HOL-CSPM - Architectural operators for HOL-CSP

Benoît Ballenghien Safouan Taha Burkhart Wolff

April 18, 2024

Abstract

Recently, a modern version of Roscoes and Brookes [1] Failure-Divergence Semantics for CSP has been formalized in Isabelle [3].

The session HOL-CSP introduces among other things some binary operators on processes that we will here generalize in a fully-abstract way.

On these "architectural operators", we will prove the properties of refinement, the rules of continuity and the laws of interaction so that they can be easily used.

Finally, we will give examples of their usefulness when trying to model complex systems.

Contents

Chapter 1

Introduction

1.1 Motivations

`HOL-CSP` [3] 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" [2] and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes" [1].

In the session `HOL-CSP` are introduced the type $'\alpha$ *process*, several classic CSP operators and number of laws that govern their interactions.

Four of them are binary operators: the non-deterministic choice $P \sqcap Q$, the deterministic choice $P \;\square\; Q$, the synchronization $P\ [\![S]\!]\ Q$ and the sequential composition $P\ ;\ Q$.

Analogously to the finite sum $\sum_{i\,=\,0}^{n}\ a\ i$ which is generalization of the addition $a + b$, we define generalisations of the binary operators of CSP.

The most straight-forward way to do so would be a fold on a list of processes. However, in many cases, we have additional properties, like commutativity, idempotency, etc. that allow for stronger/more abstract constructions. In particular, in several cases, generalization to unbounded and even infinite index-sets are possible.

The notations we choose are widely inspired by the CSP_M syntax of FDR: https://cocotec.io/fdr/manual/cspm.html.

In this session we therefore introduce the multi-operators associated respectively with $P \sqcap Q$, $P \;\square\; Q$, $P\ [\![S]\!]\ Q$ and $P\ ;\ Q$. We prove their continuity and refinements rules, as well as some laws governing their interactions.

We also give the definitions of the POTS and Dining Philosophers examples, which greatly benefit from the newly introduced generalized operators.

Since they appear naturally when modeling complex architectures, we may call them *architectural operators* of CSP.

Finally this session also includes results on the notion of *events-of*, and a very powerful result about *deadlock-free* and *Sync*: the interleaving $P|||Q$ is *deadlock-free* if P and Q are.

1.2 The Global Architecture of HOL-CSPM

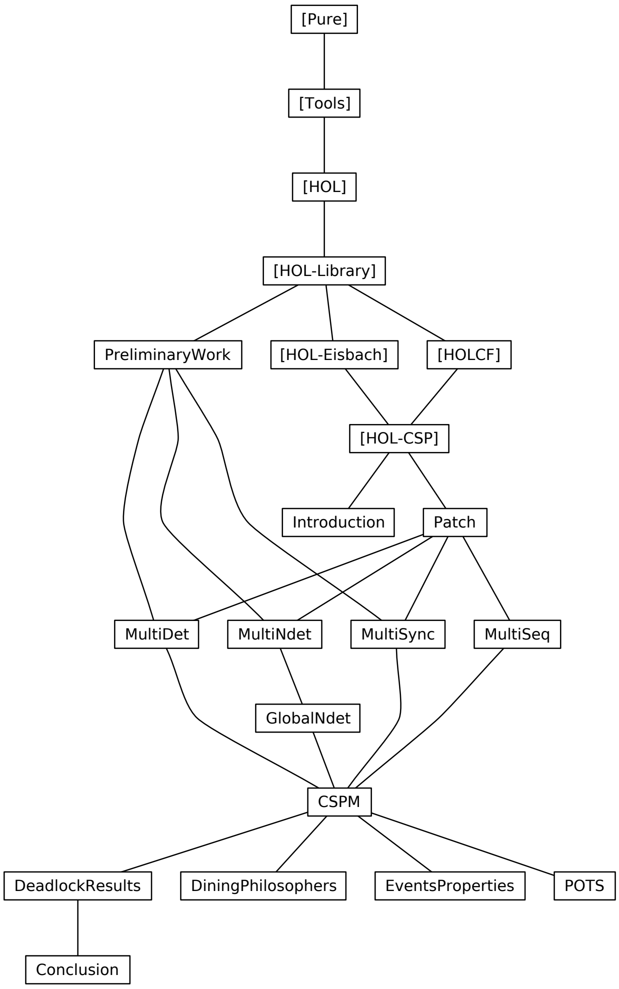

Figure 1.1: The overall architecture

The global architecture of HOL-CSPM is shown in Figure 1.1. The entire package resides on:

1. `HOL-Eisbach` from the Isabelle/HOL distribution,
2. `HOLCF` from the Isabelle/HOL distribution, and
3. `HOL-CSP` 2.0 from the Isabelle Archive of Formal Proofs.

Chapter 2

Some Preliminary Work

theory *PreliminaryWork*
  imports *HOL−Library.Multiset*
begin

2.1 Induction Rules for $'\alpha$ *set*

lemma *finite-subset-induct-singleton*
  [*consumes 3*, *case-names singleton insertion*]:
  ‹$[\![a \in A;$ *finite* $F;\ F \subseteq A;\ P\ \{a\};$
  $\bigwedge x\ F.$ *finite* $F \Longrightarrow x \in A \Longrightarrow x \notin (insert\ a\ F) \Longrightarrow P\ (insert\ a\ F)$
  $\Longrightarrow P\ (insert\ x\ (insert\ a\ F))]\!] \Longrightarrow P\ (insert\ a\ F)$›
  ⟨*proof*⟩

lemma *finite-set-induct-nonempty*
  [*consumes 2*, *case-names singleton insertion*]:
  assumes ‹$A \neq \{\}$› **and** ‹*finite* A›
  and *singleton*: ‹$\bigwedge a.\ a \in A \Longrightarrow P\ \{a\}$›
  and *insert*: ‹$\bigwedge x\ F.\ [\![F \neq \{\};$ *finite* $F;\ x \in A;\ x \notin F;\ P\ F]\!]$
  $\Longrightarrow P\ (insert\ x\ F)$›
  shows ‹$P\ A$›
⟨*proof*⟩

lemma *finite-subset-induct-singleton′*
  [*consumes 3*, *case-names singleton insertion*]:
  ‹$[\![a \in A;$ *finite* $F;\ F \subseteq A;\ P\ \{a\};$
  $\bigwedge x\ F.\ [\![$*finite* $F;\ x \in A;\ insert\ a\ F \subseteq A;\ x \notin insert\ a\ F;\ P\ (insert\ a\ F)]\!]$
  $\Longrightarrow P\ (insert\ x\ (insert\ a\ F))]\!]$
  $\Longrightarrow P\ (insert\ a\ F)$›
  ⟨*proof*⟩

lemma *induct-subset-empty-single*[*consumes 1*]:

‹⟦*finite* A; P {}; ∀ a ∈ A. P {a};
⋀F a. ⟦a ∈ A; *finite* F; F ⊆ A; F ≠ {}; P F⟧ ⟹ P (*insert* a F)⟧ ⟹ P A›
⟨*proof*⟩

2.2 Induction Rules for *'α multiset*

The following rule comes directly from *HOL−Library.Multiset* but is written with *consumes 2* instead of *consumes 1*. I rewrite here a correct version.

lemma *msubset-induct* [*consumes 1*, *case-names empty add*]:
‹⟦F ⊆# A; P {#}; ⋀a F. ⟦a ∈# A; P F⟧ ⟹ P (*add-mset* a F)⟧ ⟹ P F›
⟨*proof*⟩

lemma *msubset-induct-singleton* [*consumes 2*, *case-names m-singleton add*]:
‹⟦a ∈# A; F ⊆# A; P {#a#};
⋀x F. ⟦x ∈# A; P (*add-mset* a F)⟧ ⟹ P (*add-mset* x (*add-mset* a F))⟧
⟹ P (*add-mset* a F)›
⟨*proof*⟩

lemma *mset-induct-nonempty* [*consumes 1*, *case-names m-singleton add*]:
assumes ‹A ≠ {#}›
and *m-singleton*: ‹⋀a. a ∈# A ⟹ P {#a#}›
and *add*: ‹⋀x F. ⟦F ≠ {#}; x ∈# A; P F⟧ ⟹ P (*add-mset* x F)›
shows ‹P A›
⟨*proof*⟩

lemma *msubset-induct′* [*consumes 2*, *case-names empty add*]:
assumes ‹F ⊆# A›
and *empty*: ‹P {#}›
and *insert*: ‹⋀a F. ⟦a ∈# A − F; F ⊆# A; P F⟧ ⟹ P (*add-mset* a F)›
shows ‹P F›
⟨*proof*⟩

lemma *msubset-induct-singleton′* [*consumes 2*, *case-names m-singleton add*]:
‹⟦a ∈# A − F; F ⊆# A; P {#a#};
⋀x F. ⟦x ∈# A − F; F ⊆# A; P (*add-mset* a F)⟧
⟹ P (*add-mset* x (*add-mset* a F))⟧
⟹ P (*add-mset* a F)›
⟨*proof*⟩

lemma *msubset-induct-singleton″* [*consumes 1*, *case-names m-singleton add*]:
‹⟦*add-mset* a F ⊆# A; P {#a#};
⋀x F. ⟦*add-mset* x (*add-mset* a F) ⊆# A; P (*add-mset* a F)⟧
⟹ P (*add-mset* x (*add-mset* a F))⟧

$\Longrightarrow P$ (*add-mset a F*)›
⟨*proof*⟩

lemma *mset-induct-nonempty′* [*consumes 1*, *case-names m-singleton add*]:
assumes *nonempty*: ‹$A \neq \{\#\}$› **and** *m-singleton*: ‹$\bigwedge a.\ a \in\# A \Longrightarrow P\ \{\#a\#\}$›
and *hyp*: ‹$\bigwedge a\ x\ F.\ [\![a \in\# A;\ x \in\# A - \textit{add-mset}\ a\ F;\ \textit{add-mset}\ a\ F \subseteq\# A;$
$P\ (\textit{add-mset}\ a\ F)]\!] \Longrightarrow P\ (\textit{add-mset}\ x\ (\textit{add-mset}\ a\ F))$›
shows ‹$P\ A$›
⟨*proof*⟩

lemma *induct-subset-mset-empty-single*:
‹$[\![P\ \{\#\};\ \forall a \in\# M.\ P\ \{\#a\#\};$
$\bigwedge N\ a.\ [\![a \in\# M;\ N \subseteq\# M;\ N \neq \{\#\};\ P\ N]\!] \Longrightarrow P\ (\textit{add-mset}\ a\ N)]\!] \Longrightarrow P\ M$›
⟨*proof*⟩

2.3 Strong Induction for *nat*

lemma *strong-nat-induct*[*consumes 0*, *case-names 0 Suc*]:
‹$[\![P\ 0;\ \bigwedge n.\ (\bigwedge m.\ m \leq n \Longrightarrow P\ m) \Longrightarrow P\ (\textit{Suc}\ n)]\!] \Longrightarrow P\ n$›
⟨*proof*⟩

lemma *strong-nat-induct-non-zero*[*consumes 1*, *case-names 1 Suc*]:
‹$[\![0 < n;\ P\ 1;\ \bigwedge n.\ 0 < n \Longrightarrow (\bigwedge m.\ 0 < m \wedge m \leq n \Longrightarrow P\ m) \Longrightarrow P\ (\textit{Suc}\ n)]\!]$
$\Longrightarrow P\ n$›
⟨*proof*⟩

2.4 Preliminaries for Cartesian Product Results

lemma *prem-Multi-cartprod*:
‹$(\lambda(x,\ y).\ x\ @\ y\ )\ `\ (A\ \times B\ ) = \{s\ @\ t\ \ |s\ t.\ (s,\ t) \in A\ \times B\ \}$›
‹$(\lambda(x,\ y).\ x\ \#\ y\ )\ `\ (A' \times B\ ) = \{s\ \#\ t\ \ |s\ t.\ (s,\ t) \in A' \times B\ \}$›
‹$(\lambda(x,\ y).\ [x,\ y])\ `\ (A' \times B') = \{[s,\ t]\ |s\ t.\ (s,\ t) \in A' \times B'\}$›
⟨*proof*⟩

end

Chapter 3

Patch for Compatibility

theory *Patch*
  imports *HOL−CSP.Assertions*
begin

`HOL-CSP` significantly changed during the past months, but not all the modifications appear in the current version on the AFP. This theory fixes the incompatibilities and will be removed in the next release.

3.1 Results

lemma *Mprefix-Det-distr*:
  ‹$(\square\ a \in A \rightarrow P\ a)\ \square\ (\square\ b \in B \rightarrow Q\ b) =$
   $\square\ x \in A \cup B \rightarrow (\ \ \textit{if}\ x \in A \cap B\ \textit{then}\ P\ x \sqcap Q\ x$
                    $\textit{else if}\ x \in A\ \textit{then}\ P\ x\ \textit{else}\ Q\ x)$›
  (**is** ‹*?lhs* = *?rhs*›)
⟨*proof*⟩

lemma *D-expand* :
  ‹$\mathcal{D}\ P = \{t1\ @\ t2|\ t1\ t2.\ t1 \in \mathcal{D}\ P \wedge \textit{tickFree}\ t1 \wedge \textit{front-tickFree}\ t2\}$›
  (**is** ‹$\mathcal{D}\ P$ = *?rhs*›)
⟨*proof*⟩

3.1.1 Continuity Rule

Monotonicity of *Renaming*.

lemma *mono-Renaming*[*simp*] : ‹$(\textit{Renaming}\ P\ f) \sqsubseteq (\textit{Renaming}\ Q\ f)$› **if** ‹$P \sqsubseteq Q$›
⟨*proof*⟩

Useful Results about *finitary*, and Preliminaries Lemmas for Continuity.

lemma *le-snoc-is* : ‹$t \leq s\ @\ [x] \longleftrightarrow t = s\ @\ [x] \vee t \leq s$›
  ⟨*proof*⟩

lemma *Cont-RenH5*: ‹*finite* ($\bigcup$ *t* ∈ {*t*. *t* ≤ (*s* :: ′α *trace*)}. {*s*. *t*=*map* (*EvExt f*) *s*})› **if** ‹*finitary f*›
⟨*proof*⟩

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

Finally, Continuity !

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

3.1.2 Nice Properties

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

3.1.3 Renaming Laws

lemma *Renaming-Mprefix-inj-on*:
‹*Renaming* (*Mprefix A P*) *f* = □ *b* ∈ *f* ‘ *A* → *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f*›
if *inj-on-f*: ‹*inj-on f A*›
⟨*proof*⟩

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

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

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

3.2 Assertions

abbreviation $deadlock\text{-}free_{SKIP}$ :: *'a process ⇒ bool*
where $deadlock\text{-}free_{SKIP} \equiv deadlock\text{-}free\text{-}v2$

lemma *deadlock-free-implies-lifelock-free*: ‹*deadlock-free P* ⟹ *lifelock-free P*›
⟨*proof*⟩

lemmas $deadlock\text{-}free_{SKIP}$*-def* = *deadlock-free-v2-def*
and $deadlock\text{-}free_{SKIP}$*-is-right* = *deadlock-free-v2-is-right*
and $deadlock\text{-}free_{SKIP}$*-implies-div-free* = *deadlock-free-v2-implies-div-free*
and $deadlock\text{-}free_{SKIP}$*-FD* = *deadlock-free-v2-FD*
and $deadlock\text{-}free_{SKIP}$*-is-right-wrt-events* = *deadlock-free-v2-is-right-wrt-events*
and *deadlock-free-is-*$deadlock\text{-}free_{SKIP}$ = *deadlock-free-is-deadlock-free-v2*
and $deadlock\text{-}free_{SKIP}$*-SKIP* = *deadlock-free-v2-SKIP*
and *non-*$deadlock\text{-}free_{SKIP}$*-STOP* = *non-deadlock-free-v2-STOP*

3.3 Lifelock Freeness

definition $lifelock\text{-}free_{SKIP}$ :: *'a process ⇒ bool*
where $lifelock\text{-}free_{SKIP}\ P \equiv CHAOS_{SKIP}\ UNIV \sqsubseteq_{FD} P$

lemma *div-free-is-*$lifelock\text{-}free_{SKIP}$: $lifelock\text{-}free_{SKIP}\ P \longleftrightarrow \mathcal{D}\ P = \{\}$
⟨*proof*⟩

lemma *lifelock-free-is-*$lifelock\text{-}free_{SKIP}$: *lifelock-free P* ⟹ $lifelock\text{-}free_{SKIP}\ P$
⟨*proof*⟩

corollary $deadlock\text{-}free_{SKIP}$*-is-*$lifelock\text{-}free_{SKIP}$: $deadlock\text{-}free_{SKIP}\ P \Longrightarrow lifelock\text{-}free_{SKIP}\ P$
⟨*proof*⟩

3.4 New Laws

lemma *non-terminating-Sync*:
‹*non-terminating P* ⟹ $lifelock\text{-}free_{SKIP}\ Q$ ⟹ *non-terminating* $(P\ [\![A]\!]\ Q)$›
⟨*proof*⟩

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

syntax
-writeS :: [$'b \Rightarrow 'a$, *pttrn*, *'b set*, *'a process*] => *'a process* ((*4*(-**!**-|-) /→ -)
[*0*,*0*,*50*,*78*] *50*)
translations
-writeS c p b P => *CONST Mndetprefix* (*c* ‘ {*p*. *b*}) (λ-. *P*)

end

Chapter 4

The MultiDet Operator

theory *MultiDet*
imports *Patch PreliminaryWork*
begin

4.1 Definition

definition *MultiDet* :: ‹[*'a set*, *'a* ⇒ *'b process*] ⇒ *'b process*›
where ‹*MultiDet A P* = *Finite-Set.fold* (λ*a r*. *r* □ *P a*) *STOP A*›

syntax *-MultiDet* :: ‹[*pttrn*, *'a set*, *'b process*] ⇒ *'b process*› (‹(*3*□-∈-. / -)› *75*)
translations □ *p* ∈ *A*. *P* ⇌ *CONST MultiDet A* (λ*p*. *P*)

4.2 First Properties

lemma *MultiDet-rec0*[*simp*]: ‹(□ *p* ∈ {}. *P p*) = *STOP*›
⟨*proof*⟩

lemma *MultiDet-rec1*[*simp*]: ‹(□ *p* ∈ {*a*}. *P p*) = *P a*›
⟨*proof*⟩

lemma *MultiDet-in-id*[*simp*]:
‹*a* ∈ *A* ⟹ (□ *p* ∈ *insert a A*. *P p*) = □ *p* ∈ *A*. *P p*›
⟨*proof*⟩

lemma *MultiDet-insert*[*simp*]:
‹*finite A* ⟹ (□ *p* ∈ *insert a A*. *P p*) = *P a* □ (□ *p* ∈ *A* − {*a*}. *P p*)›
⟨*proof*⟩

lemma *MultiDet-insert'*[*simp*]:

‹*finite* $A \Longrightarrow (\Box\ p \in$ *insert* $a\ A.\ P\ p) = (P\ a \Box (\Box\ p \in A.\ P\ p))$›
⟨*proof*⟩

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

4.3 Some Tests

lemma *test-MultiDet*: ‹$(\Box\ p \in \{1{::}int\ ..\ 3\}.\ P\ p) = P\ 1 \Box P\ 2 \Box P\ 3$›
⟨*proof*⟩

lemma *test-MultiDet′*:
‹$(\Box\ p \in \{0{::}nat\ ..\ a\}.\ P\ p) = (\Box\ p \in \{a\} \cup \{1\ ..\ a\} \cup \{0\}\ .\ P\ p)$›
⟨*proof*⟩

4.4 Continuity

lemma *MultiDet-cont*[*simp*]:
‹⟦*finite* A; $\forall x \in A.$ *cont* $(P\ x)$⟧ $\Longrightarrow$ *cont* $(\lambda y.\ \Box\ z{\in}A.\ P\ z\ y)$›
⟨*proof*⟩

4.5 Factorization of $(\Box)$ in front of *MultiDet*

lemma *MultiDet-factorization-union*:
‹⟦*finite* A; *finite* B⟧ $\Longrightarrow (\Box\ p \in A.\ P\ p) \Box (\Box\ p \in B.\ P\ p) = \Box\ p \in A \cup B\ .\ P\ p$›
⟨*proof*⟩

4.6 $\bot$ Absorbance

lemma *MultiDet-BOT-absorb*:
assumes *fin*: ‹*finite* A› **and** *bot*: ‹$P\ a = \bot$› **and** *dom*: ‹$a \in A$›
shows ‹$(\Box\ x \in A.\ P\ x) = \bot$›
⟨*proof*⟩

lemma *MultiDet-is-BOT-iff*:
‹*finite* $A \Longrightarrow$ *MultiDet* $A\ P = \bot \longleftrightarrow (\exists a \in A.\ P\ a = \bot)$›
⟨*proof*⟩

4.7 First Properties

lemma *MultiDet-id*: ‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow (\Box p \in A.\ P) = P$›
⟨*proof*⟩

lemma *MultiDet-STOP-id*: ‹*finite* $A \Longrightarrow (\Box p \in A.\ STOP) = STOP$›
⟨*proof*⟩

lemma *MultiDet-STOP-neutral*:
‹*finite* $A \Longrightarrow P\ a = STOP \Longrightarrow (\Box\ z \in insert\ a\ A.\ P\ z) = \Box\ z \in A.\ P\ z$›
⟨*proof*⟩

lemma *MultiDet-is-STOP-iff*:
‹*finite* $A \Longrightarrow (\Box\ a \in A.\ P\ a) = STOP \longleftrightarrow A = \{\} \vee (\forall\, a \in A.\ P\ a = STOP)$›
⟨*proof*⟩

4.8 Behaviour of *MultiDet* with (□)

lemma *MultiDet-Det*:
‹*finite* $A \Longrightarrow (\Box\ a \in A.\ P\ a)\ \Box\ (\Box\ a \in A.\ Q\ a) = \Box\ a \in A.\ P\ a\ \Box\ Q\ a$›
⟨*proof*⟩

4.9 Commutativity

lemma *MultiDet-sets-commute*:
‹$[\![$*finite* A; *finite* $B]\!] \Longrightarrow (\Box\ a \in A.\ \Box\ b \in B.\ P\ a\ b) = \Box\ b \in B.\ \Box\ a \in A.\ P\ a\ b$›
⟨*proof*⟩

4.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiDet*:
‹$[\![$*finite* A; *inj-on* $f\ A]\!] \Longrightarrow (\Box\ x \in A.\ P\ x) = \Box\ x \in f\ `\ A.\ P\ (inv\text{-}into\ A\ f\ x)$›
⟨*proof*⟩

4.11 The Projections

lemma *D-MultiDet*: ‹*finite* $A \Longrightarrow \mathcal{D}\ (\Box\ x \in A.\ P\ x) = (\bigcup\ p \in A.\ \mathcal{D}\ (P\ p))$›
⟨*proof*⟩

lemma *T-MultiDet*:
‹*finite* $A \Longrightarrow \mathcal{T}\ (\Box\ x \in A.\ P\ x) = (if\ A = \{\}\ then\ \{[]\}\ else \bigcup\ p \in A.\ \mathcal{T}\ (P\ p))$›
⟨*proof*⟩

4.12 Cartesian Product Results

lemma *MultiDet-cartprod-σs-set-σs-set*:

‹⟦*finite* A; *finite* B; $\forall s \in A$. *length* $s = len_1$⟧ $\Longrightarrow$
$(\square\ (s,\ t) \in A \times B.\ P\ (s\ @\ t)) = \square\ u \in \{s\ @\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *MultiDet-cartprod-s-set-σs-set*:
‹⟦*finite* A; *finite* B⟧ $\Longrightarrow$
$(\square\ (s,\ t) \in A \times B.\ P\ (s\ \#\ t)) = \square\ u \in \{s\ \#\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *MultiDet-cartprod-s-set-s-set*:
‹⟦*finite* A; *finite* B⟧ $\Longrightarrow$
$(\square\ (s,\ t) \in A \times B.\ P\ [s,\ t]) = \square\ u \in \{[s,\ t]\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *MultiDet-cartprod*:
‹*finite* $A \Longrightarrow$ *finite* $B \Longrightarrow (\square\ (s,\ t) \in A \times B.\ P\ s\ t) = \square\ s{\in}A.\ \square\ t{\in}B.\ P\ s\ t$›
⟨*proof*⟩

end

Chapter 5

The MultiNdet Operator

theory *MultiNdet*
  imports *Patch PreliminaryWork*
begin

5.1 Definition

Defining the multi operator of (⊓) requires more work than with (□) since there is no neutral element. We will first build a version on *'α list* that we will generalize to *'α set*.

fun *MultiNdet-list* :: ‹['a list, 'a ⇒ 'b process] ⇒ 'b process›
  where ‹*MultiNdet-list* [] *P* = *STOP*›
  | ‹*MultiNdet-list* (*a* # *l*) *P* = *fold* (λ*x r*. *r* ⊓ *P x*) *l* (*P a*)›

syntax -*MultiNdet-list* :: ‹[pttrn,'a set,'b process] ⇒ 'b process›
  (‹(3⨅$_l$ -∈-. / -)› 76)
translations ⨅$_l$ *p* ∈ *l*. *P* ⇌ *CONST MultiNdet-list l* (λ*p*. *P*)

interpretation *MultiNdet*: *comp-fun-idem* **where** *f*=‹λ*x r*. *r* ⊓ *P x*›
  ⟨*proof*⟩

lemma *MultiNdet-list-set*:
  ‹*set L* = *set L'* ⟹ *MultiNdet-list L P* = *MultiNdet-list L' P*›
  ⟨*proof*⟩

definition *MultiNdet* :: ‹['a set, 'a ⇒ 'b process] ⇒ 'b process›
  where ‹*MultiNdet A P* = *MultiNdet-list* (*SOME L*. *set L* = *A*) *P*›

syntax -*MultiNdet* :: ‹[pttrn, 'a set, 'b process] ⇒ 'b process› (‹(3⨅ -∈-. / -)› 76)

translations $\bigsqcap\ p \in A.\ P \rightleftharpoons$ *CONST MultiNdet A* $(\lambda p.\ P)$

5.2 First Properties

lemma *MultiNdet-rec0*[*simp*]: ‹$(\bigsqcap\ p \in \{\}.\ P\ p) = STOP$›
⟨*proof*⟩

lemma *MultiNdet-rec1*[*simp*]: ‹$(\bigsqcap\ p \in \{a\}.\ P\ p) = P\ a$›
⟨*proof*⟩

lemma *MultiNdet-in-id*[*simp*]:
‹$a \in A \Longrightarrow (\bigsqcap\ p \in insert\ a\ A.\ P\ p) = \bigsqcap\ p \in A.\ P\ p$›
⟨*proof*⟩

lemma *MultiNdet-insert*[*simp*]:
assumes *fin*: ‹*finite A*› **and** *notempty*: ‹$A \neq \{\}$› **and** *notin*: ‹$a \notin A$›
shows ‹$(\bigsqcap\ p \in insert\ a\ A.\ P\ p) = P\ a \sqcap (\bigsqcap\ p \in A.\ P\ p)$›
⟨*proof*⟩

lemma *MultiNdet-insert′*[*simp*]:
‹$[\![finite\ A;\ A \neq \{\}]\!] \Longrightarrow (\bigsqcap\ p \in insert\ a\ A.\ P\ p) = P\ a \sqcap (\bigsqcap\ p \in A.\ P\ p)$›
⟨*proof*⟩

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

5.3 Some Tests

lemma ‹$(\bigsqcap_l\ p \in [].\ P\ p) = STOP$›
and ‹$(\bigsqcap_l\ p \in [a].\ P\ p) = P\ a$›
and ‹$(\bigsqcap_l\ p \in [a,\ b].\ P\ p) = P\ a \sqcap P\ b$›
and ‹$(\bigsqcap_l\ p \in [a,\ b,\ c].\ P\ p) = P\ a \sqcap P\ b \sqcap P\ c$›
⟨*proof*⟩

lemma ‹$(\bigsqcap\ p \in \{\}.\ P\ p) = STOP$›
and ‹$(\bigsqcap\ p \in \{a\}.\ P\ p) = P\ a$›
and ‹$(\bigsqcap\ p \in \{a,\ b\}.\ P\ p) = P\ a \sqcap P\ b$›
and ‹$(\bigsqcap\ p \in \{a,\ b,\ c\}.\ P\ p) = P\ a \sqcap P\ b \sqcap P\ c$›
⟨*proof*⟩

lemma *test-MultiNdet*: ‹($\bigsqcap$ $p \in \{1::int\ ..\ 3\}$. $P\ p$) = $P\ 1 \sqcap P\ 2 \sqcap P\ 3$›
⟨*proof*⟩

lemma *test-MultiNdet'*:
‹($\bigsqcap$ $p \in \{0::nat\ ..\ a\}$. $P\ p$) = ($\bigsqcap$ $p \in \{a\} \cup \{1\ ..\ a\} \cup \{0\}$. $P\ p$)›
⟨*proof*⟩

5.4 Continuity

lemma *MultiNdet-cont*[*simp*]:
‹⟦*finite A*; $\forall x \in A$. *cont* ($P\ x$)⟧ $\Longrightarrow$ *cont* (λy. $\bigsqcap$ $z{\in}A$. $P\ z\ y$)›
⟨*proof*⟩

5.5 Factorization of ($\sqcap$) in front of *MultiNdet*

lemma *MultiNdet-factorization-union*:
‹⟦$A \neq \{\}$; *finite A*; $B \neq \{\}$; *finite B*⟧ $\Longrightarrow$
($\bigsqcap$ $p \in A$. $P\ p$) $\sqcap$ ($\bigsqcap$ $p \in B$. $P\ p$) = $\bigsqcap$ $p \in A \cup B$. $P\ p$›
⟨*proof*⟩

5.6 $\perp$ Absorbance

lemma *MultiNdet-BOT-absorb*:
assumes *fin*: ‹*finite A*› **and** *bot*: ‹$P\ a = \perp$› **and** *dom*: ‹$a \in A$›
shows ‹($\bigsqcap$ $x \in A$. $P\ x$) = $\perp$›
⟨*proof*⟩

lemma *MultiNdet-is-BOT-iff*:
‹*finite A* $\Longrightarrow$ ($\bigsqcap$ $p \in A$. $P\ p$) = $\perp \longleftrightarrow (\exists a \in A.\ P\ a = \perp)$›
⟨*proof*⟩

5.7 First Properties

lemma *MultiNdet-id*: ‹$A \neq \{\}$ $\Longrightarrow$ *finite A* $\Longrightarrow$ ($\bigsqcap$ $p \in A$. P) = P›
⟨*proof*⟩

lemma *MultiNdet-STOP-id*: ‹*finite A* $\Longrightarrow$ ($\bigsqcap$ $p \in A$. *STOP*) = *STOP*›
⟨*proof*⟩

lemma *MultiNdet-is-STOP-iff*:
‹*finite A* $\Longrightarrow$ ($\bigsqcap$ $p \in A$. $P\ p$) = *STOP* $\longleftrightarrow$ $A = \{\} \vee (\forall a \in A.\ P\ a = STOP)$›
⟨*proof*⟩

5.8 Behaviour of *MultiNdet* with $(\sqcap)$

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

5.9 Commutativity

lemma *MultiNdet-sets-commute*:
‹$[\![finite\ A;\ finite\ B]\!] \Longrightarrow$
$(\bigsqcap\ a \in A.\ \bigsqcap\ b \in B.\ P\ a\ b) = \bigsqcap\ b \in B.\ \bigsqcap\ a \in A.\ P\ a\ b$›
⟨*proof*⟩

5.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiNdet*:
‹$[\![finite\ A;\ inj\text{-}on\ f\ A]\!] \Longrightarrow (\bigsqcap\ x \in A.\ P\ x) = \bigsqcap\ x \in f\ `\ A.\ P\ (inv\text{-}into\ A\ f\ x)$›
⟨*proof*⟩

5.11 The Projections

lemma *D-MultiNdet*: ‹$finite\ A \Longrightarrow \mathcal{D}\ (\bigsqcap\ x \in A.\ P\ x) = (\bigcup\ p \in A.\ \mathcal{D}\ (P\ p))$›
⟨*proof*⟩

lemma *F-MultiNdet*:
‹$finite\ A \Longrightarrow \mathcal{F}\ (\bigsqcap\ x \in A.\ P\ x) =$
$(if\ A = \{\}\ then\ \{(s,\ X).\ s = []\}\ else \bigcup\ p \in A.\ \mathcal{F}\ (P\ p))$›
⟨*proof*⟩

lemma *T-MultiNdet*:
‹$finite\ A \Longrightarrow \mathcal{T}\ (\bigsqcap\ x \in A.\ P\ x) =$
$(if\ A = \{\}\ then\ \{[]\}\ else \bigcup\ p \in A.\ \mathcal{T}\ (P\ p))$›
⟨*proof*⟩

5.12 Cartesian Product Results

lemma *MultiNdet-cartprod-σs-set-σs-set*:

‹$[\![finite\ A;\ finite\ B;\ \forall\, s \in A.\ length\ s = len_1]\!] \Longrightarrow$
$(\bigsqcap\ (s,\ t) \in A \times B.\ P\ (s\ @\ t)) = \bigsqcap\ u \in \{s\ @\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *MultiNdet-cartprod-s-set-σs-set*:
‹$[\![finite\ A;\ finite\ B]\!] \Longrightarrow$
$(\bigsqcap\ (s,\ t) \in A \times B.\ P\ (s\ \#\ t)) = \bigsqcap\ u \in \{s\ \#\ t\ |s\ t.\ (s,\ t) \in A \times B\}.\ P\ u$›
⟨*proof*⟩

lemma *MultiNdet-cartprod-s-set-s-set*:
‹⟦*finite* A; *finite* B⟧ $\Longrightarrow$
($\bigsqcap$ $(s, t) \in A \times B$. P $[s, t]$) $=$ $\bigsqcap$ $u \in \{[s, t] \mid s\ t.\ (s, t) \in A \times B\}$. P u›
⟨*proof*⟩

lemma *MultiNdet-cartprod*:
‹⟦*finite* A; *finite* B⟧ $\Longrightarrow$ ($\bigsqcap$ $(s, t) \in A \times B$. P s t) $=$ $\bigsqcap$ $s{\in}A$. $\bigsqcap$ $t{\in}B$. P s t›
⟨*proof*⟩

end

Chapter 6

The MultiSync Operator

theory *MultiSync*
  imports *HOL−Library.Multiset PreliminaryWork Patch*
begin

6.1 Definition

As in the (⊓) case, we have no neutral element so we will also have to go through lists first. But the binary operator *Sync* is not idempotent either, so the generalization will be done on *′α multiset* and not on *′α set*.

Note that a *′α multiset* is by construction finite (cf. theorem *finite* (*set-mset* *M*)).

fun *MultiSync-list* :: ‹[*′b set*, *′a list*, *′a* ⇒ *′b process*] ⇒ *′b process*›
  where ‹*MultiSync-list S* [] *P* = *STOP*›
  | ‹*MultiSync-list S* (*l* # *L*) *P* = *fold* (λ*x r*. *r* ⟦*S*⟧ *P x*) *L* (*P l*)›

syntax *-MultiSync-list* :: ‹[*pttrn*, *′b set*, *′a list*, *′b process*] ⇒ *′b process*›
  (‹(*3*⟦-⟧$_l$-∈-./ -)› *63*)
translations ⟦*S*⟧$_l$ *p* ∈ *L*. *P* ⇌ *CONST MultiSync-list S L* (λ*p*. *P*)

interpretation *MultiSync*: *comp-fun-commute* **where** *f* = ‹λ*x r*. *r* ⟦*E*⟧ *P x*›
  ⟨*proof*⟩

lemma *MultiSync-list-mset*:
  ‹*mset L* = *mset L′* ⟹ *MultiSync-list S L P* = *MultiSync-list S L′ P*›
  ⟨*proof*⟩

definition *MultiSync* :: ‹[*′b set*, *′a multiset*, *′a* ⇒ *′b process*] ⇒ *′b process*›
  where ‹*MultiSync S M P* = *MultiSync-list S* (*SOME L*. *mset L* = *M*) *P*›

syntax *-MultiSync* :: ‹[*pttrn*,′*b set*,′*a multiset*,′*b process*] ⇒ ′*b process*›
(‹(*3*⟦-⟧ -∈#-. / -)› *63*)
translations ⟦*S*⟧ *p* ∈# *M*. *P* ⇌ *CONST MultiSync S M* (*λp*. *P*)

Special case of *MultiSync E P* when *E* = {}.

abbreviation *MultiInter* :: ‹[′*a multiset*, ′*a* ⇒ ′*b process*] ⇒ ′*b process*›
where ‹*MultiInter M P* ≡ *MultiSync* {} *M P*›

syntax *-MultiInter* :: ‹[*pttrn*, ′*a multiset*, ′*b process*] ⇒ ′*b process*›
(‹(*3*||| -∈#-. / -)› *77*)
translations ||| *p* ∈# *M*. *P* ⇌ *CONST MultiInter M* (*λp*. *P*)

Special case of *MultiSync E P* when *E* = *UNIV*.

abbreviation *MultiPar* :: ‹[′*a multiset*, ′*a* ⇒ ′*b process*] ⇒ ′*b process*›
where ‹*MultiPar M P* ≡ *MultiSync UNIV M P*›

syntax *-MultiPar* :: ‹[*pttrn*, ′*a multiset*, ′*b process*] ⇒ ′*b process*›
(‹(*3*|| -∈#-. / -)› *77*)
translations || *p* ∈# *M*. *P* ⇌ *CONST MultiPar M* (*λp*. *P*)

6.2 First Properties

lemma *MultiSync-rec0*[*simp*]: ‹(⟦*S*⟧ *p* ∈# {#}. *P p*) = *STOP*›
⟨*proof*⟩

lemma *MultiSync-rec1*[*simp*]: ‹(⟦*S*⟧ *p* ∈# {#*a*#}. *P p*) = *P a*›
⟨*proof*⟩

lemma *MultiSync-add*[*simp*]:
‹*M* ≠ {#} ⟹ (⟦*S*⟧ *p* ∈# *add-mset m M*. *P p*) = *P m* ⟦*S*⟧ (⟦*S*⟧ *p* ∈# *M*. *P p*)›
⟨*proof*⟩

lemma *mono-MultiSync-eq*:
‹∀ *x* ∈# *M*. *P x* = *Q x* ⟹ *MultiSync S M P* = *MultiSync S M Q*›
⟨*proof*⟩

lemmas *MultiInter-rec0* = *MultiSync-rec0*[**where** *S* = ‹{}›]
and *MultiPar-rec0* = *MultiSync-rec0*[**where** *S* = ‹*UNIV*›]
and *MultiInter-rec1* = *MultiSync-rec1*[**where** *S* = ‹{}›]
and *MultiPar-rec1* = *MultiSync-rec1*[**where** *S* = ‹*UNIV*›]
and *MultiInter-add* = *MultiSync-add*[**where** *S* = ‹{}›]
and *MultiPar-add* = *MultiSync-add*[**where** *S* = ‹*UNIV*›]
and *mono-MultiInter-eq* = *mono-MultiSync-eq*[**where** *S* = ‹{}›]
and *mono-MultiPar-eq* = *mono-MultiSync-eq*[**where** *S* = ‹*UNIV*›]

6.3 Some Tests

lemma ‹(⟦S⟧$_l$ *p* ∈ []. *P p*) = *STOP*›
and ‹(⟦S⟧$_l$ *p* ∈ [*a*]. *P p*) = *P a*›
and ‹(⟦S⟧$_l$ *p* ∈ [*a*, *b*]. *P p*) = *P a* ⟦S⟧ *P b*›
and ‹(⟦S⟧$_l$ *p* ∈ [*a*, *b*, *c*]. *P p*) = *P a* ⟦S⟧ *P b* ⟦S⟧ *P c*›
⟨*proof*⟩

lemma *test-MultiSync*:
‹(⟦S⟧ *p* ∈# *mset* []. *P p*) = *STOP*›
‹(⟦S⟧ *p* ∈# *mset* [*a*]. *P p*) = *P a*›
‹(⟦S⟧ *p* ∈# *mset* [*a*, *b*]. *P p*) = *P a* ⟦S⟧ *P b*›
‹(⟦S⟧ *p* ∈# *mset* [*a*, *b*, *c*]. *P p*) = *P a* ⟦S⟧ *P b* ⟦S⟧ *P c*›
⟨*proof*⟩

lemma *MultiSync-set1*: ‹*MultiSync S* (*mset-set* {*k*::*nat*..<*k*}) *P* = *STOP*›
⟨*proof*⟩

lemma *MultiSync-set2*: ‹*MultiSync S* (*mset-set* {*k*..<*Suc k*}) *P* = *P k*›
⟨*proof*⟩

lemma *MultiSync-set3*:
‹*l* < *k* ⟹ *MultiSync S* (*mset-set* {*l* ..< *Suc k*}) *P* =
P l ⟦S⟧ (*MultiSync S* (*mset-set* {*Suc l* ..< *Suc k*}) *P*)›
⟨*proof*⟩

lemma *test-MultiSync'*:
‹(⟦S⟧ *p* ∈# *mset-set* {*1*::*int* .. *3*}. *P p*) = *P 1* ⟦S⟧ *P 2* ⟦S⟧ *P 3*›
⟨*proof*⟩

lemma *test-MultiSync''*:
‹(⟦S⟧ *p* ∈# *mset-set* {*0*::*nat* .. *a*}. *P p*) =
⟦S⟧ *p* ∈# *mset-set* ({*a*} ∪ {*1* .. *a*} ∪ {*0*}) . *P p*›
⟨*proof*⟩

lemmas *test-MultiInter* = *test-MultiSync*[**where** *S* = ‹{}›]
and *test-MultiPar* = *test-MultiSync*[**where** *S* = ‹*UNIV*›]
and *MultiInter-set1* = *MultiSync-set1*[**where** *S* = ‹{}›]
and *MultiPar-set1* = *MultiSync-set1*[**where** *S* = ‹*UNIV*›]
and *MultiInter-set2* = *MultiSync-set2*[**where** *S* = ‹{}›]
and *MultiPar-set2* = *MultiSync-set2*[**where** *S* = ‹*UNIV*›]

and *MultiInter-set3* = *MultiSync-set3*[**where** S = ‹{}›]
and *MultiPar-set3* = *MultiSync-set3*[**where** S = ‹*UNIV*›]
and *test-MultiInter′* = *test-MultiSync′*[**where** S = ‹{}›]
and *test-MultiPar′* = *test-MultiSync′*[**where** S = ‹*UNIV*›]
and *test-MultiInter″* = *test-MultiSync″*[**where** S = ‹{}›]
and *test-MultiPar″* = *test-MultiSync″*[**where** S = ‹*UNIV*›]

6.4 Continuity

lemma *MultiSync-cont*[*simp*]:
‹$\forall x \in\# M.\ cont\ (P\ x) \Longrightarrow cont\ (\lambda y.\ [\![S]\!]\ z \in\# M.\ P\ z\ y)$›
⟨*proof*⟩

lemmas *MultiInter-cont*[*simp*] = *MultiSync-cont*[**where** S = ‹{}›]
and *MultiPar-cont*[*simp*] = *MultiSync-cont*[**where** S = ‹*UNIV*›]

6.5 Factorization of *Sync* in front of *MultiSync*

lemma *MultiSync-factorization-union*:
‹$[\![M \neq \{\#\};\ N \neq \{\#\}]\!] \Longrightarrow$
$([\![S]\!]\ z \in\# M.\ P\ z)\ [\![S]\!]\ ([\![S]\!]\ z \in\# N.\ P\ z) = [\![S]\!]\ z {\in\#}\ M + N.\ P\ z$›
⟨*proof*⟩

lemmas *MultiInter-factorization-union* =
MultiSync-factorization-union[**where** S = ‹{}›]
and *MultiPar-factorization-union* =
MultiSync-factorization-union[**where** S = ‹*UNIV*›]

6.6 ⊥ Absorbance

lemma *MultiSync-BOT-absorb*:
‹$m \in\# M \Longrightarrow P\ m = \bot \Longrightarrow ([\![S]\!]\ z \in\# M.\ P\ z) = \bot$›
⟨*proof*⟩

lemmas *MultiInter-BOT-absorb* = *MultiSync-BOT-absorb*[**where** S = ‹{}›]
and *MultiPar-BOT-absorb* = *MultiSync-BOT-absorb*[**where** S = ‹*UNIV*›]

lemma *MultiSync-is-BOT-iff*:
‹$([\![S]\!]\ m \in\# M.\ P\ m) = \bot \longleftrightarrow (\exists\, m \in\# M.\ P\ m = \bot)$›
⟨*proof*⟩

lemmas *MultiInter-is-BOT-iff* = *MultiSync-is-BOT-iff*[**where** S = ‹{}›]
and *MultiPar-is-BOT-iff* = *MultiSync-is-BOT-iff*[**where** S = ‹*UNIV*›]

6.7 Other Properties

lemma *MultiSync-SKIP-id*: $\langle M \neq \{\#\} \Longrightarrow (\llbracket S \rrbracket\ z \in\#\ M.\ SKIP) = SKIP \rangle$
$\langle proof \rangle$

lemmas *MultiInter-SKIP-id* = *MultiSync-SKIP-id*[**where** $S = \langle\{\}\rangle$]
and *MultiPar-SKIP-id* = *MultiSync-SKIP-id*[**where** $S = \langle UNIV \rangle$]

lemma *MultiPar-prefix-two-distincts-STOP*:
assumes $\langle m \in\#\ M \rangle$ **and** $\langle m' \in\#\ M \rangle$ **and** $\langle fst\ m \neq fst\ m' \rangle$
shows $\langle (||\ a \in\#\ M.\ (fst\ a \rightarrow P\ (snd\ a))) = STOP \rangle$
$\langle proof \rangle$

lemma *MultiPar-prefix-two-distincts-STOP'*:
$\langle \llbracket (m,\ n) \in\#\ M;\ (m',\ n') \in\#\ M;\ m \neq m' \rrbracket \Longrightarrow$
$(||\ (m,\ n) \in\#\ M.\ (m \rightarrow P\ n)) = STOP \rangle$
$\langle proof \rangle$

6.8 Behaviour of *MultiSync* with *Sync*

lemma *Sync-STOP-STOP*: $\langle STOP\ \llbracket S \rrbracket\ STOP = STOP \rangle$
$\langle proof \rangle$

lemma *MultiSync-Sync*:
$\langle (\llbracket S \rrbracket\ z \in\#\ M.\ P\ z)\ \llbracket S \rrbracket\ (\llbracket S \rrbracket\ z \in\#\ M.\ P'\ z) = \llbracket S \rrbracket\ z \in\#\ M.\ P\ z\ \llbracket S \rrbracket\ P'\ z \rangle$
$\langle proof \rangle$

lemmas *MultiInter-Inter* = *MultiSync-Sync*[**where** $S = \langle\{\}\rangle$]
and *MultiPar-Par* = *MultiSync-Sync*[**where** $S = \langle UNIV \rangle$]

6.9 Commutativity

lemma *MultiSync-sets-commute*:
$\langle (\llbracket S \rrbracket\ a \in\#\ M.\ \llbracket S \rrbracket\ b \in\#\ N.\ P\ a\ b) = \llbracket S \rrbracket\ b \in\#\ N.\ \llbracket S \rrbracket\ a \in\#\ M.\ P\ a\ b \rangle$
$\langle proof \rangle$

lemmas *MultiInter-sets-commute* = *MultiSync-sets-commute*[**where** $S = \langle\{\}\rangle$]
and *MultiPar-sets-commute* = *MultiSync-sets-commute*[**where** $S = \langle UNIV \rangle$]

6.10 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSync*:

‹*inj-on f* (*set-mset M*) $\Longrightarrow$
($[\![S]\!]$ $x \in\#$ M. P x) = $[\![S]\!]$ $x \in\#$ *image-mset f M*. P (*inv-into* (*set-mset M*) *f x*)›
⟨*proof*⟩

lemmas *inj-on-mapping-over-MultiInter* =
inj-on-mapping-over-MultiSync[**where** S = ‹{}›]
and *inj-on-mapping-over-MultiPar* =
inj-on-mapping-over-MultiSync[**where** S = ‹*UNIV*›]

end

Chapter 7

The MultiSeq Operator

theory *MultiSeq*
imports *Patch*
begin

7.1 Definition

definition *MultiSeq* :: ‹[′*a list*, ′*a* ⇒ ′*b process*] ⇒ ′*b process*›
where ‹*MultiSeq S P* = *foldr* (λ*a r*. (*P a*) **;** *r*) *S SKIP*›

syntax -*MultiSeq* :: ‹[*pttrn*,′*a list*, ′*b process*] ⇒ ′*b process*›
(‹(*3SEQ* -∈@-./ -)› *73*)
translations *SEQ i* ∈@ *A*. *P* ⇌ *CONST MultiSeq A* (λ*i*. *P*)

7.2 First Properties

lemma *MultiSeq-rec0*[*simp*]: ‹(*SEQ p* ∈@ []. *P p*) = *SKIP*›
⟨*proof*⟩

lemma *MultiSeq-rec1*[*simp*]: ‹(*SEQ p* ∈@ [*a*]. *P p*) = *P a*›
⟨*proof*⟩

lemma *MultiSeq-Cons*[*simp*]: ‹(*SEQ i* ∈@ *a* # *L*. *P i*) = *P a* **;** (*SEQ i* ∈@ *L*. *P i*)›
⟨*proof*⟩

7.3 Some Tests

lemma ‹(*SEQ p* ∈@ []. *P p*) = *SKIP*›
and ‹(*SEQ p* ∈@ [*a*]. *P p*) = *P a*›
and ‹(*SEQ p* ∈@ [*a*,*b*]. *P p*) = *P a* **;** *P b*›

and ‹(*SEQ p* ∈@ [*a*,*b*,*c*]. *P p*) = *P a* **;** *P b* **;** *P c*›
⟨*proof*⟩

lemma *test-MultiSeq*: ‹(*SEQ p* ∈@ [*1*::*int* .. *3*]. *P p*) = *P 1* **;** *P 2* **;** *P 3*›
⟨*proof*⟩

7.4 Continuity

lemma *MultiSeq-cont*[*simp*]:
‹∀ *x* ∈ *set L*. *cont* (*P x*) ⟹ *cont* (λ*y*. *SEQ z* ∈@ *L*. *P z y*)›
⟨*proof*⟩

7.5 Factorization of (;) in front of *MultiSeq*

lemma *MultiSeq-factorization-append*:
‹(*SEQ p* ∈@ *A*. *P p*) **;** (*SEQ p* ∈@ *B*. *P p*) = (*SEQ p* ∈@ *A* @ *B* . *P p*)›
⟨*proof*⟩

7.6 ⊥ Absorbance

lemma *MultiSeq-BOT-absorb*:
‹*P a* = ⊥ ⟹ (*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = (*SEQ z* ∈@ *l1*. *P z*) **;** ⊥›
⟨*proof*⟩

7.7 First Properties

lemma *MultiSeq-SKIP-neutral*:
‹*P a* = *SKIP* ⟹ (*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = *SEQ z* ∈@ *l1* @ *l2*. *P z*›
⟨*proof*⟩

lemma *MultiSeq-STOP-absorb*:
‹*P a* = *STOP* ⟹
(*SEQ z* ∈@ *l1* @ [*a*] @ *l2*. *P z*) = (*SEQ z* ∈@ *l1*. *P z*) **;** *STOP*›
⟨*proof*⟩

lemma *mono-MultiSeq-eq*:
‹∀ *x* ∈ *set L*. *P x* = *Q x* ⟹ *MultiSeq L P* = *MultiSeq L Q*›
⟨*proof*⟩

lemma *MultiSeq-is-SKIP-iff*:
‹*MultiSeq L P* = *SKIP* ⟷ (∀ *a* ∈ *set L*. *P a* = *SKIP*)›
⟨*proof*⟩

7.8 Commutativity

Of course, since the sequential composition *P* **;** *Q* is not commutative, the result here is negative: the order of the elements of list *L* does matter in *SEQ z∈@L. P z*.

7.9 Behaviour with Injectivity

lemma *inj-on-mapping-over-MultiSeq*:
‹*inj-on f* (*set C*) ⟹
(*SEQ x* ∈@ *C*. *P x*) = *SEQ x* ∈@ *map f C*. *P* (*inv-into* (*set C*) *f x*)›
⟨*proof*⟩

7.10 Definition of *first-elem*

primrec *first-elem* :: ‹[′α ⇒ *bool*, ′α *list*] ⇒ *nat*›
where ‹*first-elem P* [] = *0*›
| ‹*first-elem P* (*x* # *L*) = (*if P x then 0 else Suc* (*first-elem P L*))›

first-elem returns the first index *i* such that *P* (*L* ! *i*) = *True* if it exists, *length L* otherwise.

This will be very useful later.

value ‹*first-elem* (λ*x*. *4* < *x*) [*0*::*nat*, *2*, *5*]›
lemma ‹*first-elem* (λ*x*. *5* < *x*) [*0*::*nat*, *2*, *5*] = *3*› ⟨*proof*⟩
lemma ‹*P* ‘ *set L* ⊆ {*False*} ⟹ *first-elem P L* = *length L*› ⟨*proof*⟩

end

Chapter 8

The Global Non-Deterministic Choice

theory *GlobalNdet*
  imports *MultiNdet*
begin

8.1 General Non-Deterministic Choice Definition

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

lemma ‹$\nexists P.\ \forall Q.\ (P :: {}'\alpha\ process) \sqcap Q = Q$›
⟨*proof*⟩

lift-definition *GlobalNdet* :: ‹$[{}'\alpha\ set,\ {}'\alpha \Rightarrow {}'\beta\ process] \Rightarrow {}'\beta\ process$›
  is ‹$\lambda A\ P.\quad if\ A = \{\}$
    $then\ (\{(s, X).\ s = []\},\ \{\})$
    $else\ (\bigcup a{\in}A.\ \mathcal{F}\ (P\ a),\ \bigcup a{\in}A.\ \mathcal{D}\ (P\ a))$›
⟨*proof*⟩

syntax *-GlobalNdet* :: ‹$[pttrn,{}'a\ set,{}'b\ process] \Rightarrow {}'b\ process$› (‹(3⊓ -∈-. / -)› *76*)
translations $\sqcap\ p \in A.\ P \rightleftharpoons CONST\ GlobalNdet\ A\ (\lambda p.\ P)$

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

It is also subtly different from the multi-non-deterministic choice $\bigsqcap\ p{\in}A.\ P\ p$ which is only defined when A is *finite*.

lemma *empty-GlobalNdet*[*simp*] : ‹$GlobalNdet\ \{\}\ P = STOP$›
⟨*proof*⟩

8.2 The Projections

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

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

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

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

lemma *mono-GlobalNdet-eq2*:
‹$\forall\, x \in A.\ P\ (f\ x) = Q\ x \Longrightarrow GlobalNdet\ (f\ `\ A)\ P = GlobalNdet\ A\ Q$›
⟨*proof*⟩

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

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

8.4 ⊥ Absorbance

lemma *GlobalNdet-BOT-absorb*: ‹$P\ a = \bot \Longrightarrow a \in A \Longrightarrow (\sqcap\ x \in A.\ P\ x) = \bot$›
⟨*proof*⟩

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

8.5 First Properties

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

lemma *GlobalNdet-STOP-id*: ‹$(\sqcap\ p \in A.\ STOP) = STOP$›
⟨*proof*⟩

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

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

8.6 Behaviour of *GlobalNdet* with ($\sqcap$)

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

8.7 Commutativity

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

8.8 Behaviour with Injectivity

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

8.9 Cartesian Product Results

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

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

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

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

8.10 Link with *MultiNdet*

This operator is in fact an extension of *MultiNdet* to arbitrary sets: when A is *finite*, we have $\sqcap\ a{\in}A.\ \ P\ a = \bigsqcap\ a{\in}A.\ \ P\ a.$

lemma *finite-GlobalNdet-is-MultiNdet*:
‹$finite\ A \Longrightarrow (\sqcap\ p \in A.\ P\ p) = \bigsqcap\ p \in A.\ P\ p$›
⟨*proof*⟩

We obtain immediately the continuity when A is *finite* (and this is a necessary hypothesis for continuity).

lemma *GlobalNdet-cont*[*simp*]:
‹$[\![finite\ A;\ \forall x.\ cont\ (f\ x)]\!] \Longrightarrow cont\ (\lambda y.\ (\sqcap\ z \in A.\ (f\ z\ y)))$›
⟨*proof*⟩

8.11 Link with *Mndetprefix*

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

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

lemma *write0-GlobalNdet*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ x{\in}A.\ (a \rightarrow P\ x)) = (a \rightarrow (\sqcap\ x{\in}A.\ P\ x))$›
⟨*proof*⟩

8.12 Properties

lemma *GlobalNdet-Det*:
‹$A \neq \{\} \Longrightarrow (\sqcap\ a \in A.\ P\ a)\ \Box\ Q = \sqcap\ a \in A.\ P\ a\ \Box\ Q$›
⟨*proof*⟩

lemma *Mndetprefix-STOP*: ‹$A \subseteq C \Longrightarrow (\sqcap\ a \in A \rightarrow P\ a)\ [\![C]\!]\ STOP = STOP$›
⟨*proof*⟩

lemma *GlobalNdet-Sync-distr*:

‹$A \neq \{\} \Longrightarrow (\sqcap\ x \in A.\ P\ x)\ [\![\ C\ ]\!]\ Q = \sqcap\ x \in A.\ (P\ x\ [\![\ C\ ]\!]\ Q)$›
⟨*proof*⟩

lemma *Mndetprefix-Mprefix-Sync-distr*:
‹$[\![A \subseteq B;\ B \subseteq C]\!] \Longrightarrow (\sqcap\ a \in A \to P\ a)\ [\![\ C\ ]\!]\ (\square\ b \in B \to Q\ b) = \sqcap\ a \in A \to (P\ a\ [\![\ C\ ]\!]\ Q\ a)$›
— does not hold in general when $A \subseteq C$
⟨*proof*⟩

corollary *Mndetprefix-Mprefix-Par-distr*:
‹$A \subseteq B \Longrightarrow ((\sqcap\ a \in A \to P\ a) \mid\mid (\square\ b \in B \to Q\ b)) = \sqcap\ a \in A \to P\ a \mid\mid Q\ a$›
⟨*proof*⟩

lemma *Mndetprefix-Sync-Det-distr*:
‹$(\sqcap\ a \in A \to (P\ a\ [\![\ C\ ]\!]\ (\sqcap\ b \in B \to Q\ b)))\ \square$
$(\sqcap\ b \in B \to ((\sqcap\ a \in A \to P\ a)\ [\![\ C\ ]\!]\ Q\ b))$
$\sqsubseteq_{FD} (\sqcap\ a \in A \to P\ a)\ [\![\ C\ ]\!]\ (\sqcap\ b \in B \to Q\ b)$›
if *set-hyps* : ‹$A \neq \{\}$› ‹$B \neq \{\}$› ‹$A \cap C = \{\}$› ‹$B \cap C = \{\}$›
— both surprising: equality does not hold + deterministic choice
⟨*proof*⟩

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

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

end

Chapter 9

CSPM

theory *CSPM*
imports *MultiDet MultiNdet MultiSync MultiSeq GlobalNdet HOL−CSP.Assertions*
begin

From the binary laws of HOL-CSP, we immediately obtain refinement results and lemmas about the combination of multi-operators.

9.1 Refinements Results

lemma *mono-MultiDet-F*:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_F$ *MultiDet A Q*›
⟨*proof*⟩

lemma *mono-MultiDet-D*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_D$ *MultiDet A Q*›
and *mono-MultiDet-T*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_T$ *MultiDet A Q*›
and *mono-MultiDet-DT*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_{DT}$ *MultiDet A Q*›
and *mono-MultiDet-FD*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow$ *MultiDet A P* $\sqsubseteq_{FD}$ *MultiDet A Q*›
⟨*proof*⟩

lemma *mono-MultiNdet-F*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_F$ *MultiNdet A Q*›
and *mono-MultiNdet-D*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_D$ *MultiNdet A Q*›
and *mono-MultiNdet-T*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_T$ *MultiNdet A Q*›
and *mono-MultiNdet-DT*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow \forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow$ *MultiNdet A P* $\sqsubseteq_{DT}$ *MultiNdet A Q*›

and *mono-MultiNdet-FD*[*simp*, *elim*]:
‹*finite* $A \Longrightarrow \forall\, x \in A.\ P\ x \sqsubseteq_{FD}\ Q\ x \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_{FD}$ *MultiNdet* $A\ Q$›
⟨*proof*⟩

lemma *mono-MultiNdet-F-single*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow \forall\, a \in A.\ P \sqsubseteq_F\ Q\ a \Longrightarrow P \sqsubseteq_F$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-D-single*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow \forall\, a \in A.\ P \sqsubseteq_D\ Q\ a \Longrightarrow P \sqsubseteq_D$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-T-single*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow \forall\, a \in A.\ P \sqsubseteq_T\ Q\ a \Longrightarrow P \sqsubseteq_T$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-DT-single*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow \forall\, a \in A.\ P \sqsubseteq_{DT}\ Q\ a \Longrightarrow P \sqsubseteq_{DT}$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-FD-single*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow \forall\, a \in A.\ P \sqsubseteq_{FD}\ Q\ a \Longrightarrow P \sqsubseteq_{FD}$ *MultiNdet* $A\ Q$›
⟨*proof*⟩

lemma
assumes ‹$A \neq \{\}$› **and** ‹*finite* B› **and** ‹$A \subseteq B$›
shows *mono-MultiNdet-F-left-absorb-subset*:
‹$\forall\, x \in A.\ P\ x \sqsubseteq_F\ Q\ x \Longrightarrow$ *MultiNdet* $B\ P \sqsubseteq_F$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-D-left-absorb-subset*:
‹$\forall\, x \in A.\ P\ x \sqsubseteq_D\ Q\ x \Longrightarrow$ *MultiNdet* $B\ P \sqsubseteq_D$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-T-left-absorb-subset*:
‹$\forall\, x \in A.\ P\ x \sqsubseteq_T\ Q\ x \Longrightarrow$ *MultiNdet* $B\ P \sqsubseteq_T$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-FD-left-absorb-subset*:
‹$\forall\, x \in A.\ P\ x \sqsubseteq_{FD}\ Q\ x \Longrightarrow$ *MultiNdet* $B\ P \sqsubseteq_{FD}$ *MultiNdet* $A\ Q$›
and *mono-MultiNdet-DT-left-absorb-subset*:
‹$\forall\, x \in A.\ P\ x \sqsubseteq_{DT}\ Q\ x \Longrightarrow$ *MultiNdet* $B\ P \sqsubseteq_{DT}$ *MultiNdet* $A\ Q$›
⟨*proof*⟩

corollary *mono-MultiNdet-F-left-absorb*[*simp*]:
‹*finite* $A \Longrightarrow x \in A \Longrightarrow P\ x \sqsubseteq_F\ Q \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_F\ Q$›
and *mono-MultiNdet-D-left-absorb*[*simp*]:
‹*finite* $A \Longrightarrow x \in A \Longrightarrow P\ x \sqsubseteq_D\ Q \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_D\ Q$›
and *mono-MultiNdet-T-left-absorb*[*simp*]:
‹*finite* $A \Longrightarrow x \in A \Longrightarrow P\ x \sqsubseteq_T\ Q \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_T\ Q$›
and *mono-MultiNdet-FD-left-absorb*[*simp*]:
‹*finite* $A \Longrightarrow x \in A \Longrightarrow P\ x \sqsubseteq_{FD}\ Q \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_{FD}\ Q$›
and *mono-MultiNdet-DT-left-absorb*[*simp*]:
‹*finite* $A \Longrightarrow x \in A \Longrightarrow P\ x \sqsubseteq_{DT}\ Q \Longrightarrow$ *MultiNdet* $A\ P \sqsubseteq_{DT}\ Q$›
⟨*proof*⟩

lemma *mono-MultiNdet-MultiDet-F*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow$ *MultiNdet A P* $\sqsubseteq_F$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-D*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow$ *MultiNdet A P* $\sqsubseteq_D$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-T*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow$ *MultiNdet A P* $\sqsubseteq_T$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-FD*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow$ *MultiNdet A P* $\sqsubseteq_{FD}$ *MultiDet A P*›
and *mono-MultiNdet-MultiDet-DT*[*simp*, *elim*]:
‹*finite A* $\Longrightarrow$ *MultiNdet A P* $\sqsubseteq_{DT}$ *MultiDet A P*›
⟨*proof*⟩

lemma *mono-MultiSync-F*: ‹$\forall x \in\# M$. *P x* $\sqsubseteq_F$ *Q x* $\Longrightarrow$ *MultiSync S M P* $\sqsubseteq_F$ *MultiSync S M Q*›
⟨*proof*⟩

lemma *mono-MultiSync-D*: ‹$\forall x \in\# M$. *P x* $\sqsubseteq_D$ *Q x* $\Longrightarrow$ *MultiSync S M P* $\sqsubseteq_D$ *MultiSync S M Q*›
⟨*proof*⟩

lemma *mono-MultiSync-T*: ‹$\forall x \in\# M$. *P x* $\sqsubseteq_T$ *Q x* $\Longrightarrow$ *MultiSync S M P* $\sqsubseteq_T$ *MultiSync S M Q*›
⟨*proof*⟩

lemma *mono-MultiSync-DT*[*simp*, *elim*]:
‹$\forall x \in\# M$. *P x* $\sqsubseteq_{DT}$ *Q x* $\Longrightarrow$ *MultiSync S M P* $\sqsubseteq_{DT}$ *MultiSync S M Q*›
and *mono-MultiSync-FD*[*simp*, *elim*]:
‹$\forall x \in\# M$. *P x* $\sqsubseteq_{FD}$ *Q x* $\Longrightarrow$ *MultiSync S M P* $\sqsubseteq_{FD}$ *MultiSync S M Q*›
⟨*proof*⟩

find-theorems *name*: *mset name*: *ind*
lemmas *mono-MultiInter-DT*[*simp*, *elim*] = *mono-MultiSync-DT*[**where** *S* = ‹{}›]
and *mono-MultiInter-FD*[*simp*, *elim*] = *mono-MultiSync-FD*[**where** *S* = ‹{}›]
and *mono-MultiPar-DT*[*simp*, *elim*] = *mono-MultiSync-DT*[**where** *S* = ‹*UNIV*›]
and *mono-MultiPar-FD*[*simp*, *elim*] = *mono-MultiSync-FD*[**where** *S* = ‹*UNIV*›]

lemma *mono-MultiSeq-F*:
‹$\forall x \in$ *set L*. *P x* $\sqsubseteq_F$ *Q x* $\Longrightarrow$ *MultiSeq L P* $\sqsubseteq_F$ *MultiSeq L Q*›
⟨*proof*⟩

lemma *mono-MultiSeq-D*:
‹$\forall x \in$ *set L*. *P x* $\sqsubseteq_D$ *Q x* $\Longrightarrow$ *MultiSeq L P* $\sqsubseteq_D$ *MultiSeq L Q*›
⟨*proof*⟩

lemma *mono-MultiSeq-T*:
‹$\forall x \in set\ L.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow MultiSeq\ L\ P \sqsubseteq_T MultiSeq\ L\ Q$›
⟨*proof*⟩

lemma *mono-MultiSeq-DT*[*simp*, *elim*]:
‹$\forall x \in set\ L.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow MultiSeq\ L\ P \sqsubseteq_{DT} MultiSeq\ L\ Q$›
and *mono-MultiSeq-FD*[*simp*, *elim*]:
‹$\forall x \in set\ L.\ P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow MultiSeq\ L\ P \sqsubseteq_{FD} MultiSeq\ L\ Q$›
⟨*proof*⟩

lemma *mono-GlobalNdet*[*simp*] : ‹$GlobalNdet\ A\ P \sqsubseteq GlobalNdet\ A\ Q$›
if ‹$\forall x \in A.\ P\ x \sqsubseteq Q\ x$›
⟨*proof*⟩

lemma *mono-GlobalNdet-F*[*simp*, *elim*]:
‹$\forall x \in A.\ P\ x \sqsubseteq_F Q\ x \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_F GlobalNdet\ A\ Q$›
and *mono-GlobalNdet-D*[*simp*, *elim*]:
‹$\forall x \in A.\ P\ x \sqsubseteq_D Q\ x \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_D GlobalNdet\ A\ Q$›
and *mono-GlobalNdet-T*[*simp*, *elim*]:
‹$\forall x \in A.\ P\ x \sqsubseteq_T Q\ x \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_T GlobalNdet\ A\ Q$›
and *mono-GlobalNdet-DT*[*simp*, *elim*]:
‹$\forall x \in A.\ P\ x \sqsubseteq_{DT} Q\ x \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_{DT} GlobalNdet\ A\ Q$›
and *mono-GlobalNdet-FD*[*simp*, *elim*]:
‹$\forall x \in A.\ P\ x \sqsubseteq_{FD} Q\ x \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_{FD} GlobalNdet\ A\ Q$›
⟨*proof*⟩

lemma *GlobalNdet-refine-FD-subset*:
‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow GlobalNdet\ B\ P \sqsubseteq_{FD} GlobalNdet\ A\ P$›
⟨*proof*⟩

lemma *GlobalNdet-refine-F-subset*:
‹$A \neq \{\} \Longrightarrow A \subseteq B \Longrightarrow GlobalNdet\ B\ P \sqsubseteq_F GlobalNdet\ A\ P$›
⟨*proof*⟩

lemma *GlobalNdet-refine-FD*: ‹$a \in A \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_{FD} P\ a$›
⟨*proof*⟩

lemma *GlobalNdet-refine-F*: ‹$a \in A \Longrightarrow GlobalNdet\ A\ P \sqsubseteq_F P\ a$›
⟨*proof*⟩

lemma *mono-GlobalNdet-FD-const*:
‹$A \neq \{\} \Longrightarrow \forall x \in A.\ P \sqsubseteq_{FD} Q\ x \Longrightarrow P \sqsubseteq_{FD} GlobalNdet\ A\ Q$›
⟨*proof*⟩

lemma *mono-GlobalNdet-F-const*:

‹$A \neq \{\} \Longrightarrow \forall x \in A.\ P \sqsubseteq_F Q\ x \Longrightarrow P \sqsubseteq_F$ *GlobalNdet* $A\ Q$›
⟨*proof*⟩

9.2 Combination of Multi-Operators Laws

lemma *finite-Mprefix-is-MultiDet*:
‹*finite* $A \Longrightarrow (\Box\ p \in A \rightarrow P\ p) = \Box\ p \in A.\ (p \rightarrow P\ p)$›
⟨*proof*⟩

lemma *finite-Mndetprefix-is-MultiNdet*:
‹*finite* $A \Longrightarrow$ *Mndetprefix* $A\ P = \bigsqcap\ p \in A.\ (p \rightarrow P\ p)$›
⟨*proof*⟩

lemma ‹$Q \Box (\bigsqcap\ p \in \{\}.\ P\ p) = \bigsqcap\ p \in \{\}.\ (Q \Box P\ p) \Longrightarrow Q = STOP$›
⟨*proof*⟩

lemma *Det-MultiNdet-distrib*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow M \Box (\bigsqcap\ p \in A.\ P\ p) = \bigsqcap\ p \in A.\ M \Box P\ p$›
⟨*proof*⟩

lemma ‹$M \sqcap (\Box\ p \in \{\}.\ P\ p) = \Box\ p \in \{\}.\ (M \sqcap P\ p) \Longrightarrow M \sqcap STOP = STOP$›
⟨*proof*⟩

lemma *Ndet-MultiDet-distrib*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow M \sqcap (\Box\ p \in A.\ P\ p) = \Box\ p \in A.\ M \sqcap P\ p$›
⟨*proof*⟩

lemma *MultiNdet-Sync-left-distrib*:
‹$B \neq \{\} \Longrightarrow$ *finite* $B \Longrightarrow (\bigsqcap\ a \in B.\ P\ a)\ [\![S]\!]\ M = \bigsqcap\ a \in B.\ (P\ a\ [\![S]\!]\ M)$›
⟨*proof*⟩

lemma *MultiNdet-Sync-right-distrib*:
‹$B \neq \{\} \Longrightarrow$ *finite* $B \Longrightarrow M\ [\![S]\!]$ *MultiNdet* $B\ P = \bigsqcap\ a{\in}B.\ (M\ [\![S]\!]\ P\ a)$›
⟨*proof*⟩

lemma *Sync-MultiNdet-cartprod*:
‹$A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow B \neq \{\} \Longrightarrow$ *finite* $B \Longrightarrow$
$(\bigsqcap\ (s,\ t) \in A \times B.\ (x\ s\ [\![S]\!]\ y\ t)) = (\bigsqcap\ s \in A.\ x\ s)\ [\![S]\!]\ (\bigsqcap\ t \in B.\ y\ t)$›
⟨*proof*⟩

lemmas *Inter-MultiNdet-cartprod* = *Sync-MultiNdet-cartprod*[**where** S = ‹{}›]
and *Par-MultiNdet-cartprod* = *Sync-MultiNdet-cartprod*[**where** S = *UNIV*]

lemmas *MultiNdet-Inter-left-distrib* =
MultiNdet-Sync-left-distrib[**where** S = ‹{}›]
and *MultiNdet-Par-left-distrib* =
MultiNdet-Sync-left-distrib[**where** S = ‹*UNIV*›]

lemma *Seq-MultiNdet-distribR*:
‹$\mathit{finite}\ A \Longrightarrow (\bigsqcap\ p \in A.\ P\ p)\ \mathbf{;}\ S = (\bigsqcap\ p \in A.\ (P\ p\ \mathbf{;}\ S))$›
⟨*proof*⟩

lemma *Seq-MultiNdet-distribL*:
‹$A \neq \{\} \Longrightarrow \mathit{finite}\ A \Longrightarrow S\ \mathbf{;}\ (\bigsqcap\ p \in A.\ P\ p) = (\bigsqcap\ p \in A.\ (S\ \mathbf{;}\ P\ p))$›
⟨*proof*⟩

lemma *prefix-MultiNdet-distrib*:
‹$A \neq \{\} \Longrightarrow \mathit{finite}\ A \Longrightarrow (a \to (\bigsqcap\ p \in A.\ P\ p) = \bigsqcap\ p \in A.\ (a \to P\ p))$›
⟨*proof*⟩

lemma *Mndetprefix-MultiNdet-distrib*:
‹$(\sqcap\ q \in B \to (\bigsqcap\ p \in A.\ P\ p\ q)) = \bigsqcap\ p \in A.\ \sqcap\ q \in B \to P\ p\ q$›
if *finB*: ‹*finite* B› **and** *nonemptyA*: ‹$A \neq \{\}$› **and** *finA*: ‹*finite* A›
⟨*proof*⟩

lemma *MultiDet-Mprefix*:
‹$\mathit{finite}\ A \Longrightarrow (\Box a \in A.\ \Box x \in S\ a \to P\ a\ x) =$
$\Box x \in (\bigcup a \in A.\ S\ a) \to \bigsqcap\ a \in \{a \in A.\ x \in S\ a\}.\ P\ a\ x$›
⟨*proof*⟩

lemma *MultiDet-prefix-is-MultiNdet-prefix*:
‹$\mathit{finite}\ A \Longrightarrow (\Box\ p \in A.\ (a \to P\ p)) = \bigsqcap\ p \in A.\ (a \to P\ p)$›
⟨*proof*⟩

lemma *prefix-MultiNdet-is-MultiDet-prefix*:
‹$A \neq \{\} \Longrightarrow \mathit{finite}\ A \Longrightarrow (a \to (\bigsqcap\ p \in A.\ P\ p) = \Box\ p \in A.\ (a \to P\ p))$›
⟨*proof*⟩

lemma *Mprefix-MultiNdet-distrib′*:
‹*finite* $B \Longrightarrow A \neq \{\} \Longrightarrow$ *finite* $A \Longrightarrow$
$(\square\ q \in B \rightarrow \bigsqcap\ p \in A.\ P\ p\ q) = \square\ p \in A.\ \square\ q \in B \rightarrow P\ p\ q$›
⟨*proof*⟩

lemma *MultiSync-Hiding-pseudo-distrib*:
‹*finite* $B \Longrightarrow B \cap S = \{\} \Longrightarrow$
$([\![S]\!]\ p \in\#\ M.\ ((P\ p) \setminus B)) = ([\![S]\!]\ p \in\#\ M.\ P\ p) \setminus B$›
⟨*proof*⟩

lemma *MultiSync-prefix-pseudo-distrib*:
‹$M \neq \{\#\} \Longrightarrow a \in S \Longrightarrow$
$(([\![S]\!]\ p \in\#\ M.\ (a \rightarrow P\ p)) = (a \rightarrow ([\![S]\!]\ p \in\#\ M.\ P\ p)))$›
⟨*proof*⟩

lemmas *MultiInter-Hiding-pseudo-distrib* =
MultiSync-Hiding-pseudo-distrib[**where** S = ‹{}›, *simplified*]
and *MultiPar-prefix-pseudo-distrib* =
MultiSync-prefix-pseudo-distrib[**where** S = ‹*UNIV*›, *simplified*]

lemma *Hiding-MultiNdet-distrib*:
‹*finite* $A \Longrightarrow (\bigsqcap\ p \in A.\ P\ p) \setminus B = (\bigsqcap\ p \in A.\ (P\ p \setminus B))$›
⟨*proof*⟩

lemma *Mndetprefix-Hiding-is-MultiNdet-prefix-Hiding*:
‹*finite* $A \Longrightarrow \sqcap\ p \in A - B \rightarrow ((P\ p) \setminus B) = \bigsqcap\ p \in A - B.\ (p \rightarrow ((P\ p) \setminus B))$›
⟨*proof*⟩

lemma *Hiding-Mndetprefix-is-MultiNdet-Hiding*:
‹*finite* $A \Longrightarrow A \subseteq B \Longrightarrow (\sqcap\ a \in A \rightarrow P) \setminus B = \bigsqcap\ a \in A.\ (P \setminus B)$›
⟨*proof*⟩

lemma *MultiSync-Mprefix-pseudo-distrib*:
‹$([\![S]\!]\ B \in\#\ M.\ \square\ x \in B \rightarrow P\ B\ x) =$
$\square\ x \in (\bigcap B \in$ *set-mset* $M.\ B) \rightarrow ([\![S]\!]\ B \in\#\ M.\ P\ B\ x)$›
if *nonempty*: ‹$M \neq \{\#\}$› **and** *hyp*: ‹$\forall\ B \in\#\ M.\ B \subseteq S$›
⟨*proof*⟩

lemmas *MultiPar-Mprefix-pseudo-distrib* =

MultiSync-Mprefix-pseudo-distrib[**where** $S = \langle UNIV \rangle$, *simplified*]

A result on Mndetprefix and Sync.

lemma *Mndetprefix-Sync-distr*: $\langle A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow$
$(\sqcap\ a \in A \to P\ a)\ [\![S]\!]\ (\sqcap\ b \in B \to Q\ b) =$
$\sqcap\ a{\in}A.\ \sqcap\ b{\in}B.\ (\Box c \in \{a\} - S \to (P\ a\ [\![S]\!]\ (b \to Q\ b)))\ \Box$
$(\Box d \in \{b\} - S \to ((a \to P\ a)\ [\![S]\!]\ Q\ b))\ \Box$
$(\Box c{\in}\{a\} \cap \{b\} \cap S \to (P\ a\ [\![S]\!]\ Q\ b))\rangle$
⟨*proof*⟩

A result on *MultiSeq* with *non-terminating*.

lemma *non-terminating-MultiSeq*:
$\langle (SEQ\ a \in @\ L.\ P\ a) =$
$SEQ\ a \in @\ take\ (Suc\ (first\text{-}elem\ (\lambda a.\ non\text{-}terminating\ (P\ a))\ L))\ L.\ P\ a\rangle$
⟨*proof*⟩

9.3 Results on *Renaming*

lemma *Renaming-GlobalNdet*:
$\langle Renaming\ (\sqcap\ a \in A.\ P\ (f\ a))\ f = \sqcap\ b \in f\ `\ A.\ Renaming\ (P\ b)\ f\rangle$
⟨*proof*⟩

lemma *Renaming-GlobalNdet-inj-on*:
$\langle Renaming\ (\sqcap\ a \in A.\ P\ a)\ f =$
$\sqcap\ b \in f\ `\ A.\ Renaming\ (P\ (THE\ a.\ a \in A \wedge f\ a = b))\ f\rangle$
if *inj-on-f*: $\langle inj\text{-}on\ f\ A\rangle$
⟨*proof*⟩

corollary *Renaming-GlobalNdet-inj*:
$\langle Renaming\ (\sqcap\ a \in A.\ P\ a)\ f =$
$\sqcap\ b \in f\ `\ A.\ Renaming\ (P\ (THE\ a.\ f\ a = b))\ f\rangle$ **if** *inj-f*: $\langle inj\ f\rangle$
⟨*proof*⟩

lemma *Renaming-MultiNdet*: $\langle finite\ A \Longrightarrow Renaming\ (\prod\ a \in A.\ P\ (f\ a))\ f =$
$\prod\ b \in f\ `\ A.\ Renaming\ (P\ b)\ f\rangle$
⟨*proof*⟩

lemma *Renaming-MultiNdet-inj-on*:
$\langle finite\ A \Longrightarrow inj\text{-}on\ f\ A \Longrightarrow$
$Renaming\ (\prod\ a \in A.\ P\ a)\ f =$
$\prod\ b \in f\ `\ A.\ Renaming\ (P\ (THE\ a.\ a \in A \wedge f\ a = b))\ f\rangle$
⟨*proof*⟩

corollary *Renaming-MultiNdet-inj*:

‹*finite A* ⟹ *inj f* ⟹
Renaming (⨅ *a* ∈ *A*. *P a*) *f* = ⨅ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*›
⟨*proof*⟩

lemma *Renaming-MultiDet*:
‹*finite A* ⟹ *Renaming* (□ *a* ∈ *A*. *P* (*f a*)) *f* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P b*) *f*›
⟨*proof*⟩

lemma *Renaming-MultiDet-inj-on*:
‹*Renaming* (□ *a* ∈ *A*. *P a*) *f* =
□ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *a* ∈ *A* ∧ *f a* = *b*)) *f*›
if *finite-A*: ‹*finite A*› **and** *inj-on-f*: ‹*inj-on f A*›
⟨*proof*⟩

corollary *Renaming-MultiDet-inj*:
‹*Renaming* (□ *a* ∈ *A*. *P a*) *f* = □ *b* ∈ *f* ‘ *A*. *Renaming* (*P* (*THE a*. *f a* = *b*)) *f*›
if *finite-A*: ‹*finite A*› **and** *inj-f*: ‹*inj f*›
⟨*proof*⟩

lemma *Renaming-MultiSeq*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P* (*f l*)) *f* = *SEQ m* ∈@ *map f L*. *Renaming* (*P m*) *f*›
⟨*proof*⟩

lemma *Renaming-MultiSeq-inj-on*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P l*) *f* =
SEQ m ∈@ *map f L*. *Renaming* (*P* (*THE l*. *l* ∈ *set L* ∧ *f l* = *m*)) *f*›
if *inj-on-f*: ‹*inj-on f* (*set L*)›
⟨*proof*⟩

corollary *Renaming-MultiSeq-inj*:
‹*Renaming* (*SEQ l* ∈@ *L*. *P l*) *f* =
SEQ m ∈@ *map f L*. *Renaming* (*P* (*THE l*. *f l* = *m*)) *f*› **if** *inj-f*: ‹*inj f*›
⟨*proof*⟩

end

Chapter 10

Example: Dining Philosophers

theory *DiningPhilosophers*
  imports *CSPM*
begin

10.1 Classic Version

We formalize here the Dining Philosophers problem with a locale.

locale *DiningPhilosophers* =

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

begin

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

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

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

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

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

definition *FORK* :: ‹*nat* ⇒ *dining-event process*›
where ‹$FORK\ i \equiv \mu\ X.\ (picks\ i\ i \rightarrow (putsdown\ i\ i \rightarrow X))\ \square$
$(picks\ ((i+1)\ mod\ N)\ i \rightarrow (putsdown\ ((i+1)\ mod\ N)\ i \rightarrow X))$›

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

definition ‹$PHILS \equiv |||\ P \in\#\ add\text{-}mset\ LPHIL0\ (mset\ (map\ RPHIL\ [1..<\ N])).\ P$›
definition ‹$FORKS \equiv |||\ P \in\#\ mset\ (map\ FORK\ [0..<\ N]).\ P$›

corollary ‹$N = 3 \Longrightarrow PHILS = (LPHIL0\ |||\ RPHIL\ 1\ |||\ RPHIL\ 2)$›
— just a test
⟨*proof*⟩

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

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

end

10.2 Formalization with fixrec Package

The **fixrec** package of `HOLCF` provides a more readable syntax (essentially, it allows us to "get rid of μ" in equations like $\mu\ x.\ P\ x$).

First, we need to see *nat* as *cpo*.

instantiation *nat* :: *discrete-cpo*
begin

definition *below-nat-def*:
$(x::nat) \sqsubseteq y \longleftrightarrow x = y$

instance ⟨*proof*⟩

end

locale *DiningPhilosophers-fixrec* =

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

begin

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

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

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

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

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

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

corollary ‹*N* = *3* ⟹ *PHILS* = (*LPHIL0* ||| *RPHIL*·*1* ||| *RPHIL*·*2*)›
— just a test
⟨*proof*⟩

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

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

end

end

Chapter 11

Example: Plain Old Telephone System

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

Plain old telephone service (POTS), or plain ordinary telephone system,[1] is a retronym for voice-grade telephone service employing analog signal transmission over copper loops. POTS was the standard service offering from telephone companies from 1876 until 1988[2] in the United States when the Integrated Services Digital Network (ISDN) Basic Rate Interface (BRI) was introduced, followed by cellular telephone systems, and voice over IP (VoIP). POTS remains the basic form of residential and small business service connection to the telephone network in many parts of the world. The term reflects the technology that has been available since the introduction of the public telephone system in the late 19th century, in a form mostly unchanged despite the introduction of Touch-Tone dialing, electronic telephone exchanges and fiber-optic communication into the public switched telephone network (PSTN).

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

theory *POTS*
  imports *CSPM*
begin

We need to see *int* as a *cpo*.

instantiation *int* :: *discrete-cpo*
begin

definition *below-int-def*:
  $(x::int) \sqsubseteq y \longleftrightarrow x = y$

instance ⟨*proof*⟩

end

11.1 The Alphabet and Basic Types of POTS

Underlying terminology apparent in the acronyms:

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

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

type-synonym *Phones* = ‹*int*›

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

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

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

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

lemma *nonempty-phones*[*simp*]: ‹*phones* ≠ {}›
and *finite-phones*[*simp*]: ‹*finite phones*›
and *at-least-two-phones*[*simp*]: ‹*2* ≤ *card phones*›
and *not-singl-phone*[*simp*]: ‹*phones* − {*p*} ≠ {}›
⟨*proof*⟩

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

11.2 Auxilliaries to Substructure the Specification

abbreviation *Sliding* :: ‹′α *process* ⇒ ′α *process* ⇒ ′α *process*› (**infixl** ‹▷› *78*)
where ‹*P* ▷ *Q* ≡ (*P* □ *Q*) ⊓ *Q*›
— This operator is also called Timeout, more studied in future theories.

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

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

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

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

11.3 A Telephone

```
fixrec     T          :: ‹Phones → channels process›
       and Oside      :: ‹Phones → channels process›
       and Tside      :: ‹Phones → channels process›
       and NoReject   :: ‹Phones → channels process›
       and Reject     :: ‹Phones → channels process›
where
   T-rec        [simp del]: ‹T·ts       = (Tside·ts ; T·ts) ▷ (Oside·ts ; T·ts)›
 | Oside-rec    [simp del]: ‹Oside·ts   = StartReject!ts
                                        → tcO!(ts,Osetup)
                                        → (⨅ p ∈ phones. Oside1 ts p)›
 | Tside-rec    [simp del]: ‹Tside·ts   = ctT?(y,z,os)|((y,z)=(ts,Tsetup))
                                        → StartReject!ts
                                        → (    tcT!(ts,Talert,os)
                                                → tcT!(ts,Tconnect,os)
                                                →(Tside-connected ts os)
                                             ⊓ (tcT!(ts,Tconnect,os)
                                                → (Tside-connected ts os)))›
 | NoReject-rec [simp del]: ‹NoReject·ts = StartReject!ts → Reject·ts›
 | Reject-rec   [simp del]: ‹Reject·ts  = ctT?(y,z,os)|(y=ts ∧ z=Tsetup ∧ os∈phones
∧ os≠ts)
                                        →      (tcT!(ts,Tbusy,os) → Reject·ts)
                                            □  (EndReject!ts → NoReject·ts)›
```

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

11.4 A Connector with the Network

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

11.5 Combining NETWORK and TELEPHONES to a SYSTEM

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

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

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

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

11.6 Simple Model of a User

fixrec *User* :: ‹*Phones* → *channels process*›
and *UserSCon* :: ‹*Phones* → *channels process*›
where
User-rec[*simp del*] : ‹*User*·*u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon*·*u*)
□ (*connected*!*u*→*UserSCon*·*u*)
□ (*tone-busy*!*u*→*on-hook*!*u*→*User*·*u*)
)
)
)
□ (*connected*!*u* → *UserSCon*·*u*)
)
□ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*UserSCon*·*u*)›
| *UserSCon-rec*[*simp del*]: ‹*UserSCon*·*u* = (*tone-busy*!*u* → *on-hook*!*u* → *User*·*u*)
▷ (*on-hook*!*u* → *User*·*u*)›

fixrec *User-Ndet* :: ‹*Phones* → *channels process*›
and *UserSCon-Ndet* :: ‹*Phones* → *channels process*›
where
User-Ndet-rec[*simp del*] : ‹*User-Ndet*·*u* = (*off-hook*!*u* →
(*tone-dial*!*u* →
(⊓ *p* ∈ *phones*. *digits*!(*u*,*p*)→*tone-quiet*!*u*→
((*tone-ring*!*u*→*connected*!*u*→*UserSCon-Ndet*·*u*)
⊓ (*connected*!*u*→*UserSCon-Ndet*·*u*)
⊓ (*tone-busy*!*u*→*on-hook*!*u*→*User-Ndet*·*u*)
)
)

)
⊓ (*connected*!*u* → *UserSCon-Ndet*·*u*)
)
⊓ (*tone-ring*!*u*→*off-hook*!*u*→*connected*!*u* →*UserSCon-Ndet*·*u*)›
| *UserSCon-Ndet-rec*[*simp del*]: ‹*UserSCon-Ndet*·*u* = (*tone-busy*!*u* → *on-hook*!*u* → *User-Ndet*·*u*) ⊓ (*on-hook*!*u* → *User-Ndet*·*u*)›

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

11.7 Toplevel Proof-Goals

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

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

lemma ‹∀ *p* ∈ *phones*. *Tel'*·*p* $\sqsubseteq_F$ *RUN UNIV*› ⟨*proof*⟩

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

end

end

Chapter 12

Results on *events-of*

theory *EventsProperties*
  imports *CSPM*
begin

12.1 With Operators of HOL-CSP

lemma *events-of-def-tickFree*:
  ‹*events-of* $P = (\bigcup t{\in}\{t \in \mathcal{T}\ P.\ tickFree\ t\}.\ \{a.\ ev\ a \in set\ t\})$›
⟨*proof*⟩

lemma *events-BOT*: ‹*events-of* $\bot = UNIV$›
  and *events-SKIP*: ‹*events-of* $SKIP = \{\}$›
  and *events-STOP*: ‹*events-of* $STOP = \{\}$›
  ⟨*proof*⟩

lemma *anti-mono-events-T*: ‹$P \sqsubseteq_T Q \Longrightarrow$ *events-of* $Q \subseteq$ *events-of* P›
  ⟨*proof*⟩

lemma *anti-mono-events-F*: ‹$P \sqsubseteq_F Q \Longrightarrow$ *events-of* $Q \subseteq$ *events-of* P›
  ⟨*proof*⟩

lemma *anti-mono-events-FD*: ‹$P \sqsubseteq_{FD} Q \Longrightarrow$ *events-of* $Q \subseteq$ *events-of* P›
  ⟨*proof*⟩

lemmas *events-fix-prefix* =
    events-DF[*of* ‹$\{a\}$›, *simplified DF-def Mndetprefix-unit*] **for** *a*

lemma *events-Mndetprefix*:
  ‹*events-of* (*Mndetprefix A P*) $= A \cup (\bigcup a{\in}A.$ *events-of* $(P\ a))$›
  ⟨*proof*⟩

lemma *events-Mprefix*:
‹*events-of* (*Mprefix A P*) = *A* ∪ (⋃*a*∈*A*. *events-of* (*P a*))›
⟨*proof*⟩

lemma *events-prefix*: ‹*events-of* (*a* → *P*) = *insert a* (*events-of P*)›
⟨*proof*⟩

lemma *events-Ndet*: ‹*events-of* (*P* ⊓ *Q*) = *events-of P* ∪ *events-of Q*›
⟨*proof*⟩

lemma *events-Det*: ‹*events-of* (*P* □ *Q*) = *events-of P* ∪ *events-of Q*›
⟨*proof*⟩

lemma *events-Renaming*:
‹*events-of* (*Renaming P f*) = (*if* 𝒟 *P* = {} *then f* ‘ *events-of P else UNIV*)›
⟨*proof*⟩

lemma *events-Seq*:
‹*events-of* (*P* **;** *Q*) =
(*if non-terminating P then events-of P else events-of P* ∪ *events-of Q*)›
⟨*proof*⟩

lemma *events-Sync*: ‹*events-of* (*P* ⟦*S*⟧ *Q*) ⊆ *events-of P* ∪ *events-of Q*›
⟨*proof*⟩

lemma *events-Inter*:
‹*events-of* ((*P* :: ′α *process*) ||| *Q*) = *events-of P* ∪ *events-of Q*›
⟨*proof*⟩

lemma *empty-div-hide-events-Hiding*: ‹*events-of* (*P* \ *B*) ⊆ *events-of P* − *B*›
if ‹*div-hide P B* = {}›
⟨*proof*⟩

lemma *not-empty-div-hide-events-Hiding*:
‹*div-hide P B* ≠ {} ⟹ *events-of* (*P* \ *B*) = *UNIV*›

⟨*proof*⟩

events-of and *deadlock-free*

lemma *nonempty-events-if-deadlock-free*: ‹*deadlock-free* $P \Longrightarrow$ *events-of* $P \neq \{\}$›
⟨*proof*⟩

lemma *events-in-DF*: ‹*DF* $A \sqsubseteq_{FD} P \Longrightarrow$ *events-of* $P \subseteq A$›
⟨*proof*⟩

lemma *nonempty-events-if-deadlock-free*$_{SKIP}$:
‹*deadlock-free*$_{SKIP}$ $P \Longrightarrow [tick] \in \mathcal{T}\ P \vee$ *events-of* $P \neq \{\}$›
⟨*proof*⟩

lemma *events-in-DF*$_{SKIP}$: ‹DF_{SKIP} $A \sqsubseteq_{FD} P \Longrightarrow$ *events-of* $P \subseteq A$›
⟨*proof*⟩

lemma ‹$\neg$ *events-of* $P \subseteq A \Longrightarrow \neg$ *DF* $A \sqsubseteq_{FD} P$›
and ‹$\neg$ *events-of* $P \subseteq A \Longrightarrow \neg$ DF_{SKIP} $A \sqsubseteq_{FD} P$›
⟨*proof*⟩

12.2 With Architectural Operators of HOL-CSPM

lemma *events-MultiNdet*:
‹*finite* $A \Longrightarrow$ *events-of* (*MultiNdet* A P) = ($\bigcup a{\in}A.$ *events-of* (P a))›
⟨*proof*⟩

lemma *events-MultiDet*:
‹*finite* $A \Longrightarrow$ *events-of* (*MultiDet* A P) = ($\bigcup a{\in}A.$ *events-of* (P a))›
⟨*proof*⟩

lemma *events-MultiSeq*:
‹*events-of* (*SEQ* $a \in@ L.$ P a) =
($\bigcup a \in$ *set* (*take* (*Suc* (*first-elem* ($\lambda a.$ *non-terminating* (P a)) L)) L).
events-of (P a))›
⟨*proof*⟩

lemma *events-MultiSeq-subset*:
‹*events-of* (*SEQ* $a \in@ L.$ P a) $\subseteq$ ($\bigcup a \in$ *set* $L.$ *events-of* (P a))›
⟨*proof*⟩

lemma *events-MultiSync*:
‹*events-of* ($[\![S]\!]$ $a \in\# M.$ P a) $\subseteq$ ($\bigcup a \in$ *set-mset* $M.$ *events-of* (P a))›
⟨*proof*⟩

lemma *events-MultiInter*:
‹*events-of* ($|||$ $a \in\#$ M. P a) = ($\bigcup a \in$ *set-mset* M. *events-of* (P a))›
⟨*proof*⟩

end

Chapter 13

Deadlock Results

theory *DeadlockResults*
  imports *CSPM*
begin

When working with the interleaving $P|||Q$, we intuitively expect it to be *deadlock-free* when both P and Q are.

This chapter contains several results about deadlock notion, and concludes with a proof of the theorem we just mentioned.

13.1 Unfolding Lemmas for the Projections of DF and DF_{SKIP}

DF and DF_{SKIP} naturally appear when we work around *deadlock-free* and *deadlock-free*$_{SKIP}$ notions (because

deadlock-free $P \equiv DF\ UNIV \sqsubseteq_{FD} P$

deadlock-free$_{SKIP}$ $P \equiv DF_{SKIP}\ UNIV \sqsubseteq_{F} P$).

It is therefore convenient to have the following rules for unfolding the projections.

lemma *F-DF*:
  ‹$\mathcal{F}$ $(DF\ A)$ =
   $(if\ A = \{\}\ then\ \{(s, X).\ s = []\}$
    $else\ (\bigcup x{\in}A.\ \{[]\} \times \{ref.\ ev\ x \notin ref\} \cup$
      $\{(tr, ref).\ tr \neq [] \wedge hd\ tr = ev\ x \wedge (tl\ tr, ref) \in \mathcal{F}\ (DF\ A)\}))$›
  and *F-DF*$_{SKIP}$:
  ‹$\mathcal{F}$ $(DF_{SKIP}\ A)$ =
   $(if\ A = \{\}\ then\ \{(s, X).\ s = [] \vee s = [tick]\}$
    $else\ \{(s, X)|\ s\ X.\ s = [] \wedge tick \notin X \vee s = [tick]\} \cup$
     $(\bigcup x{\in}A.\ \{[]\} \times \{ref.\ ev\ x \notin ref\} \cup$
      $\{(tr, ref).\ tr \neq [] \wedge hd\ tr = ev\ x \wedge (tl\ tr, ref) \in \mathcal{F}\ (DF_{SKIP}\ A)\}))$›
  ⟨*proof*⟩

corollary *Cons-F-DF*:
‹$(x \# t, X) \in \mathcal{F}$ $(DF\ A) \Longrightarrow (t, X) \in \mathcal{F}$ $(DF\ A)$›
and *Cons-F-DF_{SKIP}*:
‹$x \neq tick \Longrightarrow (x \# t, X) \in \mathcal{F}$ $(DF_{SKIP}\ A) \Longrightarrow (t, X) \in \mathcal{F}$ $(DF_{SKIP}\ A)$›
⟨*proof*⟩

lemma *D-DF*: ‹$\mathcal{D}$ $(DF\ A) = (\textit{if}\ A = \{\}\ \textit{then}\ \{\}$
$\textit{else}\ \{s.\ s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{D}\ (DF\ A))\})$›
and *D-DF_{SKIP}*: ‹$\mathcal{D}$ $(DF_{SKIP}\ A) = (\textit{if}\ A = \{\}\ \textit{then}\ \{\}$
$\textit{else}\ \{s.\ s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{D}\ (DF_{SKIP}\ A))\})$›
⟨*proof*⟩

lemma *T-DF*:
‹$\mathcal{T}$ $(DF\ A) =$
$(\textit{if}\ A = \{\}\ \textit{then}\ \{[]\}$
$\textit{else}\ \{s.\ s = [] \vee s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{T}\ (DF\ A))\})$›
and *T-DF_{SKIP}*:
‹$\mathcal{T}$ $(DF_{SKIP}\ A) =$
$(\textit{if}\ A = \{\}\ \textit{then}\ \{[], [tick]\}$
$\textit{else}\ \{s.\ s = [] \vee s = [tick] \vee$
$s \neq [] \wedge (\exists x \in A.\ hd\ s = ev\ x \wedge tl\ s \in \mathcal{T}\ (DF_{SKIP}\ A))\})$›
⟨*proof*⟩

13.2 Characterizations for *deadlock-free*, *deadlock-free$_{SKIP}$*

We want more results like ⟦*deadlock-free P*; *deadlock-free Q*⟧ $\Longrightarrow$ *deadlock-free* $(P \sqcap Q)$, and we want to add the reciprocal when possible.

The first thing we notice is that we only have to care about the failures

lemma ‹*deadlock-free$_{SKIP}$* $P \equiv DF_{SKIP}\ UNIV \sqsubseteq_F P$›
⟨*proof*⟩

lemma *deadlock-free-F*: ‹*deadlock-free* $P \longleftrightarrow DF\ UNIV \sqsubseteq_F P$›
⟨*proof*⟩

lemma *deadlock-free-Mprefix-iff*: ‹*deadlock-free* $(\square\ a \in A \rightarrow P\ a) \longleftrightarrow$
$A \neq \{\} \wedge (\forall a \in A.$ *deadlock-free* $(P\ a))$›
and *deadlock-free$_{SKIP}$-Mprefix-iff*: ‹*deadlock-free$_{SKIP}$* (*Mprefix* A P) $\longleftrightarrow$
$A \neq \{\} \wedge (\forall a \in A.$ *deadlock-free$_{SKIP}$* $(P\ a))$›
⟨*proof*⟩

lemmas *deadlock-free-prefix-iff* =

deadlock-free-Mprefix-iff [*of* ‹{*a*}› ‹λ*a*. *P*›, *folded write0-def*, *simplified*]
and $deadlock\text{-}free_{SKIP}$*-prefix-iff* =
$deadlock\text{-}free_{SKIP}$*-Mprefix-iff*[*of* ‹{*a*}› ‹λ*a*. *P*›, *folded write0-def*, *simplified*]
for *a P*

lemma *deadlock-free-Mndetprefix-iff*: ‹*deadlock-free* (⊓ *a* ∈ *A* → *P a*) ⟷
A ≠ {} ∧ (∀ *a*∈*A*. *deadlock-free* (*P a*))›
and $deadlock\text{-}free_{SKIP}$*-Mndetprefix-iff*: ‹$deadlock\text{-}free_{SKIP}$ (⊓ *a* ∈ *A* → *P a*) ⟷
A ≠ {} ∧ (∀ *a*∈*A*. $deadlock\text{-}free_{SKIP}$ (*P a*))›
⟨*proof*⟩

lemma *deadlock-free-Ndet-iff*: ‹*deadlock-free* (*P* ⊓ *Q*) ⟷
deadlock-free P ∧ *deadlock-free Q*›
and $deadlock\text{-}free_{SKIP}$*-Ndet-iff*: ‹$deadlock\text{-}free_{SKIP}$ (*P* ⊓ *Q*) ⟷
$deadlock\text{-}free_{SKIP}$ *P* ∧ $deadlock\text{-}free_{SKIP}$ *Q*›
⟨*proof*⟩

lemma *deadlock-free-GlobalNdet-iff*: ‹*deadlock-free* (⊓ *a* ∈ *A*. *P a*) ⟷
A ≠ {} ∧ (∀ *a* ∈ *A*. *deadlock-free* (*P a*))›
and $deadlock\text{-}free_{SKIP}$*-GlobalNdet-iff*: ‹$deadlock\text{-}free_{SKIP}$ (⊓ *a* ∈ *A*. *P a*) ⟷
A ≠ {} ∧ (∀ *a* ∈ *A*. $deadlock\text{-}free_{SKIP}$ (*P a*))›
⟨*proof*⟩

lemma *deadlock-free-MultiNdet-iff*: ‹*deadlock-free* (⨅ *a* ∈ *A*. *P a*) ⟷
A ≠ {} ∧ (∀ *a* ∈ *A*. *deadlock-free* (*P a*))›
and $deadlock\text{-}free_{SKIP}$*-MultiNdet-iff*: ‹$deadlock\text{-}free_{SKIP}$ (⨅ *a* ∈ *A*. *P a*) ⟷
A ≠ {} ∧ (∀ *a* ∈ *A*. $deadlock\text{-}free_{SKIP}$ (*P a*))›
if *fin*: ‹*finite A*›
⟨*proof*⟩

lemma *deadlock-free-MultiDet*:
‹⟦*A* ≠ {}; *finite A*; ∀ *a*∈*A*. *deadlock-free* (*P a*)⟧ ⟹ *deadlock-free* (□*a* ∈ *A*. *P a*)›
and $deadlock\text{-}free_{SKIP}$*-MultiDet*:
‹⟦*A* ≠ {}; *finite A*; ∀ *a*∈*A*. $deadlock\text{-}free_{SKIP}$ (*P a*)⟧ ⟹ $deadlock\text{-}free_{SKIP}$ (□*a* ∈ *A*. *P a*)›
⟨*proof*⟩

lemma *deadlock-free-Det*:
‹*deadlock-free* $P \Longrightarrow$ *deadlock-free* $Q \Longrightarrow$ *deadlock-free* $(P \square Q)$›
and *deadlock-free*$_{SKIP}$*-Det*:
‹*deadlock-free*$_{SKIP}$ $P \Longrightarrow$ *deadlock-free*$_{SKIP}$ $Q \Longrightarrow$ *deadlock-free*$_{SKIP}$ $(P \square Q)$›
⟨*proof*⟩

For $P \sqcap Q$, we can not expect more:

lemma
‹$\exists P\ Q.$ *deadlock-free* $P \wedge \neg$ *deadlock-free* $Q \wedge$
deadlock-free $(P \sqcap Q)$›
‹$\exists P\ Q.$ *deadlock-free*$_{SKIP}$ $P \wedge \neg$ *deadlock-free*$_{SKIP}$ $Q \wedge$
deadlock-free$_{SKIP}$ $(P \sqcap Q)$›
⟨*proof*⟩

lemma *FD-Mndetprefix-iff*:
‹$A \neq \{\} \Longrightarrow P \sqsubseteq_{FD} \sqcap a \in A \rightarrow Q \longleftrightarrow (\forall a \in A.\ P \sqsubseteq_{FD} (a \rightarrow Q))$›
⟨*proof*⟩

lemma *Mndetprefix-FD*: ‹$(\exists a \in A.\ (a \rightarrow Q) \sqsubseteq_{FD} P) \Longrightarrow \sqcap a \in A \rightarrow Q \sqsubseteq_{FD} P$›
⟨*proof*⟩

Mprefix, *Sync* and *deadlock-free*

lemma *Mprefix-Sync-deadlock-free*:
assumes *not-all-empty*: ‹$A \neq \{\} \vee B \neq \{\} \vee A' \cap B' \neq \{\}$›
and ‹$A \cap S = \{\}$› **and** ‹$A' \subseteq S$› **and** ‹$B \cap S = \{\}$› **and** ‹$B' \subseteq S$›
and ‹$\forall x \in A.$ *deadlock-free* $(P\ x\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q)$›
and ‹$\forall y \in B.$ *deadlock-free* $($*Mprefix* $(A \cup A')\ P\ [\![S]\!]\ Q\ y)$›
and ‹$\forall x \in A' \cap B'.$ *deadlock-free* $((P\ x\ [\![S]\!]\ Q\ x))$›
shows ‹*deadlock-free* $($*Mprefix* $(A \cup A')\ P\ [\![S]\!]$ *Mprefix* $(B \cup B')\ Q)$›
⟨*proof*⟩

lemmas *Mprefix-Sync-subset-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A = ‹$\{\}$› **and** B = ‹$\{\}$›, *simplified*]
and *Mprefix-Sync-indep-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A' = ‹$\{\}$› **and** B' = ‹$\{\}$›, *simplified*]
and *Mprefix-Sync-right-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A = ‹$\{\}$› **and** B' = ‹$\{\}$›, *simplified*]
and *Mprefix-Sync-left-deadlock-free* = *Mprefix-Sync-deadlock-free*
[**where** A' = ‹$\{\}$› **and** B = ‹$\{\}$›, *simplified*]

13.3 Results on *Renaming*

The *Renaming* operator is new (release of 2023), so here are its properties on reference processes from *HOL−CSP.Assertions*, and deadlock notion.

13.3.1 Behaviour with References Processes

For *DF*

lemma *DF-FD-Renaming-DF*: ‹*DF* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*DF A*) *f*›
⟨*proof*⟩

lemma *Renaming-DF-FD-DF*: ‹*Renaming* (*DF A*) *f* $\sqsubseteq_{FD}$ *DF* (*f* ‘ *A*)›
 if *finitary*: ‹*finitary f*›
⟨*proof*⟩

For DF_{SKIP}

lemma DF_{SKIP}-*FD-Renaming*-DF_{SKIP}:
 ‹DF_{SKIP} (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (DF_{SKIP} *A*) *f*›
⟨*proof*⟩

lemma *Renaming*-DF_{SKIP}-*FD*-DF_{SKIP}:
 ‹*Renaming* (DF_{SKIP} *A*) *f* $\sqsubseteq_{FD}$ DF_{SKIP} (*f* ‘ *A*)›
 if *finitary*: ‹*finitary f*›
⟨*proof*⟩

For *RUN*

lemma *RUN-FD-Renaming-RUN*: ‹*RUN* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*RUN A*) *f*›
⟨*proof*⟩

lemma *Renaming-RUN-FD-RUN*: ‹*Renaming* (*RUN A*) *f* $\sqsubseteq_{FD}$ *RUN* (*f* ‘ *A*)›
 if *finitary*: ‹*finitary f*›
⟨*proof*⟩

For *CHAOS*

lemma *CHAOS-FD-Renaming-CHAOS*:
 ‹*CHAOS* (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* (*CHAOS A*) *f*›
⟨*proof*⟩

lemma *Renaming-CHAOS-FD-CHAOS*:
 ‹*Renaming* (*CHAOS A*) *f* $\sqsubseteq_{FD}$ *CHAOS* (*f* ‘ *A*)›
 if *finitary*: ‹*finitary f*›
⟨*proof*⟩

For $CHAOS_{SKIP}$

lemma $CHAOS_{SKIP}$-*FD-Renaming*-$CHAOS_{SKIP}$:
 ‹$CHAOS_{SKIP}$ (*f* ‘ *A*) $\sqsubseteq_{FD}$ *Renaming* ($CHAOS_{SKIP}$ *A*) *f*›
⟨*proof*⟩

lemma *Renaming-CHAOS*$_{SKIP}$*-FD-CHAOS*$_{SKIP}$:
‹*Renaming* (*CHAOS*$_{SKIP}$ *A*) *f* $\sqsubseteq_{FD}$ *CHAOS*$_{SKIP}$ (*f* ‘ *A*)›
if *finitary*: ‹*finitary f*›
⟨*proof*⟩

13.3.2 Corollaries on *deadlock-free* and *deadlock-free*$_{SKIP}$

lemmas *Renaming-DF* =
FD-antisym[*OF Renaming-DF-FD-DF DF-FD-Renaming-DF*]
and *Renaming-DF*$_{SKIP}$ =
FD-antisym[*OF Renaming-DF*$_{SKIP}$*-FD-DF*$_{SKIP}$ *DF*$_{SKIP}$*-FD-Renaming-DF*$_{SKIP}$]
and *Renaming-RUN* =
FD-antisym[*OF Renaming-RUN-FD-RUN RUN-FD-Renaming-RUN*]
and *Renaming-CHAOS* =
FD-antisym[*OF Renaming-CHAOS-FD-CHAOS CHAOS-FD-Renaming-CHAOS*]
and *Renaming-CHAOS*$_{SKIP}$ =
FD-antisym[*OF Renaming-CHAOS*$_{SKIP}$*-FD-CHAOS*$_{SKIP}$
CHAOS$_{SKIP}$*-FD-Renaming-CHAOS*$_{SKIP}$]

lemma *deadlock-free-imp-deadlock-free-Renaming*: ‹*deadlock-free* (*Renaming P f*)›
if ‹*deadlock-free P*›
⟨*proof*⟩

lemma *deadlock-free-Renaming-imp-deadlock-free*: ‹*deadlock-free P*›
if ‹*inj f*› **and** ‹*deadlock-free* (*Renaming P f*)›
⟨*proof*⟩

corollary *deadlock-free-Renaming-iff*:
‹*inj f* $\Longrightarrow$ *deadlock-free* (*Renaming P f*) $\longleftrightarrow$ *deadlock-free P*›
⟨*proof*⟩

lemma *deadlock-free*$_{SKIP}$*-imp-deadlock-free*$_{SKIP}$*-Renaming*:
‹*deadlock-free*$_{SKIP}$ (*Renaming P f*)› **if** ‹*deadlock-free*$_{SKIP}$ *P*›
⟨*proof*⟩

lemma *deadlock-free*$_{SKIP}$*-Renaming-imp-deadlock-free*$_{SKIP}$:
‹*deadlock-free*$_{SKIP}$ *P*› **if** ‹*inj f*› **and** ‹*deadlock-free*$_{SKIP}$ (*Renaming P f*)›
⟨*proof*⟩

corollary *deadlock-free*$_{SKIP}$*-Renaming-iff*:
‹*inj f* $\Longrightarrow$ *deadlock-free*$_{SKIP}$ (*Renaming P f*) $\longleftrightarrow$ *deadlock-free*$_{SKIP}$ *P*›
⟨*proof*⟩

13.4 Big Results

13.4.1 Interesting Equivalence

lemma *deadlock-free-of-Sync-iff-DF-FD-DF-Sync-DF*:
‹($\forall$ *P Q*. *deadlock-free* (*P*::′α *process*) $\longrightarrow$ *deadlock-free Q* $\longrightarrow$
deadlock-free (*P* ⟦*S*⟧ *Q*))
$\longleftrightarrow$ (*DF* (*UNIV*::′α *set*) $\sqsubseteq_{FD}$ (*DF UNIV* ⟦*S*⟧ *DF UNIV*))› (**is** ‹*?lhs* $\longleftrightarrow$ *?rhs*›)
⟨*proof*⟩

From this general equivalence on *Sync*, we immediately obtain the equivalence on $A|||B$: ($\forall$ *P Q*. *deadlock-free P* $\longrightarrow$ *deadlock-free Q* $\longrightarrow$ *deadlock-free* $(P|||Q)$) = *DF UNIV* $\sqsubseteq_{FD}$ *DF UNIV*$|||$*DF UNIV*.

13.4.2 *STOP* and *SKIP* Synchronized with *DF A*

lemma *DF-FD-DF-Sync-STOP-or-SKIP-iff*:
‹(*DF A* $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *P*) $\longleftrightarrow$ $A \cap S = \{\}$›
if *P-disj*: ‹$P = STOP \lor P = SKIP$›
⟨*proof*⟩

lemma *DF-Sync-STOP-or-SKIP-FD-DF*: ‹*DF A* ⟦*S*⟧ *P* $\sqsubseteq_{FD}$ *DF A*›
if *P-disj*: ‹$P = STOP \lor P = SKIP$› **and** *empty-inter*: ‹$A \cap S = \{\}$›
⟨*proof*⟩

lemmas *DF-FD-DF-Sync-STOP-iff* =
DF-FD-DF-Sync-STOP-or-SKIP-iff[*of STOP*, *simplified*]
and *DF-FD-DF-Sync-SKIP-iff* =
DF-FD-DF-Sync-STOP-or-SKIP-iff[*of SKIP*, *simplified*]
and *DF-Sync-STOP-FD-DF* =
DF-Sync-STOP-or-SKIP-FD-DF[*of STOP*, *simplified*]
and *DF-Sync-SKIP-FD-DF* =
DF-Sync-STOP-or-SKIP-FD-DF[*of SKIP*, *simplified*]

13.4.3 Finally, *deadlock-free* $(P|||Q)$

theorem *DF-F-DF-Sync-DF*: ‹*DF* (*A* $\cup$ *B*::′α *set*) $\sqsubseteq_F$ *DF A* ⟦*S*⟧ *DF B*›
if *nonempty*: ‹$A \neq \{\} \land B \neq \{\}$›
and *intersect-hyp*: ‹$B \cap S = \{\} \lor (\exists y.\ B \cap S = \{y\} \land A \cap S \subseteq \{y\})$›
⟨*proof*⟩

lemma *DF-FD-DF-Sync-DF*:
‹$A \neq \{\} \land B \neq \{\} \Longrightarrow B \cap S = \{\} \lor (\exists y.\ B \cap S = \{y\} \land A \cap S \subseteq \{y\}) \Longrightarrow$
DF (*A* $\cup$ *B*) $\sqsubseteq_{FD}$ *DF A* ⟦*S*⟧ *DF B*›
⟨*proof*⟩

theorem *DF-FD-DF-Sync-DF-iff*:
‹$DF\ (A \cup B) \sqsubseteq_{FD} DF\ A\ [\![S]\!]\ DF\ B \longleftrightarrow$
$(\quad$ *if* $A = \{\}$ *then* $B \cap S = \{\}$
else if $B = \{\}$ *then* $A \cap S = \{\}$
else $A \cap S = \{\} \vee (\exists a.\ A \cap S = \{a\} \wedge B \cap S \subseteq \{a\}) \vee$
$B \cap S = \{\} \vee (\exists b.\ B \cap S = \{b\} \wedge A \cap S \subseteq \{b\}))$›
(**is** ‹*?FD-ref* $\longleftrightarrow$ (*if* $A = \{\}$ *then* $B \cap S = \{\}$
else if $B = \{\}$ *then* $A \cap S = \{\}$
else ?cases)›)
⟨*proof*⟩

lemma
‹$(\forall a \in A.\ X\ a \cap S = \{\} \vee (\forall b \in A.\ \exists y.\ X\ a \cap S = \{y\} \wedge X\ b \cap S \subseteq \{y\}))$
$\longleftrightarrow (\forall a \in A.\ \forall b \in A.\ \exists y.\ (X\ a \cup X\ b) \cap S \subseteq \{y\})$›
— this is the reason we write ugly_hyp this way
⟨*proof*⟩

lemma *DF-FD-DF-MultiSync-DF*:
‹$DF\ (\bigcup\ x \in (insert\ a\ A).\ X\ x) \sqsubseteq_{FD} [\![S]\!]\ x \in\#\ mset\text{-}set\ (insert\ a\ A).\ DF\ (X\ x)$›
if *fin*: ‹*finite A*› **and** *nonempty*: ‹$X\ a \neq \{\}$› ‹$\forall b \in A.\ X\ b \neq \{\}$›
and *ugly-hyp*: ‹$\forall b \in A.\ X\ b \cap S = \{\} \vee (\exists y.\ X\ b \cap S = \{y\} \wedge X\ a \cap S \subseteq \{y\})$›
‹$\forall b \in A.\ \forall c \in A.\ \exists y.\ (X\ b \cup X\ c) \cap S \subseteq \{y\}$›
⟨*proof*⟩

lemma *DF-FD-DF-MultiSync-DF'*:
‹$[\![finite\ A;\ \forall a \in A.\ X\ a \neq \{\};\ \forall a \in A.\ \forall b \in A.\ \exists y.\ (X\ a \cup X\ b) \cap S \subseteq \{y\}]\!]$
$\Longrightarrow DF\ (\bigcup\ a \in A.\ X\ a) \sqsubseteq_{FD} [\![S]\!]\ a \in\#\ mset\text{-}set\ A.\ DF\ (X\ a)$›
⟨*proof*⟩

lemmas *DF-FD-DF-MultiInter-DF* =
DF-FD-DF-MultiSync-DF'[**where** S = ‹{}›, *simplified*]
and *DF-FD-DF-MultiPar-DF* =
DF-FD-DF-MultiSync-DF [**where** $S = UNIV$, *simplified*]
and *DF-FD-DF-MultiPar-DF'* =
DF-FD-DF-MultiSync-DF'[**where** $S = UNIV$, *simplified*]

lemma ‹$DF\ \{a\} = DF\ \{a\}\ [\![S]\!]\ STOP \longleftrightarrow a \notin S$›

⟨*proof*⟩

lemma ‹$DF\ \{a\}\ [\![S]\!]\ STOP = STOP \longleftrightarrow a \in S$›
⟨*proof*⟩

corollary *DF-FD-DF-Inter-DF*: ‹$DF\ (A{::}'\alpha\ set) \sqsubseteq_{FD} DF\ A\ |||\ DF\ A$›
⟨*proof*⟩

corollary *DF-UNIV-FD-DF-UNIV-Inter-DF-UNIV*:
‹$DF\ UNIV \sqsubseteq_{FD} DF\ UNIV\ |||\ DF\ UNIV$›
⟨*proof*⟩

corollary *Inter-deadlock-free*:
‹*deadlock-free* $P \Longrightarrow$ *deadlock-free* $Q \Longrightarrow$ *deadlock-free* $(P\ |||\ Q)$›
⟨*proof*⟩

theorem *MultiInter-deadlock-free*:
‹$M \neq \{\#\} \Longrightarrow \forall\, p \in\!\#\ M.$ *deadlock-free* $(P\ p) \Longrightarrow$
deadlock-free $(|||\ p \in\!\#\ M.\ P\ p)$›
⟨*proof*⟩

end

Chapter 14

Conclusion

In this session, we defined five architectural operators: *MultiDet*, *MultiNdet* and *GlobalNdet*, *MultiSync*, and *MultiSeq* as respective generalizations of $P \square Q$, $P \sqcap Q$, $P [\![S]\!] Q$, and $P \mathbin{;} Q$.

We did this in a fully-abstract way, that is:

- $(\square)$ is commutative, idempotent and admits *STOP* as a neutral element so we defined *MultiDet* on a *finite* $'\alpha$ *set* A by making it equal to *STOP* when $A = \emptyset$.
- $(\sqcap)$ is also commutative and idempotent so we defined *MultiNdet* on a *finite* $'\alpha$ *set* A by making it equal to *STOP* when $A = \emptyset$. Beware of the fact that *STOP* is not the neutral element for $(\sqcap)$ (this operator does not admit a neutral element) so we **do not have** the equality

  $\bigsqcap\ p{\in}\{a\}.\ \ P\ p = P\ a \sqcap (\bigsqcap\ p{\in}\emptyset.\ \ P\ p)$

  while this holds for $(\square)$ and *MultiDet*).

  As its failures and divergences can easily be generalized to the infinite case, we also defined *GlobalNdet* verifying

  $\mathit{finite}\ A \Longrightarrow \sqcap\ p{\in}A.\ \ P\ p = \bigsqcap\ p{\in}A.\ \ P\ p$

- *Sync* is commutative but is not idempotent so we defined *MultiSync* on a $'\alpha$ *multiset* M to keep the multiplicity of the processes. We made it equal to *STOP* when $M = \{\#\}$ but like $(\sqcap)$, *Sync* does not admit a neutral element so beware of the fact that in general

  $[\![S]\!]\ p{\in}\#\{\#a\#\}.\ \ P\ p \neq P\ a\ [\![S]\!]\ ([\![S]\!]\ p{\in}\#\{\#\}.\ \ P\ p)$

- $(\mathbin{;})$ is neither commutative nor idempotent, so we defined *MultiSeq* on a $'\alpha$ *list* L to keep the multiplicity and the order of the processes. Since *SKIP* is the neutral element for $(\mathbin{;})$, we have

$$SEQ\ p{\in}@[a].\ P\ p = (SEQ\ p{\in}@[].\ P\ p)\ \mathbf{;}\ P\ a$$

$$SEQ\ p{\in}@[a].\ P\ p = P\ a\ \mathbf{;}\ (SEQ\ p{\in}@[].\ P\ p)$$

On our architectural operators we proved continuity (under weakest liberal assumptions), wrote refinements rules and obtained results about the behaviour with other operators inherited from the binary rules.

We presented two examples: Dining Philosophers, and POTS.
In both, we underlined the usefulness of the architectural operators for modeling complex systems.

Finally we provided powerful results on *events-of* and *deadlock-free* among which the most important is undoubtedly :

$$[\![M \neq \{\#\};\ \forall\, p{\in}\#M.\ \textit{deadlock-free}\ (P\ p)]\!] \Longrightarrow \textit{deadlock-free}\ (|||\ p{\in}\#M.\ \ P\ p)$$

This theorem allows, for example, to establish:

$$0 < n \Longrightarrow \textit{deadlock-free}\ (|||\ i{\in}\#\textit{mset}\ [0..{<}n].\ \ P\ i)$$

under the assumption that a family of processes parameterized by $i :: nat$ verifies $\forall\, i{<}n.\ \textit{deadlock-free}\ (P\ i)$.

Bibliography

[1] 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.

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

[3] S. Taha, L. Ye, and B. Wolff. Hol-csp version 2.0. *Archive of Formal Proofs*, April 2019. https://isa-afp.org/entries/HOL-CSP.html, Formal proof development.