangle$

$\langle proof \rangle$

lemma *T-Hiding*: $\langle \mathcal{T}\ (P \setminus A) = \{s. \exists t. s = trace\text{-}hide\ t\ (ev\ 'A) \wedge (t, ev\ 'A) \in \mathcal{F}\ P\} \cup div\text{-}hide\ P\ A \rangle$

$\langle proof \rangle$

4.1.4 Continuity Rule

lemma *mono-Hiding[simp]*: $\langle (P::'\alpha\ process) \sqsubseteq Q \implies (P \setminus A) \sqsubseteq (Q \setminus A) \rangle$

$\langle proof \rangle$

lemma *cont-Hiding1*: $\langle chain\ Y \implies chain\ (\lambda i. (Y\ i \setminus A)) \rangle$

$\langle proof \rangle$

lemma *KoenigLemma*:

assumes $*:infinite\ (Tr::'\alpha\ list\ set)$ **and** $**:\forall i. finite\ \{t. \exists t' \in Tr. t = take\ i\ t'\}$

shows $\exists (f::nat \Rightarrow '\alpha\ list). strict\text{-}mono\ f \wedge range\ f \subseteq \{t. \exists t' \in Tr. t \leq t'\}$

<proof>

lemma *div-Hiding-lub* :

<finite (A::'a set) \implies chain Y \implies $\mathcal{D} (\bigsqcup i. (Y i \setminus A)) \subseteq \mathcal{D} ((\bigsqcup i. Y i) \setminus A)$ >
<proof>

lemma *cont-Hiding2* : *<finite A \implies chain Y \implies $((\bigsqcup i. Y i) \setminus A) = (\bigsqcup i. (Y i \setminus A))$ >*

<proof>

lemma *cont-Hiding-base[simp]* : *<finite A \implies cont $(\lambda x. x \setminus A)$ >*

<proof>

lemma *Hiding-cont[simp]*: *<finite A \implies cont f \implies cont $(\lambda x. f x \setminus A)$ >*

<proof>

end

4.2 The Synchronizing Operator

theory *Sync*

imports *Process HOL-Library.Infinite-Set*

begin

4.2.1 Basic Concepts

fun *setinterleaving*:: *'a trace \times ('a event) set \times 'a trace \Rightarrow ('a trace) set*

where

si-empty1: *setinterleaving(\square , X, \square) = $\{\square\}$*
| *si-empty2*: *setinterleaving(\square , X, (y # t)) =*
 (if (y \in X)
 then $\{\}$
 else $\{z.\exists u. z = (y \# u) \wedge u \in \text{setinterleaving}(\square, X, t)\}$)
| *si-empty3*: *setinterleaving((x # s), X, \square) =*
 (if (x \in X)
 then $\{\}$
 else $\{z.\exists u. z = (x \# u) \wedge u \in \text{setinterleaving}(s, X, \square)\}$)
| *si-neg* : *setinterleaving((x # s), X, (y # t)) =*
 (if (x \in X)
 then if (y \in X)
 then if (x = y)
 then $\{z.\exists u. z = (x \# u) \wedge u \in \text{setinterleaving}(s, X, t)\}$
 else $\{\}$
 else $\{z.\exists u. z = (y \# u) \wedge u \in \text{setinterleaving}((x \# s), X, t)\}$
 else if (y \notin X)
 then $\{z.\exists u. z = (x \# u) \wedge u \in \text{setinterleaving}(s, X, (y \# t))\}$
 $\cup \{z.\exists u. z = (y \# u) \wedge u \in \text{setinterleaving}((x \# s), X, t)\}$

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

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

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

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

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

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

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

4.2.2 Consequences

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

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

lemma *emptyLeftNonSync*: $s \text{ setinterleaves } ([], t, A) \implies \forall a. a \in \text{set } t \longrightarrow a \notin A$
 \(\langle proof \rangle\)

lemma *ftf-Sync1*: $a \notin \text{set}(u) \wedge a \notin \text{set}(t) \wedge s \text{ setinterleaves } ((t, u), A) \longrightarrow a \notin \text{set}(s)$
 \(\langle proof \rangle\)

lemma *addNonSyncEmpty*: $sa \text{ setinterleaves } ([], u, A) \wedge y1 \notin A \longrightarrow (sa @ [y1]) \text{ setinterleaves } ([y1], u, A) \wedge (sa @ [y1]) \text{ setinterleaves } ([], u @ [y1], A)$
 \(\langle proof \rangle\)

lemma *addNonSync*: $sa \text{ setinterleaves } ((t, u), A) \wedge y1 \notin A \longrightarrow (sa @ [y1]) \text{ setinterleaves } ((t @ [y1], u), A) \wedge (sa @ [y1]) \text{ setinterleaves } ((t, u @ [y1]), A)$
 \(\langle proof \rangle\)

lemma *addSyncEmpty*: $sa \text{ setinterleaves } ([], u, A) \wedge y1 \in A \longrightarrow (sa @ [y1]) \text{ setinterleaves } ([y1], u @ [y1], A)$
 \(\langle proof \rangle\)

lemma *addSync*: $sa \text{ setinterleaves } ((t, u), A) \wedge y1 \in A \longrightarrow (sa @ [y1]) \text{ setinterleaves } ((t @ [y1], u @ [y1]), A)$
find-theorems \(\langle @ \rangle\) *name: cases*
 \(\langle proof \rangle\)

lemma *doubleReverse*: $s1 \text{ setinterleaves } ((t, u), A) \longrightarrow (\text{rev } s1) \text{ setinterleaves } ((\text{rev } t, \text{rev } u), A)$
 \(\langle proof \rangle\)

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

lemma *ftf-Sync32*: $u = u1 @ [\text{tick}] \wedge t = t1 @ [\text{tick}] \wedge s \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } A)) \implies \exists s1. s1 \text{ setinterleaves } ((t1, u1), \text{insert tick } (ev \text{ ' } A)) \wedge (s = s1 @ [\text{tick}])$
 \(\langle proof \rangle\)

lemma *SyncWithTick-imp-NTF*:
 \(\langle s @ [\text{tick}] \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } A)) \implies t \in \mathcal{T} P \implies u \in \mathcal{T} Q \implies \exists t1 u1. t = t1 @ [\text{tick}] \wedge u = u1 @ [\text{tick}] \wedge s \text{ setinterleaves } ((t1, u1), \text{insert tick } (ev \text{ ' } A)) \rangle\)

<proof>

lemma *synPrefix1*:

$$\begin{aligned} & ta = [] \\ \implies & \exists t1\ u1. (s @ t)\ \text{setinterleaves}\ ((ta, u), A) \longrightarrow \\ & t1 \leq ta \wedge u1 \leq u \wedge s\ \text{setinterleaves}\ ((t1, u1), A) \end{aligned}$$

<proof>

lemma *synPrefix*: $\exists t1\ u1. (s @ t)\ \text{setinterleaves}\ ((ta, u), A) \longrightarrow$
 $t1 \leq ta \wedge u1 \leq u \wedge s\ \text{setinterleaves}\ ((t1, u1), A)$

<proof>

lemma *SyncWithTick-imp-NTF1*:

$$\begin{aligned} & (s @ [tick])\ \text{setinterleaves}\ ((t, u), \text{insert tick}\ (ev\ 'A)) \\ \implies & t \in \mathcal{T}\ P \implies u \in \mathcal{T}\ Q \\ \implies & \exists t\ u\ Xa\ Y. (t, Xa) \in \mathcal{F}\ P \wedge (u, Y) \in \mathcal{F}\ Q \wedge \\ & s\ \text{setinterleaves}\ ((t, u), \text{insert tick}\ (ev\ 'A)) \wedge \\ & X - \{tick\} = (Xa \cup Y) \cap \text{insert tick}\ (ev\ 'A) \cup Xa \cap Y \end{aligned}$$

<proof>

lemma *ftf-Sync*:

$$\begin{aligned} & \text{front-tickFree}\ u \\ \implies & \text{front-tickFree}\ t \\ \implies & s\ \text{setinterleaves}\ ((t, u), \text{insert tick}\ (ev\ 'A)) \\ \implies & \text{front-tickFree}\ s \end{aligned}$$

<proof>

lemma *is-processT5-SYNC2*:

$$\begin{aligned} & \bigwedge sa\ Y\ t\ u\ Xa\ Ya. (t, Xa) \in \mathcal{F}\ P \wedge (u, Ya) \in \mathcal{F}\ Q \wedge (sa\ \text{setinterleaves}\ ((t, u), \text{insert tick}\ (ev\ 'A))) \\ \implies & \forall c. c \in Y \longrightarrow \\ & (\forall t1\ u1. (sa @ [c])\ \text{setinterleaves}\ ((t1, u1), \text{insert tick}\ (ev\ 'A)) \longrightarrow \\ & (((t1, \{\}) \in \mathcal{F}\ P \longrightarrow (u1, \{\}) \notin \mathcal{F}\ Q) \wedge ((t1, \{\}) \in \mathcal{F}\ Q \longrightarrow (u1, \{\}) \notin \mathcal{F}\ P))) \\ \implies & \exists t2\ u2\ Xb. (t2, Xb) \in \mathcal{F}\ P \wedge \\ & (\exists Yb. (u2, Yb) \in \mathcal{F}\ Q \wedge \\ & sa\ \text{setinterleaves}\ ((t2, u2), \text{insert tick}\ (ev\ 'A)) \wedge \\ & (Xa \cup Ya) \cap \text{insert tick}\ (ev\ 'A) \cup Xa \cap Ya \cup Y = \\ & (Xb \cup Yb) \cap \text{insert tick}\ (ev\ 'A) \cup Xb \cap Yb) \end{aligned}$$

<proof>

lemma *pt3*:

$$\begin{aligned} & ta \in \mathcal{T}\ P \\ \implies & u \in \mathcal{T}\ Q \implies (s @ t)\ \text{setinterleaves}\ ((ta, u), \text{insert tick}\ (ev\ 'A)) \\ \implies & \exists t\ u\ X. (t, X) \in \mathcal{F}\ P \wedge \\ & (\exists Y. (u, Y) \in \mathcal{F}\ Q \wedge \\ & s\ \text{setinterleaves}\ ((t, u), \text{insert tick}\ (ev\ 'A)) \wedge \\ & tick \notin X \wedge tick \notin Y \wedge (X \cup Y) \cap ev\ 'A = \{\} \wedge X \cap Y = \{\}) \end{aligned}$$

<proof>

4.2.3 The Sync Operator Definition

lift-definition *Sync* :: [*'a process, 'a set, 'a process*] => *'a process* (($\exists(- \llbracket - \rrbracket / -)$) [64,0,65] 64)

is $\lambda P A Q. \{(s,R). \exists t u X Y. (t,X) \in \mathcal{F} P \wedge (u,Y) \in \mathcal{F} Q \wedge$
 $(s \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\})) \wedge$
 $R = (X \cup Y) \cap ((ev'A) \cup \{tick\}) \cup X \cap Y\} \cup$
 $\{(s,R). \exists t u r v. \text{front-tickFree } v \wedge (\text{tickFree } r \vee v = []) \wedge$
 $s = r @ v \wedge$
 $(r \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\})) \wedge$
 $(t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P)\},$

$\{s. \exists t u r v. \text{front-tickFree } v \wedge (\text{tickFree } r \vee v = []) \wedge$
 $s = r @ v \wedge$
 $(r \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\})) \wedge$
 $(t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P)\}$

<proof>

4.2.4 The Projections

lemma

F-Sync : $\mathcal{F} (P \llbracket A \rrbracket Q) =$
 $\{(s,R). \exists t u X Y. (t,X) \in \mathcal{F} P \wedge$
 $(u,Y) \in \mathcal{F} Q \wedge$
 $s \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\}) \wedge$
 $R = (X \cup Y) \cap ((ev'A) \cup \{tick\}) \cup X \cap Y\} \cup$
 $\{(s,R). \exists t u r v. \text{front-tickFree } v \wedge$
 $(\text{tickFree } r \vee v = []) \wedge$
 $s = r @ v \wedge$
 $r \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\}) \wedge$
 $(t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P)\}$

<proof>

lemma

D-Sync : $\mathcal{D} (P \llbracket A \rrbracket Q) =$
 $\{s. \exists t u r v. \text{front-tickFree } v \wedge (\text{tickFree } r \vee v = []) \wedge$
 $s = r @ v \wedge r \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\}) \wedge$
 $(t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P)\}$

<proof>

lemma

T-Sync : $\mathcal{T} (P \llbracket A \rrbracket Q) =$
 $\{s. \exists t u. t \in \mathcal{T} P \wedge u \in \mathcal{T} Q \wedge$
 $s \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\})\} \cup$
 $\{s. \exists t u r v. \text{front-tickFree } v \wedge (\text{tickFree } r \vee v = []) \wedge$
 $s = r @ v \wedge r \text{ setinterleaves } ((t,u), (ev'A) \cup \{tick\}) \wedge$
 $(t \in \mathcal{D} P \wedge u \in \mathcal{T} Q \vee t \in \mathcal{D} Q \wedge u \in \mathcal{T} P)\}$

<proof>

4.2.5 Syntax for Interleave and Parallel Operator

abbreviation *Inter-syntax* $((-||-)$ [72,73] 72)

where $P || Q \equiv (P \llbracket \{\} \rrbracket Q)$

abbreviation *Par-syntax* $((-||-)$ [74,75] 74)

where $P || Q \equiv (P \llbracket UNIV \rrbracket Q)$

4.2.6 Continuity Rule

lemma *mono-Sync-D1*:

$P \sqsubseteq Q \implies \mathcal{D}(Q \llbracket A \rrbracket S) \subseteq \mathcal{D}(P \llbracket A \rrbracket S)$

<proof>

lemma *mono-Sync-D2*:

$P \sqsubseteq Q \implies (\forall s. s \notin \mathcal{D}(P \llbracket A \rrbracket S) \longrightarrow Ra(P \llbracket A \rrbracket S) s = Ra(Q \llbracket A \rrbracket S) s)$

<proof>

lemma *interPrefixEmpty*: s setinterleaves $((t, \llbracket \rrbracket), A) \wedge t1 < t \implies \exists s1 < s. s1$ setinterleaves $((t1, \llbracket \rrbracket), A)$

<proof>

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

<proof>

lemma *mono-Sync2a*:

$r \in \text{min-elems}(\mathcal{D}(P \llbracket A \rrbracket S))$

$\implies \exists t u. r$ setinterleaves $((t, u), \text{insert tick}(ev \text{ ' } A)) \wedge$

$(t \in \text{min-elems}(\mathcal{D} P) \wedge u \in \mathcal{T} S \vee t \in \text{min-elems}(\mathcal{D} S) \wedge$

$u \in \mathcal{T} P \wedge u \in (\mathcal{T} P - (\mathcal{D} P - \text{min-elems}(\mathcal{D} P)))$)

<proof>

lemma *mono-Sync-D3*:

assumes *ordered*: $P \sqsubseteq Q$

shows $\text{min-elems}(\mathcal{D}(P \llbracket A \rrbracket S)) \subseteq \mathcal{T}(Q \llbracket A \rrbracket S)$

<proof>

lemma *mono-D-Sync*: $\mathcal{D}(S \llbracket A \rrbracket Q) = \mathcal{D}(Q \llbracket A \rrbracket S)$

<proof>

lemma *mono-T-Sync*: $\mathcal{T}(S \llbracket A \rrbracket Q) = \mathcal{T}(Q \llbracket A \rrbracket S)$

<proof>

lemma *mono-F-Sync*: $\mathcal{F}(S \llbracket A \rrbracket Q) = \mathcal{F}(Q \llbracket A \rrbracket S)$

<proof>

lemma *mono-Ra-Sync*: $Ra(S \llbracket A \rrbracket Q) s = Ra(Q \llbracket A \rrbracket S) s$

<proof>

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

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

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

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

lemma *empty-setinterleaving* : $\llbracket \text{setinterleaves } ((t, u), A) \implies t = \llbracket \text{proof} \rrbracket$

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

lemma *inters-fin*: $\text{finite}\{(t, u, r). r \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } A)) \wedge$
 $(\exists v. x = r @ v \wedge \text{front-tickFree } v \wedge (\text{tickFree } r \vee v$
 $= \llbracket \rrbracket))\}$
 (is finite ?A)
 ⟨proof⟩

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

lemma *limproc-Sync-F-annex1*:
 $\text{chain } Y$
 $\implies \forall n. (a, b) \in \mathcal{F}(Y n \llbracket A \rrbracket S)$
 $\implies \exists x. a \notin \mathcal{D}(Y x \llbracket A \rrbracket S)$
 $\implies \exists t u X. (\forall x. (t, X) \in \mathcal{F}(Y x)) \wedge$
 $(\exists Y. (u, Y) \in \mathcal{F} S \wedge a \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } A)) \wedge$
 $b = (X \cup Y) \cap \text{insert tick } (ev \text{ ' } A) \cup X \cap Y)$
 ⟨proof⟩

lemma *limproc-Sync-F-annex2*:
 $\text{chain } Y$
 $\implies \forall t u r. r \text{ setinterleaves } ((t, u), \text{insert tick } (ev \text{ ' } A)) \longrightarrow$
 $(\forall v. a = r @ v \longrightarrow \text{front-tickFree } v \longrightarrow$
 $\neg \text{tickFree } r \wedge v \neq \llbracket \rrbracket \vee ((\exists x. t \notin \mathcal{D}(Y x)) \vee u \notin \mathcal{T} S) \wedge$
 $(t \in \mathcal{D} S \longrightarrow (\exists x. u \notin \mathcal{T}(Y x))))$
 $\implies \exists x. a \notin \mathcal{D}(Y x \llbracket A \rrbracket S)$
 ⟨proof⟩

lemma *limproc-Sync-F*:

chain $Y \implies \mathcal{F}(\text{lim-proc } (\text{range } Y) \llbracket A \rrbracket S) = \mathcal{F}(\text{lim-proc } (\text{range } (\lambda i. Y i \llbracket A \rrbracket S)))$
 ⟨*proof*⟩

lemma *cont-left-Sync* : *chain* $Y \implies ((\sqcup i. Y i) \llbracket A \rrbracket S) = (\sqcup i. (Y i \llbracket A \rrbracket S))$
 ⟨*proof*⟩

lemma *cont-right-Sync* : *chain* $Y \implies (S \llbracket A \rrbracket (\sqcup i. Y i)) = (\sqcup i. (S \llbracket A \rrbracket Y i))$
 ⟨*proof*⟩

lemma *Sync-cont[simp]*:

assumes $f:\text{cont } f$

and $g:\text{cont } g$

shows $\text{cont } (\lambda x. (f x) \llbracket A \rrbracket (g x))$

⟨*proof*⟩

end

4.3 The Multi-Prefix Operator Definition

theory *Mprefix*

imports *Process*

begin

4.3.1 The Definition and some Consequences

lift-definition *Mprefix* $:: ['a \text{ set}, 'a \Rightarrow 'a \text{ process}] \Rightarrow 'a \text{ process}$

is $\lambda A P. (\{(tr, ref). tr = [] \wedge ref \text{ Int } (ev \text{ ' } A) = \{\}\} \cup$

$\{(tr, ref). tr \neq [] \wedge hd \ tr \in (ev \text{ ' } A) \wedge$

$(\exists a. ev \ a = (hd \ tr) \wedge (tl \ tr, ref) \in \mathcal{F}(P \ a))\},$

$\{d. d \neq [] \wedge hd \ d \in (ev \text{ ' } A) \wedge (\exists a. ev \ a = hd \ d \wedge tl \ d \in \mathcal{D}(P \ a))\})$

⟨*proof*⟩

syntax

-Mprefix $:: [pttrn, 'a \text{ set}, 'a \text{ process}] \Rightarrow 'a \text{ process } ((\exists \square (-/\in -) / \rightarrow (-))[0,0,64]64)$

term *Ball* $A (\lambda x. P)$

translations

$\square x \in A \rightarrow P \rightleftharpoons \text{CONST } Mprefix \ A (\lambda x. P)$

Syntax Check:

term $\langle \square x \in A \rightarrow \square y \in A \rightarrow \square z \in A \rightarrow P \ z \ x \ y = Q \rangle$

lemma *is-process-REP-Mprefix'* :

$$\begin{aligned} \text{is-process} & \left(\{(tr, ref). tr = [] \wedge ref \cap (ev \text{ ' } A) = \{\}\} \cup \right. \\ & \{(tr, ref). tr \neq [] \wedge hd \ tr \in (ev \text{ ' } A) \wedge \\ & \quad (\exists a. ev \ a = (hd \ tr) \wedge (tl \ tr, ref) \in \mathcal{F}(P \ a))\}, \\ & \{d. d \neq [] \wedge hd \ d \in (ev \text{ ' } A) \wedge \\ & \quad (\exists a. ev \ a = hd \ d \wedge tl \ d \in \mathcal{D}(P \ a))\} \end{aligned}$$

(**is** *is-process*(?f, ?d))
 ⟨*proof*⟩

lemma *Rep-Abs-Mprefix''* :

$$\begin{aligned} \text{assumes } H1 : f & = \{(tr, ref). tr = [] \wedge ref \cap ev \text{ ' } A = \{\}\} \cup \\ & \{(tr, ref). tr \neq [] \wedge hd \ tr \in ev \text{ ' } A \\ & \quad \wedge (\exists a. ev \ a = hd \ tr \wedge (tl \ tr, ref) \in \mathcal{F}(P \ a))\} \\ \text{and } H2 : d & = \{d. d \neq [] \wedge hd \ d \in (ev \text{ ' } A) \wedge \\ & \quad (\exists a. ev \ a = hd \ d \wedge tl \ d \in \mathcal{D}(P \ a))\} \end{aligned}$$

shows *Rep-process* (*Abs-process* (f,d)) = (f,d)
 ⟨*proof*⟩

4.3.2 Projections in Prefix

lemma *F-Mprefix* :

$$\begin{aligned} \mathcal{F} (\Box x \in A \rightarrow P \ x) & = \{(tr, ref). tr = [] \wedge ref \cap (ev \text{ ' } A) = \{\}\} \cup \\ & \{(tr, ref). tr \neq [] \wedge hd \ tr \in (ev \text{ ' } A) \wedge \\ & \quad (\exists a. ev \ a = (hd \ tr) \wedge (tl \ tr, ref) \in \mathcal{F}(P \ a))\} \end{aligned}$$

⟨*proof*⟩

lemma *D-Mprefix*:

$$\begin{aligned} \mathcal{D} (\Box x \in A \rightarrow P \ x) & = \{d. d \neq [] \wedge hd \ d \in (ev \text{ ' } A) \wedge \\ & \quad (\exists a. ev \ a = hd \ d \wedge tl \ d \in \mathcal{D}(P \ a))\} \end{aligned}$$

⟨*proof*⟩

lemma *T-Mprefix*:

$$\begin{aligned} \mathcal{T} (\Box x \in A \rightarrow P \ x) & = \{s. s = [] \vee (\exists a. a \in A \wedge s \neq [] \wedge hd \ s = ev \ a \wedge tl \ s \in \mathcal{T}(P \\ & \ a))\} \end{aligned}$$

⟨*proof*⟩

4.3.3 Basic Properties

lemma *tick-T-Mprefix* [*simp*]: [tick] $\notin \mathcal{T} (\Box x \in A \rightarrow P \ x)$

⟨*proof*⟩

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

⟨*proof*⟩

4.3.4 Proof of Continuity Rule

Backpatch Isabelle 2009-1

definition

$contlub :: ('a::cpo \Rightarrow 'b::cpo) \Rightarrow bool$

where

$contlub\ f = (\forall Y. chain\ Y \longrightarrow f\ (\sqcup\ i. Y\ i) = (\sqcup\ i. f\ (Y\ i)))$

lemma *contlubE*:

$\llbracket contlub\ f; chain\ Y \rrbracket \Longrightarrow f\ (\sqcup\ i. Y\ i) = (\sqcup\ i. f\ (Y\ i))$

<proof>

lemma *monocontlub2cont*: $\llbracket monofun\ f; contlub\ f \rrbracket \Longrightarrow cont\ f$

<proof>

lemma *contlubI*:

$(\bigwedge Y. chain\ Y \Longrightarrow f\ (\sqcup\ i. Y\ i) = (\sqcup\ i. f\ (Y\ i))) \Longrightarrow contlub\ f$

<proof>

lemma *cont2contlub*: $cont\ f \Longrightarrow contlub\ f$

<proof>

Core of Proof

lemma *mono-Mprefix1*:

$\forall a \in A. P\ a \sqsubseteq Q\ a \Longrightarrow \mathcal{D}\ (Mprefix\ A\ Q) \subseteq \mathcal{D}\ (Mprefix\ A\ P)$

<proof>

lemma *mono-Mprefix2*:

$\forall x \in A. P\ x \sqsubseteq Q\ x \Longrightarrow$

$\forall s. s \notin \mathcal{D}\ (Mprefix\ A\ P) \longrightarrow Ra\ (Mprefix\ A\ P)\ s = Ra\ (Mprefix\ A\ Q)\ s$

<proof>

lemma *mono-Mprefix3* :

assumes $H: \forall x \in A. P\ x \sqsubseteq Q\ x$

shows $min\text{-}elems\ (\mathcal{D}\ (Mprefix\ A\ P)) \subseteq \mathcal{T}\ (Mprefix\ A\ Q)$

<proof>

lemma *mono-Mprefix0*:

$\forall x \in A. P\ x \sqsubseteq Q\ x \Longrightarrow Mprefix\ A\ P \sqsubseteq Mprefix\ A\ Q$

<proof>

lemma *mono-Mprefix[simp]*: $monofun(Mprefix\ A)$

<proof>

lemma *proc-ord2-set*:

$P \sqsubseteq Q \implies \{(s, X). s \notin \mathcal{D} P \wedge (s, X) \in \mathcal{F} P\} = \{(s, X). s \notin \mathcal{D} P \wedge (s, X) \in \mathcal{F} Q\}$
 $\langle proof \rangle$

lemma *proc-ord-proc-eq-spec*: $P \sqsubseteq Q \implies \mathcal{D} P \subseteq \mathcal{D} Q \implies P = Q$

$\langle proof \rangle$

lemma *Mprefix-chainpreserving*: $chain Y \implies chain (\lambda i. Mprefix A (Y i))$

$\langle proof \rangle$

lemma *limproc-is-thelub-fun*:

assumes $chain S$
shows $(Lub S c) = lim-proc (range (\lambda x. (S x c)))$
 $\langle proof \rangle$

lemma *contlub-Mprefix* : $contlub(\%P. Mprefix A P)$

$\langle proof \rangle$

lemma *Mprefix-cont [simp]*:

$(\bigwedge x. cont (f x)) \implies cont (\lambda y. \square z \in A \rightarrow f z y)$
 $\langle proof \rangle$

4.3.5 High-level Syntax for Read and Write

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

definition *read* $:: ['a \Rightarrow 'b, 'a \text{ set}, 'a \Rightarrow 'b \text{ process}] \Rightarrow 'b \text{ process}$

where $read\ c\ A\ P \equiv Mprefix(c\ 'A) (P\ o\ (inv-into\ A\ c))$

definition *write* $:: ['a \Rightarrow 'b, 'a, 'b \text{ process}] \Rightarrow 'b \text{ process}$

where $write\ c\ a\ P \equiv Mprefix\ \{c\ a\} (\lambda x. P)$

definition *write0* $:: ['a, 'a \text{ process}] \Rightarrow 'a \text{ process}$

where $write0\ a\ P \equiv Mprefix\ \{a\} (\lambda x. P)$

syntax

-read $:: [id, pptrn, 'a \text{ process}] \Rightarrow 'a \text{ process}$
 $((\exists(-?-) / \rightarrow -) [0,0,78] 78)$

-readX $:: [id, pptrn, bool, 'a \text{ process}] \Rightarrow 'a \text{ process}$
 $((\exists(-?-)|- / \rightarrow -) [0,0,78] 78)$

```

-readS :: [id, pptrn, 'b set, 'a process] => 'a process
          (( $\exists(-?) \in - / \rightarrow -$ ) [0,0,78] 78)
-write :: [id, 'b, 'a process] => 'a process
          (( $\exists(-!) / \rightarrow -$ ) [0,0,78] 78)
-writeS :: ['a, 'a process] => 'a process
          (( $\exists- / \rightarrow -$ ) [0,78] 78)

```

4.3.6 CSP_M-Style Syntax for Communication Primitives

translations

```

-read c p P     $\equiv$  CONST read c CONST UNIV ( $\lambda p. P$ )
-write c p P    $\equiv$  CONST write c p P
-readX c p b P  $\equiv$  CONST read c {p, b} ( $\lambda p. P$ )
-writeS a P     $\equiv$  CONST write0 a P
-readS c p A P  $\equiv$  CONST read c A ( $\lambda p. P$ )

```

Syntax Check:

```
term < d?y  $\rightarrow$  c!x  $\rightarrow$  P = Q >
```

```
lemma read-cont[simp]: cont P  $\implies$  cont ( $\lambda y. read c A (P y)$ )
  <proof>
```

```
lemma read-cont'[simp]: ( $\bigwedge x. cont (f x)$ )  $\implies$  cont ( $\lambda y. c?x \rightarrow f x y$ ) <proof>
```

```
lemma write-cont[simp]: cont (P::('b::cpo  $\Rightarrow$  'a process))  $\implies$  cont( $\lambda x. (c!a \rightarrow P x)$ )
  <proof>
```

```
corollary write0-cont-lub : contlub(Mprefix {a})
  <proof>
```

```
lemma write0-contlub : contlub(write0 a)
  <proof>
```

```
lemma write0-cont[simp]: cont (P::('b::cpo  $\Rightarrow$  'a process))  $\implies$  cont( $\lambda x. (a \rightarrow P x)$ )
  <proof>
```

end

theory Mndetprefix

imports Process Stop Mprefix Ndet

begin

4.4 Multiple non deterministic operator

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

syntax

$-Mndetprefix$:: $pttrn \Rightarrow 'a$ set $\Rightarrow 'a$ process $\Rightarrow 'a$ process
 $((\exists \Gamma \in - \rightarrow / -) [0, 0, \top] \top)$

translations

$\prod x \in A \rightarrow P \equiv CONST Mndetprefix A (\lambda x. P)$

lemma $mt-Mndetprefix[simp]$: $Mndetprefix \{\} P = STOP$
<proof>

lemma $rep-abs-Mndetprefix$: $A \neq \{\} \implies$
 $(Rep-process (Abs-process(\bigcup x \in A. \mathcal{F}(x \rightarrow P x), \bigcup x \in A. \mathcal{D}(x \rightarrow P x)))) =$
 $(\bigcup x \in A. \mathcal{F}(x \rightarrow P x), \bigcup x \in A. \mathcal{D}(x \rightarrow P x))$
<proof>

lemma $F-Mndetprefix$:

$\mathcal{F}(Mndetprefix A P) = (if A = \{\} then \{(s, X). s = []\} else \bigcup x \in A. \mathcal{F}(x \rightarrow P x))$
<proof>

lemma $D-Mndetprefix$: $\mathcal{D}(Mndetprefix A P) = (if A = \{\} then \{\} else \bigcup x \in A. \mathcal{D}(x \rightarrow P x))$
<proof>

lemma $T-Mndetprefix$: $\mathcal{T}(Mndetprefix A P) = (if A = \{\} then \{[]\} else \bigcup x \in A. \mathcal{T}(x \rightarrow P x))$
<proof>

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

lemma $Mndetprefix-unit$: $(\prod x \in \{a\} \rightarrow P x) = (a \rightarrow P a)$
<proof>

lemma $Mndetprefix-Un-distrib$: $A \neq \{\} \implies B \neq \{\} \implies (\prod x \in A \cup B \rightarrow P x) =$
 $((\prod x \in A \rightarrow P x) \sqcap (\prod x \in B \rightarrow P x))$
<proof>

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

finite case.

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

4.4.1 Finite case Continuity

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

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

4.4.2 General case Continuity

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

lemma *Mndetprefix-chainpreserving*: *chain Y* $\implies \text{chain } (\lambda i. (\sqcap z \in A \rightarrow Y \ i \ z))$
<proof>

lemma *contlub-Mndetprefix* : *contlub (Mndetprefix A)*
<proof>

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

end

Chapter 5

Advanced Induction Schemata

theory *CSP-Induct*

imports *HOLCF*

begin

default-sort *type*

5.1 k-fixpoint-induction

lemma *nat-k-induct* [*case-names base step*]:

fixes $k::nat$

assumes $\forall i < k. P\ i$ **and** $\forall n_0. (\forall i < k. P\ (n_0+i)) \longrightarrow P\ (n_0+k)$

shows $P\ (n::nat)$

<proof>

thm *fix-ind fix-ind2*

lemma *fix-ind-k* [*case-names admissibility base-k-steps step*]:

fixes $k::nat$

assumes *adm*: $adm\ P$

and *base-k-steps*: $\forall i < k. P\ (iterate\ i.f.\ \perp)$

and *step*: $\bigwedge x. (\forall i < k. P\ (iterate\ i.f.\ x)) \Longrightarrow P\ (iterate\ k.f.\ x)$

shows $P\ (fix.f)$

<proof>

lemma *nat-k-skip-induct* [*case-names lower-bound base-k step*]:

fixes $k::nat$

assumes $k \geq 1$ **and** $\forall i < k. P\ i$ **and** $\forall n_0. P\ (n_0) \longrightarrow P\ (n_0+k)$

shows $P\ (n::nat)$

<proof>

lemma *fix-ind-k-skip* [*case-names lower-bound admissibility base-k-steps step*]:

fixes *k::nat*
assumes *k-1: k ≥ 1*
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>

thm *parallel-fix-ind*

5.2 Parallel fixpoint-induction

lemma *parallel-fix-ind-inc*[*case-names admissibility base-fst base-snd step*]:

assumes *adm: adm (λx. 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>

end

Chapter 6

Refinements

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

definition *trace-refine* :: $\langle 'a \text{ process} \Rightarrow 'a \text{ process} \Rightarrow \text{bool} \rangle$ (**infix** $\langle \sqsubseteq_T \rangle$
60)

where $\langle P \sqsubseteq_T Q \equiv \mathcal{T} Q \subseteq \mathcal{T} P \rangle$

definition *failure-refine* :: $\langle 'a \text{ process} \Rightarrow 'a \text{ process} \Rightarrow \text{bool} \rangle$ (**infix** $\langle \sqsubseteq_F \rangle$
60)

where $\langle P \sqsubseteq_F Q \equiv \mathcal{F} Q \subseteq \mathcal{F} P \rangle$

definition *divergence-refine* :: $\langle 'a \text{ process} \Rightarrow 'a \text{ process} \Rightarrow \text{bool} \rangle$ (**infix** $\langle \sqsubseteq_D \rangle$
53)

where $\langle P \sqsubseteq_D Q \equiv \mathcal{D} Q \subseteq \mathcal{D} P \rangle$

definition *failure-divergence-refine* :: $\langle 'a \text{ process} \Rightarrow 'a \text{ process} \Rightarrow \text{bool} \rangle$ (**infix**
 $\langle \sqsubseteq_{FD} \rangle$ 60)

where $\langle P \sqsubseteq_{FD} Q \equiv P \leq Q \rangle$

definition *trace-divergence-refine* :: $\langle 'a \text{ process} \Rightarrow 'a \text{ process} \Rightarrow \text{bool} \rangle$ (**infix** $\langle \sqsubseteq_{DT} \rangle$
53)

where $\langle P \sqsubseteq_{DT} Q \equiv P \sqsubseteq_T Q \wedge P \sqsubseteq_D Q \rangle$

6.1 Idempotency

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

<proof>

6.2 Some obvious refinements

lemma *BOT-leF[simp]*: $\langle \perp \sqsubseteq_F Q \rangle$
and *BOT-leD[simp]*: $\langle \perp \sqsubseteq_D Q \rangle$
and *BOT-leT[simp]*: $\langle \perp \sqsubseteq_T Q \rangle$
<proof>

6.3 Antisymmetry

lemma *FD-antisym*: $\langle P \sqsubseteq_{FD} Q \implies Q \sqsubseteq_{FD} P \implies P = Q \rangle$
<proof>

lemma *DT-antisym*: $\langle P \sqsubseteq_{DT} Q \implies Q \sqsubseteq_{DT} P \implies P = Q \rangle$
<proof>

6.4 Transitivity

lemma *trans-F*: $\langle P \sqsubseteq_F Q \implies Q \sqsubseteq_F S \implies P \sqsubseteq_F S \rangle$
and *trans-D*: $\langle P \sqsubseteq_D Q \implies Q \sqsubseteq_D S \implies P \sqsubseteq_D S \rangle$
and *trans-T*: $\langle P \sqsubseteq_T Q \implies Q \sqsubseteq_T S \implies P \sqsubseteq_T S \rangle$
and *trans-FD*: $\langle P \sqsubseteq_{FD} Q \implies Q \sqsubseteq_{FD} S \implies P \sqsubseteq_{FD} S \rangle$
and *trans-DT*: $\langle P \sqsubseteq_{DT} Q \implies Q \sqsubseteq_{DT} S \implies P \sqsubseteq_{DT} S \rangle$
<proof>

6.5 Relations between refinements

lemma *leF-imp-leT*: $\langle P \sqsubseteq_F Q \implies P \sqsubseteq_T Q \rangle$
and *leFD-imp-leF*: $\langle P \sqsubseteq_{FD} Q \implies P \sqsubseteq_F Q \rangle$
and *leFD-imp-leD*: $\langle P \sqsubseteq_{FD} Q \implies P \sqsubseteq_D Q \rangle$
and *leDT-imp-leD*: $\langle P \sqsubseteq_{DT} Q \implies P \sqsubseteq_D Q \rangle$
and *leDT-imp-leT*: $\langle P \sqsubseteq_{DT} Q \implies P \sqsubseteq_T Q \rangle$
and *leF-leD-imp-leFD*: $\langle P \sqsubseteq_F Q \implies P \sqsubseteq_D Q \implies P \sqsubseteq_{FD} Q \rangle$
and *leD-leT-imp-leDT*: $\langle P \sqsubseteq_D Q \implies P \sqsubseteq_T Q \implies P \sqsubseteq_{DT} Q \rangle$
<proof>

6.6 More obvious refinements

lemma *BOT-leFD[simp]*: $\langle \perp \sqsubseteq_{FD} Q \rangle$
and *BOT-leDT[simp]*: $\langle \perp \sqsubseteq_{DT} Q \rangle$
<proof>

lemma *leDT-STOP[simp]*: $\langle P \sqsubseteq_{DT} STOP \rangle$
<proof>

lemma $leD\text{-STOP}[simp]$: $\langle P \sqsubseteq_D STOP \rangle$
and $leT\text{-STOP}[simp]$: $\langle P \sqsubseteq_T STOP \rangle$
 $\langle proof \rangle$

6.7 Admissibility

lemma $le\text{-F}\text{-adm}[simp]$: $\langle cont (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x. u\ x \sqsubseteq_F v\ x) \rangle$
 $\langle proof \rangle$

lemma $le\text{-T}\text{-adm}[simp]$: $\langle cont (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x. u\ x \sqsubseteq_T v\ x) \rangle$
 $\langle proof \rangle$

lemma $le\text{-D}\text{-adm}[simp]$: $\langle cont (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x. u\ x \sqsubseteq_D v\ x) \rangle$
 $\langle proof \rangle$

lemmas $le\text{-FD}\text{-adm}[simp] = le\text{-adm}[folded\ failure\text{-divergence}\text{-refine}\text{-def}]$

lemma $le\text{-DT}\text{-adm}[simp]$: $\langle cont (u::('a::cpo) \Rightarrow 'b\ process) \Longrightarrow monofun\ v \Longrightarrow adm(\lambda x. u\ x \sqsubseteq_{DT} v\ x) \rangle$
 $\langle proof \rangle$

end

Chapter 7

The "Laws" of CSP

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

method exI for y::'a = (rule-tac exI[where x = y])
```

7.1 General Laws

```
lemma SKIP-Neq-STOP: SKIP  $\neq$  STOP
  <proof>
```

```
lemma BOT-less1[simp]:  $\perp \leq (X::'a \text{ process})$ 
  <proof>
```

```
lemma BOT-less2[simp]: BOT  $\leq (X::'a \text{ process})$ 
  <proof>
```

7.2 Deterministic Choice Operator Laws

```
lemma mono-Det-FD-onside[simp]:  $P \leq P' \implies (P \square S) \leq (P' \square S)$ 
  <proof>
```

```
lemma mono-Det-FD[simp]:  $\llbracket P \leq P'; S \leq S' \rrbracket \implies (P \square S) \leq (P' \square S')$ 
  <proof>
```

```
lemma mono-Det-ref:  $\llbracket P \sqsubseteq P'; S \sqsubseteq S' \rrbracket \implies (P \square S) \sqsubseteq (P' \square S')$ 
  <proof>
```

```
lemma Det-BOT :  $(P \square \perp) = \perp$ 
  <proof>
```

```
lemma Det-STOP:  $(P \square STOP) = P$ 
```

<proof>

lemma *Det-id*: $(P \sqcap P) = P$

<proof>

lemma *Det-assoc*: $((M \sqcap P) \sqcap Q) = (M \sqcap (P \sqcap Q))$

<proof>

7.3 NonDeterministic Choice Operator Laws

lemma *mono-Ndet-FD[simp]*: $\llbracket P \leq P'; S \leq S' \rrbracket \implies (P \sqcap S) \leq (P' \sqcap S')$

<proof>

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

<proof>

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

<proof>

lemma *mono-Ndet-ref*: $\llbracket P \sqsubseteq P'; S \sqsubseteq S' \rrbracket \implies (P \sqcap S) \sqsubseteq (P' \sqcap S')$

<proof>

lemma *Ndet-BOT*: $(P \sqcap \perp) = \perp$

<proof>

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

<proof>

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

<proof>

7.3.1 Multi-Operators laws

lemma *Det-distrib*: $(M \sqcap (P \sqcap Q)) = ((M \sqcap P) \sqcap (M \sqcap Q))$

<proof>

lemma *Ndet-distrib*: $(M \sqcap (P \sqcap Q)) = ((M \sqcap P) \sqcap (M \sqcap Q))$

<proof>

lemma *Ndet-FD-Det*: $(P \sqcap Q) \leq (P \sqcap Q)$

<proof>

7.4 Sequence Operator Laws

7.4.1 Preliminaries

definition *F-minus-D-Seq* where

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

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

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

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

7.4.2 Laws

lemma *mono-Seq-FD[simp]*: $\llbracket P \leq P'; S \leq S' \rrbracket \implies (P; S) \leq (P'; S')$
 $\langle proof \rangle$

lemma *mono-Seq-ref*: $\llbracket P \sqsubseteq P'; S \sqsubseteq S' \rrbracket \implies (P; S) \sqsubseteq (P'; S')$
 $\langle proof \rangle$

lemma *BOT-Seq*: $(\perp; P) = \perp$
 $\langle proof \rangle$

lemma *Seq-SKIP*: $(P; SKIP) = P$
 $\langle proof \rangle$

lemma *SKIP-Seq*: $(SKIP; P) = P$
 $\langle proof \rangle$

lemma *STOP-Seq*: $(STOP; P) = STOP$
 $\langle proof \rangle$

7.4.3 Multi-Operators laws

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

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

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

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

7.5 The Multi-Prefix Operator Laws

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

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

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

7.5.1 Multi-Operators laws

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

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

7.5.2 Derivative Operators laws

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

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

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

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

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

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

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

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

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

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

<proof>

7.6 The Hiding Operator Laws

7.6.1 Preliminaries

lemma *elemDIselemHD*: $\langle t \in \mathcal{D} P \implies \text{trace-hide } t (ev \text{ ' } A) \in \mathcal{D} (P \setminus A) \rangle$

<proof>

lemma *length-strict-mono*: $\langle \text{strict-mono } (f::nat \Rightarrow \text{'a list}) \implies \text{length } (f \ i) \geq i + \text{length } (f \ 0) \rangle$

<proof>

lemma *mono-trace-hide*: $\langle a \leq b \implies \text{trace-hide } a (ev \text{ ' } A) \leq \text{trace-hide } b (ev \text{ ' } A) \rangle$

<proof>

lemma *mono-constant*:

assumes *mono* $(f::nat \Rightarrow \text{'a event list})$ **and** $\forall i. f \ i \leq a$

shows $\exists i. \forall j \geq i. f \ j = f \ i$

<proof>

lemma *elemTiselemHT*: $\langle t \in \mathcal{T} P \implies \text{trace-hide } t (ev \text{ ' } A) \in \mathcal{T} (P \setminus A) \rangle$

<proof>

lemma *Hiding-Un-D1*: $\langle \mathcal{D} (P \setminus (A \cup B)) \subseteq \mathcal{D} ((P \setminus A) \setminus B) \rangle$

<proof>

lemma *Hiding-Un-D2*: $\langle \text{finite } A \implies \mathcal{D} ((P \setminus A) \setminus B) \subseteq \mathcal{D} (P \setminus (A \cup B)) \rangle$

<proof>

lemma *Hiding-Un-D*: $\langle \text{finite } A \implies \mathcal{D} ((P \setminus A) \setminus B) = \mathcal{D} (P \setminus (A \cup B)) \rangle$

<proof>

7.6.2 Laws

lemma *mono-Hiding-FD[simp]*: $\langle P \leq Q \implies P \setminus A \leq Q \setminus A \rangle$

<proof>

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

lemma *Hiding-Un*: $\langle \text{finite } A \implies P \setminus (A \cup B) = (P \setminus A) \setminus B \rangle$

<proof>

lemma *Hiding-set-BOT*: $\langle (\perp \setminus A) = \perp \rangle$

<proof>

lemma *Hiding-set-STOP*: $\langle (STOP \setminus A) = STOP \rangle$

<proof>

lemma *Hiding-set-SKIP*: $\langle (SKIP \setminus A) = SKIP \rangle$
 $\langle proof \rangle$

lemma *Hiding-set-empty*: $\langle (P \setminus \{\}) = P \rangle$
 $\langle proof \rangle$

7.6.3 Multi-Operators laws

lemma *Hiding-Ndet*: $\langle (P \sqcap Q) \setminus A = ((P \setminus A) \sqcap (Q \setminus A)) \rangle$
 $\langle proof \rangle$

lemma *Hiding-Mprefix-distr*:
 $\langle (B \sqcap A) = \{\} \implies ((Mprefix A P) \setminus B) = (Mprefix A (\lambda x. ((P x) \setminus B))) \rangle$
 $\langle proof \rangle$

lemma *no-Hiding-read*: $\langle (\forall y. c y \notin B) \implies ((c?x \rightarrow (P x)) \setminus B) = (c?x \rightarrow ((P x) \setminus B)) \rangle$
 $\langle proof \rangle$

lemma *no-Hiding-write0*: $\langle a \notin B \implies ((a \rightarrow P) \setminus B) = (a \rightarrow (P \setminus B)) \rangle$
 $\langle proof \rangle$

lemma *Hiding-write0*: $\langle a \in B \implies ((a \rightarrow P) \setminus B) = (P \setminus B) \rangle$
 $\langle proof \rangle$

lemma *no-Hiding-write*: $\langle (\forall y. c y \notin B) \implies ((c!a \rightarrow P) \setminus B) = (c!a \rightarrow (P \setminus B)) \rangle$
 $\langle proof \rangle$

lemma *Hiding-write*: $\langle (c a) \in B \implies ((c!a \rightarrow P) \setminus B) = (P \setminus B) \rangle$
 $\langle proof \rangle$

7.7 The Sync Operator Laws

7.7.1 Preliminaries

lemma *SyncTLEmpty*: $a \text{ setinterleaves } (([], u), A) \implies tl a \text{ setinterleaves } (([], tl u), A)$
 $\langle proof \rangle$

lemma *SyncHd-Tl*:
 $a \text{ setinterleaves } ((t, u), A) \wedge hd t \in A \wedge hd u \notin A$
 $\implies hd a = hd u \wedge tl a \text{ setinterleaves } ((t, tl u), A)$
 $\langle proof \rangle$

lemma *SyncHdAddEmpty*:
 $(tl a) \text{ setinterleaves } (([], u), A) \wedge hd a \notin A \wedge a \neq []$
 $\implies a \text{ setinterleaves } (([], hd a \# u), A)$
 $\langle proof \rangle$

lemma *SyncHdAdd*:

$$\begin{aligned} & (tl\ a)\ setinterleaves\ ((t,\ u),\ A) \wedge hd\ a \notin A \wedge hd\ t \in A \wedge a \neq [] \\ & \implies a\ setinterleaves\ ((t,\ hd\ a\ \# \ u),\ A) \\ & \langle proof \rangle \end{aligned}$$

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

lemma *SyncSameHdTl*:

$$\begin{aligned} & a\ setinterleaves\ ((t,\ u),\ A) \wedge hd\ t \in A \wedge hd\ u \in A \\ & \implies hd\ t = hd\ u \wedge hd\ a = hd\ t \wedge (tl\ a)\ setinterleaves\ ((tl\ t,\ tl\ u),\ A) \\ & \langle proof \rangle \end{aligned}$$

lemma *SyncSingleHeadAdd*:

$$\begin{aligned} & a\ setinterleaves\ ((t,\ u),\ A) \wedge h \notin A \\ & \implies (h\ \# \ a)\ setinterleaves\ ((h\ \# \ t,\ u),\ A) \\ & \langle proof \rangle \end{aligned}$$

lemma *TickLeftSync*:

$$\begin{aligned} & tick \in A \wedge front\text{-}tickFree\ t \wedge s\ setinterleaves\ (([tick],\ t),\ A) \implies s = t \wedge last\ t \\ & = tick \\ & \langle proof \rangle \end{aligned}$$

lemma *EmptyLeftSync*: $s\ setinterleaves\ (([],\ t),\ A) \implies s = t \wedge set\ t \cap A = \{\}$
 $\langle proof \rangle$

lemma *tick-T-F*: $t@[tick] \in \mathcal{T}\ P \implies (t@[tick],\ X) \in \mathcal{F}\ P$
 $\langle proof \rangle$

lemma *event-set*: $(e::'a\ event) \in insert\ tick\ (range\ ev)$
 $\langle proof \rangle$

lemma *Mprefix-Sync-distr1-D*:

$$\begin{aligned} & A \subseteq S \\ & \implies B \subseteq S \\ & \implies \mathcal{D}\ (Mprefix\ A\ P\ \llbracket S \rrbracket\ Mprefix\ B\ Q) = \mathcal{D}\ (\sqcap\ x \in A \cap B \rightarrow (P\ x\ \llbracket S \rrbracket\ Q\ x)) \end{aligned}$$

$\langle proof \rangle$

lemma *Hiding-interleave*:

$$\begin{aligned} & A \cap C = \{\} \\ & \implies r\ setinterleaves\ ((t,\ u),\ C) \\ & \implies (trace\text{-}hide\ r\ A)\ setinterleaves\ ((trace\text{-}hide\ t\ A,\ trace\text{-}hide\ u\ A),\ C) \\ & \langle proof \rangle \end{aligned}$$

lemma *non-Sync-interleaving*:

$$\begin{aligned} & (set\ t \cup set\ u) \cap C = \{\} \implies setinterleaving\ (t,\ C,\ u) \neq \{\} \\ & \langle proof \rangle \end{aligned}$$

lemma *interleave-Hiding*:

$$\begin{aligned}
& A \cap C = \{\} \\
& \implies ra \text{ setinterleaves } ((\text{trace-hide } t \ A, \text{ trace-hide } u \ A), C) \\
& \implies \exists r. ra = \text{trace-hide } r \ A \wedge r \text{ setinterleaves } ((t, u), C) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-size*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t, u), C) \implies \text{length } s = \text{length } t + \text{length } u - \text{length}(\text{filter } (\lambda x. \\
& x \in C) \ t) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-eq-size*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t, u), C) \implies s' \text{ setinterleaves } ((t, u), C) \implies \text{length } s = \text{length } s' \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-set*: $s \text{ setinterleaves } ((t, u), C) \implies \text{set } t \cup \text{set } u \subseteq \text{set } s$
 $\langle \text{proof} \rangle$

lemma *interleave-order*: $s \text{ setinterleaves } ((t1@t2, u), C) \implies \text{set}(t2) \subseteq \text{set}(\text{drop}(\text{length } t1) \ s)$
 $\langle \text{proof} \rangle$

lemma *interleave-append*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t1@t2, u), C) \\
& \implies \exists u1 \ u2 \ s1 \ s2. u = u1@u2 \wedge s = s1@s2 \wedge \\
& \qquad \qquad \qquad s1 \text{ setinterleaves } ((t1, u1), C) \wedge s2 \text{ setinterleaves } ((t2, u2), C) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-append-sym*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t, u1@u2), C) \\
& \implies \exists t1 \ t2 \ s1 \ s2. t = t1@t2 \wedge s = s1@s2 \wedge \\
& \qquad \qquad \qquad s1 \text{ setinterleaves } ((t1, u1), C) \wedge s2 \text{ setinterleaves } ((t2, u2), C) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-append-tail*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t1, u), C) \implies (\text{set } t2) \cap C = \{\} \implies (s@t2) \text{ setinterleaves } \\
& ((t1@t2, u), C) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-append-tail-sym*:

$$\begin{aligned}
& s \text{ setinterleaves } ((t, u1), C) \implies (\text{set } u2) \cap C = \{\} \implies (s@u2) \text{ setinterleaves } \\
& ((t, u1@u2), C) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-assoc-1*:

$$\begin{aligned}
& tu \text{ setinterleaves } ((t, u), A) \\
& \implies twv \text{ setinterleaves } ((tu, v), A) \\
& \implies \exists uv. uv \text{ setinterleaves } ((u, v), A) \wedge twv \text{ setinterleaves } ((t, uv), A) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *interleave-assoc-2*:

assumes $*:uv$ *setinterleaves* $((u, v), A)$ **and**

$** :tuv$ *setinterleaves* $((t, uv), A)$

shows $\exists tu.$ *tu setinterleaves* $((t, u), A) \wedge tuv$ *setinterleaves* $((tu, v), A)$

\langle *proof* \rangle

7.7.2 Laws

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

\langle *proof* \rangle

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

\langle *proof* \rangle

lemma *mono-Sync-FD[simp]*: $\llbracket P \leq P'; Q \leq Q' \rrbracket \implies (P \llbracket S \rrbracket Q) \leq (P' \llbracket S \rrbracket Q')$

\langle *proof* \rangle

lemma *mono-Sync-ref*: $\llbracket P \sqsubseteq P'; Q \sqsubseteq Q' \rrbracket \implies (P \llbracket S \rrbracket Q) \sqsubseteq (P' \llbracket S \rrbracket Q')$

\langle *proof* \rangle

lemma *Sync-BOT*: $(P \llbracket S \rrbracket \perp) = \perp$

\langle *proof* \rangle

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

\langle *proof* \rangle

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

\langle *proof* \rangle

7.7.3 Multi-Operators laws

lemma *Mprefix-Sync-distr*:

$$\langle (\Box x \in A \cup A' \rightarrow P x) \llbracket S \rrbracket (\Box y \in B \cup B' \rightarrow Q y) =$$

$$(\Box x \in A \rightarrow (P x \llbracket S \rrbracket (\Box y \in B \cup B' \rightarrow Q y))) \Box$$

$$(\Box y \in B \rightarrow ((\Box x \in A \cup A' \rightarrow P x) \llbracket S \rrbracket Q y)) \Box$$

$$(\Box x \in A' \cap B' \rightarrow (P x \llbracket S \rrbracket Q x)) \rangle$$

(**is** $\langle ?lhs A A' P B B' Q = ?rhs A A' P B B' Q \rangle$)

if sets-assms: $\langle A \cap S = \{\} \rangle \langle A' \subseteq S \rangle \langle B \cap S = \{\} \rangle \langle B' \subseteq S \rangle$

\langle *proof* \rangle

lemma *Mprefix-Sync-distr-bis*:

$$\langle (Mprefix A P) \llbracket S \rrbracket (Mprefix B Q) =$$

$$(\Box x \in A - S \rightarrow (P x \llbracket S \rrbracket Mprefix B Q))$$

$$\Box (\Box y \in B - S \rightarrow (Mprefix A P \llbracket S \rrbracket Q y))$$

$$\Box (\Box x \in A \cap B \cap S \rightarrow (P x \llbracket S \rrbracket Q x)) \rangle$$

\langle *proof* \rangle

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

lemma *Mprefix-Sync-SKIP*: $(Mprefix\ B\ P\ \llbracket S \rrbracket\ SKIP) = (\Box\ x \in\ B - S \rightarrow (P\ x\ \llbracket S \rrbracket\ SKIP))$
<proof>

lemma *Sync-Ndet-left-distrib*: $((P \sqcap Q) \llbracket S \rrbracket M) = ((P \llbracket S \rrbracket M) \sqcap (Q \llbracket S \rrbracket M))$
<proof>

lemma *Sync-Ndet-right-distrib*: $(M \llbracket S \rrbracket (P \sqcap Q)) = ((M \llbracket S \rrbracket P) \sqcap (M \llbracket S \rrbracket Q))$
<proof>

lemma *prefix-Sync-SKIP1*: $a \notin S \implies (a \rightarrow P \llbracket S \rrbracket SKIP) = (a \rightarrow (P \llbracket S \rrbracket SKIP))$
<proof>

lemma *prefix-Sync-SKIP2*: $a \in S \implies (a \rightarrow P \llbracket S \rrbracket SKIP) = STOP$
<proof>

lemma *prefix-Sync-SKIP*: $(a \rightarrow P \llbracket S \rrbracket SKIP) = (\text{if } a \in S \text{ then } STOP \text{ else } (a \rightarrow (P \llbracket S \rrbracket SKIP)))$
<proof>

lemma *prefix-Sync1*: $\llbracket a \in S; b \in S; a \neq b \rrbracket \implies (a \rightarrow P \llbracket S \rrbracket (b \rightarrow Q)) = STOP$
<proof>

lemma *prefix-Sync2*: $a \in S \implies ((a \rightarrow P) \llbracket S \rrbracket (a \rightarrow Q)) = (a \rightarrow (P \llbracket S \rrbracket Q))$
<proof>

lemma *prefix-Sync3*: $\llbracket a \in S; b \notin S \rrbracket \implies (a \rightarrow P \llbracket S \rrbracket (b \rightarrow Q)) = (b \rightarrow ((a \rightarrow P) \llbracket S \rrbracket Q))$
<proof>

lemma *Hiding-Sync-D1*: $\langle \text{finite } A \implies A \cap S = \{\} \implies \mathcal{D}((P \llbracket S \rrbracket Q) \setminus A) \subseteq \mathcal{D}((P \setminus A) \llbracket S \rrbracket (Q \setminus A)) \rangle$
<proof>

lemma *Hiding-Sync-D2*:

assumes $*:A \cap S = \{\}$

shows $\langle \mathcal{D} ((P \setminus A) \llbracket S \rrbracket (Q \setminus A)) \subseteq \mathcal{D} ((P \llbracket S \rrbracket Q) \setminus A) \rangle$

\langle proof \rangle

lemma *Hiding-Sync*:

\langle finite A \implies A \cap S = \{\} \implies ((P \llbracket S \rrbracket Q) \setminus A) = ((P \setminus A) \llbracket S \rrbracket (Q \setminus A)) \rangle

\langle proof \rangle

lemma *Sync-assoc-oneside-D*: $\mathcal{D} (P \llbracket S \rrbracket (Q \llbracket S \rrbracket T)) \subseteq \mathcal{D} (P \llbracket S \rrbracket Q \llbracket S \rrbracket T)$

\langle proof \rangle

lemma *Sync-assoc-oneside*: $((P \llbracket S \rrbracket Q) \llbracket S \rrbracket T) \leq (P \llbracket S \rrbracket (Q \llbracket S \rrbracket T))$

\langle proof \rangle

lemma *Sync-assoc*: $((P \llbracket S \rrbracket Q) \llbracket S \rrbracket T) = (P \llbracket S \rrbracket (Q \llbracket S \rrbracket T))$

\langle proof \rangle

7.7.4 Derivative Operators laws

lemmas *Par-BOT* = *Sync-BOT*[**where** $S = \langle UNIV \rangle$]

and *Par-SKIP-SKIP* = *Sync-SKIP-SKIP*[**where** $S = \langle UNIV \rangle$]

and *Par-SKIP-STOP* = *Sync-SKIP-STOP*[**where** $S = \langle UNIV \rangle$]

and *Par-commute* = *Sync-commute*[**where** $S = \langle UNIV \rangle$]

and *Par-assoc* = *Sync-assoc*[**where** $S = \langle UNIV \rangle$]

and *Par-Ndet-right-distrib* = *Sync-Ndet-right-distrib*[**where** $S = \langle UNIV \rangle$]

and *Par-Ndet-left-distrib* = *Sync-Ndet-left-distrib*[**where** $S = \langle UNIV \rangle$]

and *Mprefix-Par-SKIP* = *Mprefix-Sync-SKIP*[**where** $S = \langle UNIV \rangle$]

and *prefix-Par-SKIP* = *prefix-Sync-SKIP*[**where** $S = \langle UNIV \rangle$, *simplified*]

and *mono-Par-FD* = *mono-Sync-FD*[**where** $S = \langle UNIV \rangle$]

and *mono-Par-ref* = *mono-Sync-ref*[**where** $S = \langle UNIV \rangle$]

and *prefix-Par1* = *prefix-Sync1*[**where** $S = \langle UNIV \rangle$, *simplified*]

and *prefix-Par2* = *prefix-Sync2*[**where** $S = \langle UNIV \rangle$, *simplified*]

and *Mprefix-Par-distr* = *Mprefix-Sync-distr-subset*[**where** $S = \langle UNIV \rangle$, *simplified*]

and *Inter-BOT* = *Sync-BOT*[**where** $S = \langle \{\} \rangle$]

and *Inter-SKIP-SKIP* = *Sync-SKIP-SKIP*[**where** $S = \langle \{\} \rangle$]

and *Inter-SKIP-STOP* = *Sync-SKIP-STOP*[**where** $S = \langle \{\} \rangle$]

and *Inter-commute* = *Sync-commute*[**where** $S = \langle \{\} \rangle$]

and *Inter-assoc* = *Sync-assoc*[**where** $S = \langle \{\} \rangle$]

and *Inter-Ndet-right-distrib* = *Sync-Ndet-right-distrib*[**where** $S = \langle \{\} \rangle$]

and *Inter-Ndet-left-distrib* = *Sync-Ndet-left-distrib*[**where** $S = \langle \{\} \rangle$]

and *Mprefix-Inter-SKIP* = *Mprefix-Sync-SKIP*[**where** $S = \langle \{\} \rangle$, *simplified*, *simplified Mprefix-STOP*]

and *prefix-Inter-SKIP* = *prefix-Sync-SKIP*[**where** $S = \langle \{\} \rangle$, *simplified*]

and *mono-Inter-FD* = *mono-Sync-FD*[**where** $S = \langle \{\} \rangle$]

and *mono-Inter-ref* = *mono-Sync-ref*[**where** $S = \langle \{\} \rangle$]

and *Hiding-Inter* = *Hiding-Sync*[**where** $S = \langle \{\} \rangle$, *simplified*]

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

lemma *Inter-SKIP*: $(P \parallel \text{SKIP}) = P$
 ⟨proof⟩

lemma *Inter-STOP-Seq-STOP*: $(P \parallel \text{STOP}) = (P ; \text{STOP})$
 ⟨proof⟩

lemma *Par-STOP*: $P \neq \perp \implies (P \parallel \text{STOP}) = \text{STOP}$
 ⟨proof⟩

7.8 Multiple Non Deterministic Operator Laws

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

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

lemma *Mndetprefix-subset-FD*: $A \neq \{\} \implies (\sqcap x a \in A \cup B \rightarrow P) \leq (\sqcap x a \in A \rightarrow P)$
 ⟨proof⟩

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

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

7.9 Infra-structure for Communication Primitives

lemma *read-read-Sync*:

assumes *contained*: $(\bigwedge y. c y \in S)$

shows $((c?x \rightarrow P x) \llbracket S \rrbracket (c?x \rightarrow Q x)) = (c?x \rightarrow ((P x) \llbracket S \rrbracket (Q x)))$

⟨proof⟩

lemma *read-read-non-Sync*:

$(\bigwedge y. c y \notin S; \bigwedge y. d y \in S) \implies (c?x \rightarrow (P x)) \llbracket S \rrbracket (d?x \rightarrow (Q x)) = c?x \rightarrow ((P x) \llbracket S \rrbracket (d?x \rightarrow (Q x)))$

⟨proof⟩

lemma *write-read-Sync*:

assumes *contained*: $\bigwedge y. c\ y \in S$

and *is-construct*: *inj* c

shows $(c!a \rightarrow P) \llbracket S \rrbracket (c?x \rightarrow Q\ x) = (c!a \rightarrow (P \llbracket S \rrbracket Q\ a))$

<proof>

lemma *write-read-non-Sync*:

$\llbracket d\ a \notin S; \bigwedge y. c\ y \in S \rrbracket \implies (d!a \rightarrow P) \llbracket S \rrbracket (c?x \rightarrow Q\ x) = d!a \rightarrow (P \llbracket S \rrbracket (c?x \rightarrow Q\ x))$

<proof>

lemma *write0-read-non-Sync*:

$\llbracket d \in S; \bigwedge y. c\ y \notin S \rrbracket \implies (d \rightarrow P) \llbracket S \rrbracket (c?x \rightarrow Q\ x) = c?x \rightarrow ((d \rightarrow P) \llbracket S \rrbracket Q\ x)$

<proof>

lemma *write0-write-non-Sync*:

$\llbracket d\ a \notin C; c \in C \rrbracket \implies (c \rightarrow Q) \llbracket C \rrbracket (d!a \rightarrow P) = d!a \rightarrow ((c \rightarrow Q) \llbracket C \rrbracket P)$

<proof>

7.10 Renaming operator laws

lemma *Renaming-BOT*: $\langle \text{Renaming } \perp\ f = \perp \rangle$

<proof>

lemma *Renaming-STOP*: $\langle \text{Renaming } \text{STOP}\ f = \text{STOP} \rangle$

<proof>

lemma *Renaming-SKIP*: $\langle \text{Renaming } \text{SKIP}\ f = \text{SKIP} \rangle$

<proof>

lemma *Renaming-Ndet*: $\langle \text{Renaming } (P \sqcap Q)\ f = \text{Renaming } P\ f \sqcap \text{Renaming } Q\ f \rangle$

<proof>

lemma *Renaming-Det*: $\langle \text{Renaming } (P \sqcup Q)\ f = \text{Renaming } P\ f \sqcup \text{Renaming } Q\ f \rangle$

<proof>

lemma *mono-Mprefix-eq*: $\langle \forall a \in A. P a = Q a \implies Mprefix A P = Mprefix A Q \rangle$
<proof>

lemma *mono-Mndetprefix-eq*: $\langle \forall a \in A. P a = Q a \implies Mndetprefix A P = Mndetprefix A Q \rangle$
<proof>

lemma *Renaming-Mprefix*: $\langle Renaming (\sqcap a \in A \rightarrow P (f a)) f = \sqcap b \in f ' A \rightarrow Renaming (P b) f \rangle$
<proof>

lemma *Renaming-Mprefix-inj-on*:
 $\langle Renaming (Mprefix A P) f = \sqcap b \in f ' A \rightarrow Renaming (P (THE a. a \in A \wedge f a = b)) f \rangle$
if *inj-on-f*: $\langle inj-on f A \rangle$
<proof>

corollary *Renaming-Mprefix-inj*:
 $\langle Renaming (Mprefix A P) f = \sqcap b \in f ' A \rightarrow Renaming (P (THE a. f a = b)) f \rangle$
if *inj-f*: $\langle inj f \rangle$
<proof>

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

corollary *Renaming-prefix*: $\langle Renaming (a \rightarrow P) f = (f a \rightarrow Renaming P f) \rangle$
<proof>

lemma *Renaming-Mndetprefix*: $\langle Renaming (\sqcap a \in A \rightarrow P (f a)) f = \sqcap b \in f ' A \rightarrow Renaming (P b) f \rangle$
<proof>

corollary *Renaming-Mndetprefix-inj-on*:
 $\langle Renaming (Mndetprefix A P) f = \sqcap b \in f ' A \rightarrow Renaming (P (THE a. a \in A \wedge f a = b)) f \rangle$
if *inj-on-f*: $\langle inj-on f A \rangle$
<proof>

corollary *Renaming-Mndetprefix-inj:*

$\langle \text{Renaming } (\text{Mndetprefix } A \ P) \ f = \sqcap b \in f \ ' \ A \rightarrow \text{Renaming } (P \ (\text{THE } a. \ f \ a = b)) \ f \rangle$
if *inj-f:* $\langle \text{inj } f \rangle$
 $\langle \text{proof} \rangle$

lemma *Renaming-Seq:* $\langle \text{Renaming } (P ; Q) \ f = \text{Renaming } P \ f ; \text{Renaming } Q \ f \rangle$
 $\langle \text{proof} \rangle$

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

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

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

end

Chapter 8

The refinements laws

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

8.1 Monotonicity

lemma *mono-Det-D[simp]*: $\langle \llbracket P \sqsubseteq_D P'; S \sqsubseteq_D S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_D (P' \sqcap S') \rangle$
<proof>

lemma *mono-Det-T[simp]*: $\langle \llbracket P \sqsubseteq_T P'; S \sqsubseteq_T S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_T (P' \sqcap S') \rangle$
<proof>

corollary *mono-Det-DT[simp]*: $\langle \llbracket P \sqsubseteq_{DT} P'; S \sqsubseteq_{DT} S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} (P' \sqcap S') \rangle$
<proof>

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

— Deterministic choice monotony doesn't hold for \sqsubseteq_F

lemma *mono-Ndet-F-left[simp]*: $\langle P \sqsubseteq_F Q \Longrightarrow (P \sqcap S) \sqsubseteq_F Q \rangle$
<proof>

lemma *mono-Ndet-D-left[simp]*: $\langle P \sqsubseteq_D Q \Longrightarrow (P \sqcap S) \sqsubseteq_D Q \rangle$
<proof>

lemma *mono-Ndet-T-left[simp]*: $\langle P \sqsubseteq_T Q \Longrightarrow (P \sqcap S) \sqsubseteq_T Q \rangle$
<proof>

8.2 Monotonicity

lemma *mono-Ndet-F[simp]*: $\langle \llbracket P \sqsubseteq_F P'; S \sqsubseteq_F S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_F (P' \sqcap S') \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-Ndet-D[simp]*: $\langle \llbracket P \sqsubseteq_D P'; S \sqsubseteq_D S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_D (P' \sqcap S') \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-Ndet-T[simp]*: $\langle \llbracket P \sqsubseteq_T P'; S \sqsubseteq_T S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_T (P' \sqcap S') \rangle$
 $\langle \text{proof} \rangle$

corollary *mono-Ndet-DT[simp]*: $\langle \llbracket P \sqsubseteq_{DT} P'; S \sqsubseteq_{DT} S' \rrbracket \Longrightarrow (P \sqcap S) \sqsubseteq_{DT} (P' \sqcap S') \rangle$
 $\langle \text{proof} \rangle$

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

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

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

lemma *mono-Ndet-Det-FD[simp]*: $\langle (P \sqcap S) \sqsubseteq_{FD} (P \sqcap S) \rangle$
 $\langle \text{proof} \rangle$

corollary *mono-Ndet-Det-F[simp]*: $\langle (P \sqcap S) \sqsubseteq_F (P \sqcap S) \rangle$
and *mono-Ndet-Det-D[simp]*: $\langle (P \sqcap S) \sqsubseteq_D (P \sqcap S) \rangle$
and *mono-Ndet-Det-T[simp]*: $\langle (P \sqcap S) \sqsubseteq_T (P \sqcap S) \rangle$
and *mono-Ndet-Det-DT[simp]*: $\langle (P \sqcap S) \sqsubseteq_{DT} (P \sqcap S) \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-Seq-F-right[simp]*: $\langle S \sqsubseteq_F S' \Longrightarrow (P ; S) \sqsubseteq_F (P ; S') \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-Seq-D-right[simp]*: $\langle S \sqsubseteq_D S' \Longrightarrow (P ; S) \sqsubseteq_D (P ; S') \rangle$
 $\langle \text{proof} \rangle$

lemma *mono-Seq-T-right[simp]*: $\langle S \sqsubseteq_T S' \Longrightarrow (P ; S) \sqsubseteq_T (P ; S') \rangle$
 $\langle \text{proof} \rangle$

corollary *mono-Seq-DT-right[simp]*: $\langle S \sqsubseteq_{DT} S' \implies (P ; S) \sqsubseteq_{DT} (P ; S') \rangle$
<proof>

lemma *mono-Seq-DT-left[simp]*: $\langle P \sqsubseteq_{DT} P' \implies (P ; S) \sqsubseteq_{DT} (P' ; S) \rangle$
<proof>

corollary *mono-Seq-DT[simp]*: $\langle P \sqsubseteq_{DT} P' \implies S \sqsubseteq_{DT} S' \implies (P ; S) \sqsubseteq_{DT} (P' ; S') \rangle$
<proof>

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

lemma *mono-Mprefix-F[simp]*: $\langle \forall x \in A. P x \sqsubseteq_F Q x \implies Mprefix A P \sqsubseteq_F Mprefix A Q \rangle$
<proof>

lemma *mono-Mprefix-D[simp]*: $\langle \forall x \in A. P x \sqsubseteq_D Q x \implies Mprefix A P \sqsubseteq_D Mprefix A Q \rangle$
<proof>

lemma *mono-Mprefix-T[simp]*: $\langle \forall x \in A. P x \sqsubseteq_T Q x \implies Mprefix A P \sqsubseteq_T Mprefix A Q \rangle$
<proof>

corollary *mono-Mprefix-DT[simp]*: $\langle \forall x \in A. P x \sqsubseteq_{DT} Q x \implies Mprefix A P \sqsubseteq_{DT} Mprefix A Q \rangle$
<proof>

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

lemma *mono-Mprefix-DT-set[simp]*: $\langle A \subseteq B \implies Mprefix B P \sqsubseteq_{DT} Mprefix A P \rangle$
<proof>

corollary *mono-Mprefix-D-set[simp]*: $\langle A \subseteq B \implies Mprefix B P \sqsubseteq_D Mprefix A P \rangle$
and *mono-Mprefix-T-set[simp]*: $\langle A \subseteq B \implies Mprefix B P \sqsubseteq_T Mprefix A P \rangle$
<proof>

lemma *mono-Hiding-F[simp]*: $\langle P \sqsubseteq_F Q \implies (P \setminus A) \sqsubseteq_F (Q \setminus A) \rangle$
<proof>

lemma *mono-Hiding-T[simp]*: $\langle P \sqsubseteq_T Q \implies (P \setminus A) \sqsubseteq_T (Q \setminus A) \rangle$
<proof>

lemma *mono-Hiding-DT[simp]*: $\langle P \sqsubseteq_{DT} Q \implies (P \setminus A) \sqsubseteq_{DT} (Q \setminus A) \rangle$
<proof>

lemmas $\text{mono-Hiding-FD}[simp] = \text{mono-Hiding-FD}[\text{folded failure-divergence-refine-def}]$

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

lemma $\text{mono-Sync-DT}[simp]$: $\langle P \sqsubseteq_{DT} P' \implies Q \sqsubseteq_{DT} Q' \implies (P \llbracket A \rrbracket Q) \sqsubseteq_{DT} (P' \llbracket A \rrbracket Q') \rangle$
 $\langle \text{proof} \rangle$

lemmas $\text{mono-Sync-FD}[simp] = \text{mono-Sync-FD}[\text{folded failure-divergence-refine-def}]$

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

lemma $\text{mono-Mndetprefix-F}[simp]$: $\langle \forall x \in A. P x \sqsubseteq_F Q x \implies \text{Mndetprefix } A P \sqsubseteq_F \text{Mndetprefix } A Q \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{mono-Mndetprefix-D}[simp]$: $\langle \forall x \in A. P x \sqsubseteq_D Q x \implies \text{Mndetprefix } A P \sqsubseteq_D \text{Mndetprefix } A Q \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{mono-Mndetprefix-T}[simp]$: $\langle \forall x \in A. P x \sqsubseteq_T Q x \implies \text{Mndetprefix } A P \sqsubseteq_T \text{Mndetprefix } A Q \rangle$
 $\langle \text{proof} \rangle$

corollary $\text{mono-Mndetprefix-DT}[simp]$: $\langle \forall x \in A. P x \sqsubseteq_{DT} Q x \implies \text{Mndetprefix } A P \sqsubseteq_{DT} \text{Mndetprefix } A Q \rangle$
 $\langle \text{proof} \rangle$

lemmas $\text{mono-Mndetprefix-FD}[simp] = \text{mono-Mndetprefix-FD}[\text{folded failure-divergence-refine-def}]$

lemmas $\text{mono-Mndetprefix-FD-set}[simp] = \text{Mndetprefix-FD-subset}[\text{folded failure-divergence-refine-def}]$

corollary $\text{mono-Mndetprefix-F-set}[simp]$: $\langle A \neq \{\} \implies A \subseteq B \implies \text{Mndetprefix } B P \sqsubseteq_F \text{Mndetprefix } A P \rangle$

and $\text{mono-Mndetprefix-D-set}[simp]$: $\langle A \subseteq B \implies \text{Mndetprefix } B P \sqsubseteq_D \text{Mndetprefix } A P \rangle$

and $\text{mono-Mndetprefix-T-set}[simp]$: $\langle A \subseteq B \implies \text{Mndetprefix } B P \sqsubseteq_T \text{Mndetprefix } A P \rangle$

and $\text{mono-Mndetprefix-DT-set}[simp]$: $\langle A \subseteq B \implies \text{Mndetprefix } B P \sqsubseteq_{DT} \text{Mndetprefix } A P \rangle$
 $\langle \text{proof} \rangle$

lemmas $\text{Mprefix-refines-Mndetprefix-FD}[simp] = \text{Mprefix-refines-Mndetprefix}[\text{folded failure-divergence-refine-def}]$

corollary $\text{Mprefix-refines-Mndetprefix-F}[simp]$: $\langle \text{Mndetprefix } A P \sqsubseteq_F \text{Mprefix } A \rangle$

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

lemma $mono-read-FD[simp, elim]$: $(\bigwedge x. P x \sqsubseteq_{FD} Q x) \implies (c?x \rightarrow (P x)) \sqsubseteq_{FD}$
 $(c?x \rightarrow (Q x))$
 $\langle proof \rangle$

lemma $mono-write-FD[simp, elim]$: $(P \sqsubseteq_{FD} Q) \implies (c!x \rightarrow P) \sqsubseteq_{FD} (c!x \rightarrow Q)$
 $\langle proof \rangle$

lemma $mono-write0-FD[simp, elim]$: $P \sqsubseteq_{FD} Q \implies (a \rightarrow P) \sqsubseteq_{FD} (a \rightarrow Q)$
 $\langle proof \rangle$

lemma $mono-read-DT[simp, elim]$: $(\bigwedge x. P x \sqsubseteq_{DT} Q x) \implies (c?x \rightarrow (P x)) \sqsubseteq_{DT}$
 $(c?x \rightarrow (Q x))$
 $\langle proof \rangle$

lemma $mono-write-DT[simp, elim]$: $(P \sqsubseteq_{DT} Q) \implies (c!x \rightarrow P) \sqsubseteq_{DT} (c!x \rightarrow Q)$
 $\langle proof \rangle$

lemma $mono-write0-DT[simp, elim]$: $P \sqsubseteq_{DT} Q \implies (a \rightarrow P) \sqsubseteq_{DT} (a \rightarrow Q)$
 $\langle proof \rangle$

lemma $mono-Renaming-D$: $\langle P \sqsubseteq_D Q \implies Renaming P f \sqsubseteq_D Renaming Q f \rangle$
 $\langle proof \rangle$

lemma $mono-Renaming-FD$: $\langle P \sqsubseteq_{FD} Q \implies Renaming P f \sqsubseteq_{FD} Renaming Q f \rangle$
 $\langle proof \rangle$

lemma $mono-Renaming-DT$: $\langle P \sqsubseteq_{DT} Q \implies Renaming P f \sqsubseteq_{DT} Renaming Q f \rangle$
 $\langle proof \rangle$

Some new and very useful results

lemma *Ndet-is-STOP-iff*: $\langle P \sqcap Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP \rangle$
 $\langle proof \rangle$

lemma *Det-is-STOP-iff*: $\langle P \sqcap Q = STOP \longleftrightarrow P = STOP \wedge Q = STOP \rangle$
 $\langle proof \rangle$

lemma *Det-is-BOT-iff*: $\langle P \sqcap Q = \perp \longleftrightarrow P = \perp \vee Q = \perp \rangle$
 $\langle proof \rangle$

lemma *Ndet-is-BOT-iff*: $\langle P \sqcap Q = \perp \longleftrightarrow P = \perp \vee Q = \perp \rangle$
 $\langle proof \rangle$

lemma *Sync-is-BOT-iff*: $\langle P \llbracket S \rrbracket Q = \perp \longleftrightarrow P = \perp \vee Q = \perp \rangle$
 $\langle proof \rangle$

lemma *STOP-neq-BOT*: $\langle STOP \neq \perp \rangle$
 $\langle proof \rangle$

lemma *SKIP-neq-BOT*: $\langle SKIP \neq \perp \rangle$
 $\langle proof \rangle$

lemma *Mprefix-neq-BOT*: $\langle Mprefix A P \neq \perp \rangle$
 $\langle proof \rangle$

lemma *Mndetprefix-neq-BOT*: $\langle Mndetprefix A P \neq \perp \rangle$
 $\langle proof \rangle$

lemma *STOP-T-iff*: $\langle STOP \sqsubseteq_T P \longleftrightarrow P = STOP \rangle$
 $\langle proof \rangle$

lemma *STOP-F-iff*: $\langle STOP \sqsubseteq_F P \longleftrightarrow P = STOP \rangle$
 $\langle proof \rangle$

lemma *STOP-FD-iff*: $\langle STOP \sqsubseteq_{FD} P \longleftrightarrow P = STOP \rangle$
 $\langle proof \rangle$

lemma *SKIP-FD-iff*: $\langle SKIP \sqsubseteq_{FD} P \longleftrightarrow P = SKIP \rangle$
 $\langle proof \rangle$

lemma *SKIP-F-iff*: $\langle SKIP \sqsubseteq_F P \longleftrightarrow P = SKIP \rangle$
 $\langle proof \rangle$

lemma *Seq-is-SKIP-iff*: $\langle P ; Q = SKIP \longleftrightarrow P = SKIP \wedge Q = SKIP \rangle$

<proof>

lemma *cont-process-rec*: $\langle P = (\mu X. f X) \implies cont f \implies P = f P \rangle$ *<proof>*

end

theory *Assertions*
imports *CSP*
begin

default-sort *type*

8.3 CSP Assertions

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

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

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

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

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

8.4 Some deadlock freeness laws

lemma *DF-subset*: $A \neq \{\} \implies A \subseteq B \implies DF B \sqsubseteq_{FD} DF A$
<proof>

lemma *DF-Univ-freeness*: $A \neq \{\} \implies (DF A) \sqsubseteq_{FD} P \implies deadlock-free P$
<proof>

lemma *deadlock-free-Ndet*: $deadlock-free P \implies deadlock-free Q \implies deadlock-free (P \sqcap Q)$
<proof>

8.5 Preliminaries

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

8.6 Deadlock Free

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

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

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

8.7 Run

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

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

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

8.8 Reference processes and their unfolding rules

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

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

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

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

8.9 Process events and reference processes events

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

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

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

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

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

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

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

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

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

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

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

8.10 Relations between refinements on reference processes

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

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

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

<proof>

lemma $DF_{SKIP}\text{-}DF\text{-refine}\text{-}F$: $DF_{SKIP} A \sqsubseteq_F DF A$
<proof>

lemma $DF\text{-}RUN\text{-refine}\text{-}F$: $DF A \sqsubseteq_F RUN A$
<proof>

lemma $CHAOS\text{-}DF\text{-refine}\text{-}F$: $CHAOS A \sqsubseteq_F DF A$
<proof>

corollary $CHAOS_{SKIP}\text{-}CHAOS\text{-refine}\text{-}F$: $CHAOS_{SKIP} A \sqsubseteq_F CHAOS A$
and $CHAOS_{SKIP}\text{-}DF_{SKIP}\text{-refine}\text{-}F$: $CHAOS_{SKIP} A \sqsubseteq_F DF_{SKIP} A$
<proof>

lemma $div\text{-free}\text{-}CHAOS_{SKIP}$: $\mathcal{D}(CHAOS_{SKIP} A) = \{\}$
<proof>

lemma $div\text{-free}\text{-}CHAOS$: $\mathcal{D}(CHAOS A) = \{\}$
<proof>

lemma $div\text{-free}\text{-}RUN$: $\mathcal{D}(RUN A) = \{\}$
<proof>

corollary $DF_{SKIP}\text{-}DF\text{-refine}\text{-}FD$: $DF_{SKIP} A \sqsubseteq_{FD} DF A$
and $DF\text{-}RUN\text{-refine}\text{-}FD$: $DF A \sqsubseteq_{FD} RUN A$
and $CHAOS\text{-}DF\text{-refine}\text{-}FD$: $CHAOS A \sqsubseteq_{FD} DF A$
and $CHAOS_{SKIP}\text{-}CHAOS\text{-refine}\text{-}FD$: $CHAOS_{SKIP} A \sqsubseteq_{FD} CHAOS A$
and $CHAOS_{SKIP}\text{-}DF_{SKIP}\text{-refine}\text{-}FD$: $CHAOS_{SKIP} A \sqsubseteq_{FD} DF_{SKIP} A$
<proof>

lemma $traces\text{-}CHAOS\text{-sub}$: $\mathcal{T}(CHAOS A) \subseteq \{s. \text{set } s \subseteq ev \text{ ' } A\}$
<proof>

lemma $traces\text{-}RUN\text{-sub}$: $\{s. \text{set } s \subseteq ev \text{ ' } A\} \subseteq \mathcal{T}(RUN A)$
<proof>

corollary $RUN\text{-all}\text{-tickfree}\text{-traces1}$: $\mathcal{T}(RUN A) = \{s. \text{set } s \subseteq ev \text{ ' } A\}$
and $DF\text{-all}\text{-tickfree}\text{-traces1}$: $\mathcal{T}(DF A) = \{s. \text{set } s \subseteq ev \text{ ' } A\}$
and $CHAOS\text{-all}\text{-tickfree}\text{-traces1}$: $\mathcal{T}(CHAOS A) = \{s. \text{set } s \subseteq ev \text{ ' } A\}$
<proof>

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

8.10. RELATIONS BETWEEN REFINEMENTS ON REFERENCE PROCESSES 97

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

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

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

corollary *DF_{SKIP}-all-front-tickfree-traces1*:
 $\mathcal{T}(\text{DF}_{\text{SKIP}} A) = \{s. \text{front-tickFree } s \wedge \text{set } s \subseteq (\text{ev } 'A \cup \{\text{tick}\})\}$

and *CHAOS_{SKIP}-all-front-tickfree-traces1*:
 $\mathcal{T}(\text{CHAOS}_{\text{SKIP}} A) = \{s. \text{front-tickFree } s \wedge \text{set } s \subseteq (\text{ev } 'A \cup \{\text{tick}\})\}$
 ⟨proof⟩

corollary *DF_{SKIP}-all-front-tickfree-traces2*: $\text{front-tickFree } s \implies s \in \mathcal{T}(\text{DF}_{\text{SKIP}} \text{UNIV})$

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

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

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

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

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

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

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

corollary *deadlock-free-v2-is-right-wrt-events:*

$$\begin{aligned} & \text{deadlock-free-v2 } (P :: 'a \text{ process}) \longleftrightarrow \\ & (\forall s \in \mathcal{T} P. \text{tickFree } s \longrightarrow (s, \{\text{tick}\} \cup \text{ev } (\text{events-of } P)) \notin \mathcal{F} P) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *deadlock-free-is-deadlock-free-v2:*

$$\begin{aligned} & \text{deadlock-free } P \Longrightarrow \text{deadlock-free-v2 } P \\ & \langle \text{proof} \rangle \end{aligned}$$

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

lemma *deadlock-free-v2-SKIP: deadlock-free-v2 SKIP*

$\langle \text{proof} \rangle$

lemma *non-deadlock-free-SKIP: $\langle \neg \text{deadlock-free SKIP} \rangle$*

$\langle \text{proof} \rangle$

lemma *non-deadlock-free-v2-STOP: $\langle \neg \text{deadlock-free-v2 STOP} \rangle$*

$\langle \text{proof} \rangle$

lemma *non-deadlock-free-STOP: $\langle \neg \text{deadlock-free STOP} \rangle$*

$\langle \text{proof} \rangle$

8.12 Non-terminating Runs

definition *non-terminating :: 'a process \Rightarrow bool*

where *non-terminating* $P \equiv \text{RUN UNIV} \sqsubseteq_T P$

corollary *non-terminating-refine-DF: non-terminating* $P = \text{DF UNIV} \sqsubseteq_T P$

and *non-terminating-refine-CHAOS: non-terminating* $P = \text{CHAOS UNIV} \sqsubseteq_T P$

$\langle \text{proof} \rangle$

lemma *non-terminating-is-right: non-terminating* $(P :: 'a \text{ process}) \longleftrightarrow (\forall s \in \mathcal{T} P. \text{tickFree } s)$

$\langle \text{proof} \rangle$

lemma *nonterminating-implies-div-free: non-terminating* $P \Longrightarrow \mathcal{D} P = \{\}$

$\langle \text{proof} \rangle$

lemma *non-terminating-implies-F: non-terminating* $P \Longrightarrow \text{CHAOS UNIV} \sqsubseteq_F P$

$\langle \text{proof} \rangle$

corollary *non-terminating-F: non-terminating* $P = \text{CHAOS UNIV} \sqsubseteq_F P$

$\langle \text{proof} \rangle$

corollary *non-terminating-FD: non-terminating* $P = \text{CHAOS UNIV} \sqsubseteq_{FD} P$

<proof>

8.13 Lifelock Freeness

thm *lifelock-free-def*

corollary *lifelock-free-is-non-terminating*: $lifelock\text{-}free\ P = non\text{-}terminating\ P$
<proof>

lemma *div-free-divergence-refine*:

$$\mathcal{D} P = \{\} \longleftrightarrow CHAOS_{SKIP}\ UNIV \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow CHAOS\ UNIV \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow RUN\ UNIV \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow DF_{SKIP}\ UNIV \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow DF\ UNIV \sqsubseteq_D P$$

<proof>

definition *lifelock-free-v2* :: 'a process \Rightarrow bool

where $lifelock\text{-}free\text{-}v2\ P \equiv CHAOS_{SKIP}\ UNIV \sqsubseteq_{FD}\ P$

lemma *div-free-is-lifelock-free-v2*: $lifelock\text{-}free\text{-}v2\ P \longleftrightarrow \mathcal{D} P = \{\}$
<proof>

lemma *lifelock-free-is-lifelock-free-v2*: $lifelock\text{-}free\ P \Longrightarrow lifelock\text{-}free\text{-}v2\ P$
<proof>

corollary *deadlock-free-v2-is-lifelock-free-v2*: $deadlock\text{-}free\text{-}v2\ P \Longrightarrow lifelock\text{-}free\text{-}v2\ P$
<proof>

8.14 New laws

lemma *non-terminating-Seq*:

$$non\text{-}terminating\ P \Longrightarrow (P ; Q) = P$$

<proof>

lemma *non-terminating-Sync*:

$$non\text{-}terminating\ P \Longrightarrow lifelock\text{-}free\text{-}v2\ Q \Longrightarrow non\text{-}terminating\ (P \llbracket A \rrbracket Q)$$

<proof>

lemmas *non-terminating-Par* = *non-terminating-Sync*[**where** $A = \langle UNIV \rangle$]
and *non-terminating-Inter* = *non-terminating-Sync*[**where** $A = \langle \{\} \rangle$]

end

Chapter 9

Conclusion

9.1 Related Work

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

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

9.2 Lessons learned

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

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

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

9.3 A Summary on New Results

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

1. $?P \sqsubseteq ?Q \implies ?P \setminus ?A \sqsubseteq ?Q \setminus ?A$ 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* $?A \implies ?P \setminus ?A \cup ?B = ?P \setminus ?A \setminus ?B$); this was not even proven in HOL-CSP 1.0 for the singleton case! It can be considered as the most complex theorem of this theory.
3. distribution laws of *Hiding* over *Sync* $\llbracket \text{finite } ?A; ?A \cap ?S = \{\} \rrbracket \implies ?P \llbracket ?S \rrbracket ?Q \setminus ?A = (?P \setminus ?A) \llbracket ?S \rrbracket (?Q \setminus ?A)$; however, this works only in the finite case. A true monster proof.
4. distribution of *Mprefix* over *Sync* $\llbracket ?A \cap ?S = \{\}; ?A' \subseteq ?S; ?B \cap ?S = \{\}; ?B' \subseteq ?S \rrbracket \implies \text{Mprefix } (?A \cup ?A') ?P \llbracket ?S \rrbracket \text{Mprefix } (?B \cup ?B') ?Q = (\Box x \in ?A \rightarrow ?P \ x \llbracket ?S \rrbracket \text{Mprefix } (?B \cup ?B') ?Q) \Box (\Box y \in ?B \rightarrow \text{Mprefix } (?A \cup ?A') ?P \llbracket ?S \rrbracket ?Q \ y) \Box (\Box x \in ?A' \cap ?B' \rightarrow ?P \ x \llbracket ?S \rrbracket ?Q \ x)$ in the most generalized case. Also a true monster proof, but reworked using symmetries and abstractions to be more reasonable (and faster)

5. the synchronization operator is associative $?P \llbracket ?S \rrbracket ?Q \llbracket ?S \rrbracket ?T = ?P \llbracket ?S \rrbracket (?Q \llbracket ?S \rrbracket ?T)$. (In HOL-CSP 1.0, this had only be shown for special cases like $?P \parallel ?Q \parallel ?T = ?P \parallel (?Q \parallel ?T)$).
6. the generalized non-deterministic choice operator — relevant for proofs of deadlock-freeness — has been added to the theory $Mndetprefix \equiv map\text{-}fun\ id\ (map\text{-}fun\ id\ Abs\text{-}process)\ (\lambda A\ P.\ if\ A = \{\}\ then\ Rep\text{-}process\ STOP\ else\ (\bigcup_{x \in A} \mathcal{F}\ (x \rightarrow P\ x), \bigcup_{x \in A} \mathcal{D}\ (x \rightarrow P\ x)))$; it is proven monotone in the general case and continuous for the special case $(\bigwedge x.\ cont\ (?f\ x)) \implies cont\ (\lambda y.\ \sqcap z \in ?A \rightarrow ?f\ z\ y)$ relevant for the definition of the deadlock reference processes $DF\ ?A \equiv \mu x.\ \sqcap xa \in ?A \rightarrow x$ and $DF_{SKIP}\ ?A \equiv \mu x.\ (\sqcap xa \in ?A \rightarrow x) \sqcap SKIP$.

end

Chapter 10

Annex: Refinement Example with Buffer over infinite Alphabet

```
theory    CopyBuffer
imports  CSP Assertions
begin
```

10.1 Defining the Copy-Buffer Example

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

```
definition SYN :: ('a channel) set
where     SYN ≡ (range mid) ∪ {ack}
```

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

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

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

```
definition SYSTEM :: ('a channel) process
where     SYSTEM ≡ (SEND [ SYN ] REC) \ SYN
```

```
thm SYSTEM-def
```

10.2 The Standard Proof

10.2.1 Channels and Synchronization Sets

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

lemma *[simp]*: $left\ x \notin SYN$
<proof>

lemma *[simp]*: $right\ x \notin SYN$
<proof>

lemma *[simp]*: $ack \in SYN$
<proof>

lemma *[simp]*: $mid\ x \in SYN$
<proof>

lemma *[simp]*: $inj\ mid$
<proof>

lemma *finite* ($SYN:: 'a\ channel\ set$) $\implies\ finite\ \{(t::'a).\ True\}$
<proof>

10.2.2 Definitions by Recursors

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

lemma *COPY-rec*:
 $COPY = left?\ x \rightarrow right!\ x \rightarrow COPY$
<proof>

lemma *SEND-rec*:
 $SEND = left?\ x \rightarrow mid!\ x \rightarrow (ack \rightarrow SEND)$
<proof>

lemma *REC-rec*:
 $REC = mid?\ x \rightarrow right!\ x \rightarrow (ack \rightarrow REC)$
<proof>

10.2.3 A Refinement Proof

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

lemma *impl-refines-spec* : $COPY \sqsubseteq_{FD} SYSTEM$
<proof>

lemma *spec-refines-impl* :
assumes *fin*: *finite (SYN:: 'a channel set)*
shows $SYSTEM \sqsubseteq_{FD} (COPY :: 'a channel process)$
<proof>

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

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

10.2.4 Deadlock Freeness Proof

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

lemma *DF-COPY* : $(DF (range left \cup range right)) \sqsubseteq_{FD} COPY$
<proof>

10.3 An Alternative Approach: Using the fixrec-Package

10.3.1 Channels and Synchronisation Sets

As before.

10.3.2 Process Definitions via fixrec-Package

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

thm *COPY'-rec*

definition $SYSTEM' :: ('a channel) process$
where $\langle SYSTEM' \equiv ((SEND' \llbracket SYN \rrbracket REC') \setminus SYN) \rangle$

10.3.3 Another Refinement Proof on fixrec-infrastructure

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

thm *COPY'-SEND'-REC'.induct*

lemma *impl-refines-spec'* : $(COPY'::'a\ channel\ process) \sqsubseteq_{FD} SYSTEM'$
<proof>

lemma *spec-refines-impl'* :

assumes *fin*: *finite* (*SYN::('a channel) set*)

shows $SYSTEM' \sqsubseteq_{FD} (COPY'::'a\ channel\ process)$

<proof>

lemma *spec-equal-impl'* :

assumes *fin*: *finite* (*SYN::('a channel) set*)

shows $SYSTEM' = (COPY'::'a\ channel\ process)$

<proof>

end

Bibliography

- [1] P. Broadfoot and B. Roscoe. Tutorial on fdr and its applications. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, pages 322–322, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
- [2] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
- [3] A. J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer London.
- [4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, July 1993.
- [5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [6] Y. Isobe and M. Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.
- [7] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. Uniform workbench — universelle entwicklungs-umgebung 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.