

# The HOL-CSP Refinement Toolkit

Safouan Taha      Burkhart Wolff      Lina Ye

May 14, 2024



# Abstract

Recently, a modern version of Roscoes and Brookes [3] Failure-Divergence Semantics for CSP has been formalized in Isabelle [10].

We use this formal development called HOL-CSP2.0 to analyse a family of refinement notions, comprising classic and new ones. This analysis enables to derive a number of properties that allow to deepen the understanding of these notions, in particular with respect to specification decomposition principles for the case of infinite sets of events. The established relations between the refinement relations help to clarify some obscure points in the CSP literature, but also provide a weapon for shorter refinement proofs. Furthermore, we provide a framework for state-normalisation allowing to formally reason on parameterised process architectures.

As a result, we have a modern environment for formal proofs of concurrent systems that allow for the combination of general infinite processes with locally finite ones in a logically safe way. We demonstrate these verification-techniques for classical, generalised examples: The CopyBuffer for arbitrary data and the Dijkstra's Dining Philosopher Problem of arbitrary size.

If you consider to cite this work, please refer to [11].



# Contents

<b>1</b>	<b>Context</b>	<b>7</b>
1.1	Introduction . . . . .	7
1.2	The Global Architecture of CSP_RefTk . . . . .	9
<b>2</b>	<b>Normalisation of Deterministic CSP Processes</b>	<b>11</b>
2.1	Deterministic normal-forms with explicit state . . . . .	11
2.2	Interleaving product lemma . . . . .	11
2.3	Synchronous product lemma . . . . .	13
2.4	Consequences . . . . .	14
<b>3</b>	<b>Examples</b>	<b>17</b>
3.1	CopyBuffer Refinement over an infinite alphabet . . . . .	17
3.1.1	The Copy-Buffer vs. reference processes . . . . .	17
3.1.2	... and abstract consequences . . . . .	17
3.2	Generalized Dining Philosophers . . . . .	18
3.2.1	Preliminary lemmas for proof automation . . . . .	18
3.2.2	The dining processes definition . . . . .	18
3.2.3	Translation into normal form . . . . .	20
3.2.4	The normal form for the global philosopher network . . . . .	29
3.2.5	The complete process system under normal form . . . . .	32
3.2.6	And finally: Philosophers may dine ! Always ! . . . . .	33
<b>4</b>	<b>Conclusion</b>	<b>39</b>



# Chapter 1

## Context

### 1.1 Introduction

Communicating Sequential Processes CSP is a language to specify and verify patterns of interaction of concurrent systems. Together with CCS and LOTOS, it belongs to the family of *process algebras*. CSP's rich theory comprises denotational, operational and algebraic semantic facets and has influenced programming languages such as Limbo, Crystal, Clojure and most notably Golang [5]. CSP has been applied in industry as a tool for specifying and verifying the concurrent aspects of hardware systems, such as the T9000 transputer [1].

The theory of CSP, in particular the denotational Failure/Divergence Denotational Semantics, has been initially proposed in the book by Tony Hoare [6], but evolved substantially since [2, 3, 8].

Verification of CSP properties has been centered around the notion of *process refinement orderings*, most notably  $\sqsubseteq_{FD}$ - and  $\sqsubseteq$ -. The latter turns the denotational domain of CSP into a Scott cpo [9], which yields semantics for the fixed point operator  $\mu x. f(x)$  provided that  $f$  is continuous with respect to  $\sqsubseteq$ -. Since it is possible to express deadlock-freeness and livelock-freeness as a refinement problem, the verification of properties has been reduced traditionally to a model-checking problem for a finite set of events  $A$ .

We are interested in verification techniques for arbitrary event sets  $A$  or arbitrarily parameterized processes. Such processes can be used to model dense-timed processes, processes with dynamic thread creation, and processes with unbounded thread-local variables and buffers. Events may even be higher-order objects such as functions or again processes, paving the way for the modeling of re-programmable compute servers or dynamic distributed computing architectures. However, this adds substantial complexity to the process theory: when it comes to study the interplay of different denotational models, refinement-orderings, and side-conditions for continuity, paper-and-pencil proofs easily reach their limits of precision.

Several attempts have been undertaken to develop the formal theory of CSP in an interactive proof system, mostly in Isabelle/HOL [4, 12, 7]. This work is based on the most recent instance in this line, HOL-CSP 2.0, which has been published as AFP submission [10] and whose development is hosted at <https://gitlri.lri.fr/burkhart.wolff/hol-csp2.0>.

The present AFP Module is an add-on on this work and develops some support for

1. example of induction schemes (mutual fixed-point Induction, K-induction),
2. a theory of explicit state normalisation which allows for proofs over certain communicating networks of arbitrary size.



## 1.2 The Global Architecture of CSP\_RefTk

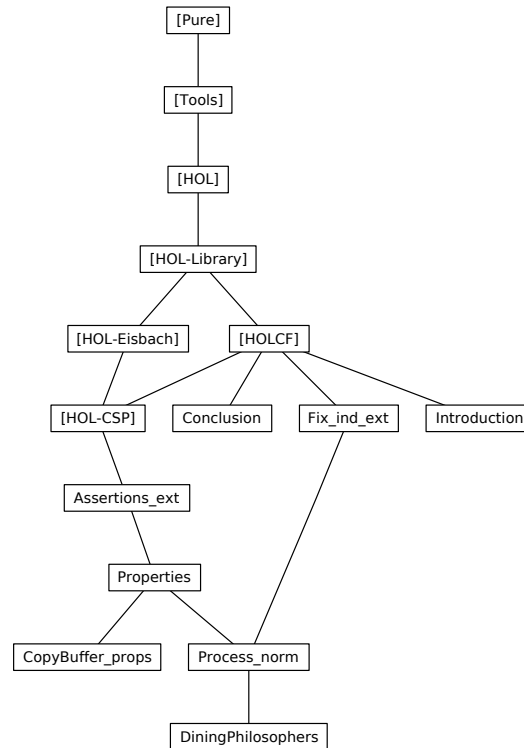


Figure 1.1: The overall architecture: HOLCF, HOL-CSP, and CSP\_RefTk

The global architecture of CSP\_RefTk 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

# Normalisation of Deterministic CSP Processes

theory *Process-norm*

imports *HOL-CSP.Assertions HOL-CSP.CSP-Induct*

begin

### 2.1 Deterministic normal-forms with explicit state

abbreviation *P-dnorm*  $\tau v \equiv (\mu X. (\lambda s. \square e \in (\tau s) \rightarrow X (v s e)))$

notation *P-dnorm* (*P<sub>norm</sub>*[[ $\tau, v$ ]] 60)

lemma *dnorm-cont[simp]*:

fixes  $\tau :: 'a :: \text{type} \Rightarrow 'b :: \text{type set}$  and  $v :: 'a \Rightarrow 'b \Rightarrow 'c$   
shows *cont*  $(\lambda X. (\lambda s. \square e \in (\tau s) \rightarrow X (v s e)))$  (*is cont* ?f)

proof –

have *cont*  $(\lambda X. ?f X s)$  for  $s$  by (*simp add: cont-fun*)  
then show ?thesis by *simp*

qed

### 2.2 Interleaving product lemma

lemma *dnorm-inter*:

fixes  $\tau_1 :: 'a_1 :: \text{type} \Rightarrow 'b_1 :: \text{type set}$  and  $\tau_2 :: 'a_2 :: \text{type} \Rightarrow 'b_2 :: \text{type set}$   
and  $v_1 :: 'a_1 \Rightarrow 'b_1 \Rightarrow 'c$  and  $v_2 :: 'a_2 \Rightarrow 'b_2 \Rightarrow 'c$

defines  $P: P \equiv P_{norm}[[\tau_1, v_1]]$  (*is*  $P \equiv \text{fix}(\lambda X. ?P X)$ )

defines  $Q: Q \equiv P_{norm}[[\tau_2, v_2]]$  (*is*  $Q \equiv \text{fix}(\lambda X. ?Q X)$ )

assumes *indep*:  $\langle \forall s_1 s_2. \tau_1 s_1 \cap \tau_2 s_2 = \{\} \rangle$

defines *Tr*:  $\tau \equiv (\lambda (s_1, s_2). \tau_1 s_1 \cup \tau_2 s_2)$

**defines**  $Up$ :  $v \equiv (\lambda(s_1, s_2) e. \text{if } e \in \tau_1 s_1 \text{ then } (v_1 s_1 e, s_2) \\ \text{else if } e \in \tau_2 s_2 \text{ then } (s_1, v_2 s_2 e) \text{ else } (s_1, s_2))$   
**defines**  $S$ :  $S \equiv P_{norm}[\tau, v]$  (**is**  $S \equiv \text{fix}(\Lambda X. ?S X)$ )

**shows**  $(P s_1 ||| Q s_2) = S (s_1, s_2)$

**proof** –

**have**  $P$ -rec:  $P = ?P P$  **using**  $\text{fix-eq}[of (\Lambda X. ?P X)] P$  **by**  $\text{simp}$   
**have**  $Q$ -rec:  $Q = ?Q Q$  **using**  $\text{fix-eq}[of (\Lambda X. ?Q X)] Q$  **by**  $\text{simp}$   
**have**  $S$ -rec:  $S = ?S S$  **using**  $\text{fix-eq}[of (\Lambda X. ?S X)] S$  **by**  $\text{simp}$   
**have**  $\text{dir1}$ :  $\forall s_1 s_2. (P s_1 ||| Q s_2) \sqsubseteq_{FD} S (s_1, s_2)$   
**proof**( $\text{subst } P, \text{subst } Q,$   
 $\text{induct rule:parallel-fix-ind-inc}[of \lambda x y. \forall s_1 s_2. (x s_1 ||| y s_2) \sqsubseteq_{FD} S (s_1, s_2)]$ )  
**case**  $\text{admissibility}$   
**then show**  $?case$   
**by** ( $\text{intro adm-all le-FD-adm}$ ) ( $\text{simp-all add: cont2cont-fun monofunI}$ )  
**next**  
**case** ( $\text{base-fst } y$ )  
**then show**  $?case$  **by** ( $\text{metis app-strict BOT-leFD Sync-BOT Sync-commute}$ )  
**next**  
**case** ( $\text{base-snd } x$ )  
**then show**  $?case$  **by** ( $\text{simp add: Sync-BOT}$ )  
**next**  
**case** ( $\text{step } x$ )  
**then show**  $?case$  (**is**  $\forall s_1 s_2. ?C s_1 s_2$ )  
**proof**( $\text{intro allI}$ )  
**fix**  $s_1 s_2$   
**show**  $?C s_1 s_2$   
**apply**  $\text{simp}$   
**apply** ( $\text{subst Mprefix-Sync-distr-indep}[\text{where } S = \{\}, \text{simplified}]$ )  
**apply** ( $\text{subst } S\text{-rec, simp add: Tr Up Mprefix-Un-distrib}$ )  
**apply** ( $\text{intro mono-Det-FD mono-Mprefix-FD}$ )  
**using**  $\text{step}(3)[\text{simplified}] \text{ indep}$  **apply**  $\text{simp}$   
**using**  $\text{step}(2)[\text{simplified}] \text{ indep}$  **by**  $\text{fastforce}$   
**qed**  
**qed**  
**have**  $\text{dir2}$ :  $\forall s_1 s_2. S (s_1, s_2) \sqsubseteq_{FD} (P s_1 ||| Q s_2)$   
**proof**( $\text{subst } S, \text{induct rule:fix-ind-k}[of \lambda x. \forall s_1 s_2. x (s_1, s_2) \sqsubseteq_{FD} (P s_1 ||| Q s_2) I]$ )  
**case**  $\text{admissibility}$   
**show**  $?case$  **by** ( $\text{intro adm-all le-FD-adm}$ ) ( $\text{simp-all add: cont-fun monofunI}$ )  
**next**  
**case**  $\text{base-k-steps}$   
**then show**  $?case$  **by**  $\text{simp}$   
**next**  
**case**  $\text{step}$   
**then show**  $?case$  (**is**  $\forall s_1 s_2. ?C s_1 s_2$ )  
**proof**( $\text{intro allI}$ )

```

fix  $s_1 s_2$ 
  have  $P\text{-rec-sym: Mprefix } (\tau_1 s_1) (\lambda e. P (v_1 s_1 e)) = P s_1$  using  $P\text{-rec}$  by
metis
  have  $Q\text{-rec-sym: Mprefix } (\tau_2 s_2) (\lambda e. Q (v_2 s_2 e)) = Q s_2$  using  $Q\text{-rec}$  by
metis
  show  $?C s_1 s_2$ 
    apply ( $\text{simp add: Tr Up Mprefix-Un-distrib}$ )
    apply ( $\text{subst } P\text{-rec, subst } Q\text{-rec, subst Mprefix-Sync-distr-indep}$  where
 $S = \{\}, \text{ simplified}$ )
    apply ( $\text{intro mono-Det-FD mono-Mprefix-FD}$ )
    apply ( $\text{subst } Q\text{-rec-sym, simp add: step[simplified]}$ )
    apply ( $\text{subst } P\text{-rec-sym}$ ) using  $\text{step[simplified]}$  indep by  $\text{fastforce}$ 
  qed
qed
from  $\text{dir1 dir2}$  show  $?thesis$  using  $\text{FD-antisym}$  by  $\text{blast}$ 
qed

```

## 2.3 Synchronous product lemma

**lemma**  $\text{dnorm-par}$ :

```

fixes  $\tau_1 :: '\sigma_1::\text{type} \Rightarrow 'event::\text{type set}$  and  $\tau_2 :: '\sigma_2::\text{type} \Rightarrow 'event \text{ set}$ 
and  $v_1 :: '\sigma_1 \Rightarrow 'event \Rightarrow '\sigma_1$  and  $v_2 :: '\sigma_2 \Rightarrow 'event \Rightarrow '\sigma_2$ 
defines  $P: P \equiv P_{\text{norm}}[\tau_1, v_1]$  (is  $P \equiv \text{fix} \cdot (\Lambda X. ?P X)$ )
defines  $Q: Q \equiv P_{\text{norm}}[\tau_2, v_2]$  (is  $Q \equiv \text{fix} \cdot (\Lambda X. ?Q X)$ )

```

```

defines  $\text{Tr}: \tau \equiv (\lambda(s_1, s_2). \tau_1 s_1 \cap \tau_2 s_2)$ 
defines  $\text{Up}: v \equiv (\lambda(s_1, s_2) e. (v_1 s_1 e, v_2 s_2 e))$ 
defines  $S: S \equiv P_{\text{norm}}[\tau, v]$  (is  $S \equiv \text{fix} \cdot (\Lambda X. ?S X)$ )

```

**shows**  $(P s_1 \parallel Q s_2) = S (s_1, s_2)$

**proof** –

```

have  $P\text{-rec}: P = ?P P$  using  $\text{fix-eq[of } (\Lambda X. ?P X)] P$  by  $\text{simp}$ 
have  $Q\text{-rec}: Q = ?Q Q$  using  $\text{fix-eq[of } (\Lambda X. ?Q X)] Q$  by  $\text{simp}$ 
have  $S\text{-rec}: S = ?S S$  using  $\text{fix-eq[of } (\Lambda X. ?S X)] S$  by  $\text{simp}$ 
have  $\text{dir1}: \forall s_1 s_2. (P s_1 \parallel Q s_2) \sqsubseteq_{FD} S (s_1, s_2)$ 
proof( $\text{subst } P, \text{ subst } Q,$ 
   $\text{induct rule: parallel-fix-ind[of } \lambda x y. \forall s_1 s_2. (x s_1 \parallel y s_2) \sqsubseteq_{FD} S (s_1, s_2)]$ )
  case  $\text{adm:1}$ 
    then show  $?case$ 
    by ( $\text{intro adm-all le-FD-adm}$ ) ( $\text{simp-all add: cont2cont-fun monofunI}$ )
  next
    case  $\text{base:2}$ 
    then show  $?case$  by ( $\text{simp add: Sync-BOT}$ )
  next
    case  $\text{step:}(3 x y)$ 
    then show  $?case$  (is  $\forall s_1 s_2. ?C s_1 s_2$ )
    proof( $\text{intro allI}$ )
      fix  $s_1 s_2$ 

```

```

show ?C s1 s2
  apply(simp)
  apply(subst Mprefix-Sync-distr-subset[where S=UNIV, simplified])
  apply(subst S-rec, simp add: Tr Up Mprefix-Un-distrib)
  by (simp add: step)
qed
qed
have dir2:  $\forall s_1 s_2. S (s_1, s_2) \sqsubseteq_{FD} (P s_1 \parallel Q s_2)$ 
  proof(subst S, induct rule:fix-ind-k[of  $\lambda x. \forall s_1 s_2. x (s_1, s_2) \sqsubseteq_{FD} (P s_1 \parallel Q s_2) I$ ])
    case admissibility
    show ?case by (intro adm-all le-FD-adm) (simp-all add: cont-fun monofunI)

next
  case base-k-steps
  then show ?case by simp
next
  case step
  then show ?case (is  $\forall s_1 s_2. ?C s_1 s_2$ )
  proof(intro allI)
    fix s1 s2
    have P-rec-sym:Mprefix ( $\tau_1 s_1$ ) ( $\lambda e. P (v_1 s_1 e)$ ) = P s1 using P-rec by
metis
    have Q-rec-sym:Mprefix ( $\tau_2 s_2$ ) ( $\lambda e. Q (v_2 s_2 e)$ ) = Q s2 using Q-rec by
metis
    show ?C s1 s2
    apply(simp add: Tr Up)
    apply(subst P-rec, subst Q-rec, subst Mprefix-Sync-distr-subset[where
S=UNIV, simplified])
    apply(rule mono-Mprefix-FD)
    using step by auto
    qed
  qed
from dir1 dir2 show ?thesis using FD-antisym by blast
qed

```

## 2.4 Consequences

```

inductive-set  $\mathfrak{R}$  for  $\tau :: 'a::type \Rightarrow 'event::type\ set$ 
  and  $v :: 'a \Rightarrow 'event \Rightarrow 'a$ 
  and  $\sigma_0 :: 'a$ 
where rbase:  $\sigma_0 \in \mathfrak{R} \tau v \sigma_0$ 
  | rstep:  $s \in \mathfrak{R} \tau v \sigma_0 \Longrightarrow e \in \tau s \Longrightarrow v s e \in \mathfrak{R} \tau v \sigma_0$ 

```

— Deadlock freeness

**lemma** deadlock-free-dnorm- :

**fixes**  $\tau :: 'a::type \Rightarrow 'event::type\ set$

```

and  $v :: 'σ ⇒ 'event ⇒ 'σ$ 
and  $σ_0 :: 'σ$ 
assumes non-reachable-sink:  $∀ s ∈ \mathfrak{R} \ τ \ v \ σ_0. \ τ \ s \neq \{\}$ 
defines  $P$ :  $P ≡ P_{norm} \llbracket \tau, v \rrbracket$  (is  $P ≡ \text{fix} \cdot (\Lambda X. \ ?P \ X)$ )
shows  $s ∈ \mathfrak{R} \ \tau \ v \ \sigma_0 \implies \text{deadlock-free-v2} \ (P \ s)$ 
proof(unfold deadlock-free-v2-FD DF_SKIP-def, induct arbitrary:s rule:fix-ind)
  show  $\text{adm} \ (\lambda a. \ \forall x. \ x \in \mathfrak{R} \ \tau \ v \ \sigma_0 \longrightarrow a \sqsubseteq_{FD} P \ x)$  by (simp add: monofun-def)
next
  fix  $s :: 'σ$ 
  show  $s \in \mathfrak{R} \ \tau \ v \ \sigma_0 \implies \perp \sqsubseteq_{FD} P \ s$  by simp
next
  fix  $s :: 'σ$  and  $x :: 'event \ \text{process}$ 
  have  $P\text{-rec}$ :  $P = \ ?P \ P$  using fix-eq[of  $(\Lambda X. \ ?P \ X)$ ] P by simp
  assume  $1$ :  $\bigwedge s. \ s \in \mathfrak{R} \ \tau \ v \ \sigma_0 \implies x \sqsubseteq_{FD} P \ s$ 
  and  $2$ :  $s \in \mathfrak{R} \ \tau \ v \ \sigma_0$ 
  from  $1 \ 2$  show  $(\Lambda x. \ (\sqcap x a \in UNIV \rightarrow x) \sqcap SKIP) \cdot x \sqsubseteq_{FD} P \ s$ 
  apply (subst P-rec, rule-tac trans-FD[rotated, OF Mprefix-refines-Mndetprefix-FD])
  apply simp
  apply (rule mono-Ndet-FD-left)
  apply (rule trans-FD[OF mono-Mndetprefix-FD-set[of  $\langle \tau \ s \rangle \langle UNIV \rangle$ ] mono-Mndetprefix-FD[rule-format, OF 1]])
  using non-reachable-sink[rule-format, OF 2] apply assumption
  by blast (meson \mathfrak{R}.rstep)
qed

```

**lemmas** *deadlock-free-dnorm = deadlock-free-dnorm-[rotated, OF rbase, rule-format]*

**end**





# Chapter 3

## Examples

### 3.1 CopyBuffer Refinement over an infinite alphabet

```
theory CopyBuffer-props
  imports HOL-CSP.CopyBuffer HOL-CSP.Assertions
begin
```

#### 3.1.1 The Copy-Buffer vs. reference processes

```
lemma DF-COPY: (DF (range left  $\cup$  range right))  $\sqsubseteq_{FD}$  COPY
  by (simp add: DF-COPY)
```

#### 3.1.2 ... and abstract consequences

```
corollary df-COPY: deadlock-free COPY
  and lf-COPY: lifelock-free COPY
  apply (meson DF-COPY DF-Univ-freeness UNIV-not-empty image-is-empty
sup-eq-bot-iff)
  by (meson CHAOS-DF-refine-FD DF-COPY DF-Univ-freeness UNIV-not-empty
deadlock-free-def
image-is-empty lifelock-free-def sup-eq-bot-iff trans-FD)
```

```
corollary df-v2-COPY: deadlock-free-v2 COPY
  and lf-v2-COPY: lifelock-free-v2 COPY
  and nt-COPY: non-terminating COPY
  apply (simp add: df-COPY deadlock-free-is-deadlock-free-v2)
  apply (simp add: lf-COPY lifelock-free-is-lifelock-free-v2)
  using lf-COPY lifelock-free-is-non-terminating by blast
```

```
lemma DF-SYSTEM: DF UNIV  $\sqsubseteq_{FD}$  SYSTEM
  using DF-subset[of (range left  $\cup$  range right) UNIV, simplified]
DF-COPY impl-refines-spec trans-FD by blast
```

```
corollary df-SYSTEM: deadlock-free SYSTEM
```

```

and lf-SYSTEM: lifelock-free SYSTEM
apply (simp add: DF-SYSTEM deadlock-free-def)
using CHAOS-DF-refine-FD DF-SYSTEM lifelock-free-def trans-FD by blast

corollary df-v2-SYSTEM: deadlock-free-v2 SYSTEM
and lf-v2-SYSTEM: lifelock-free-v2 SYSTEM
and nt-SYSTEM: non-terminating SYSTEM
apply (simp add: df-SYSTEM deadlock-free-is-deadlock-free-v2)
apply (simp add: lf-SYSTEM lifelock-free-is-lifelock-free-v2)
using lf-SYSTEM lifelock-free-is-non-terminating by blast

end

```

## 3.2 Generalized Dining Philosophers

```

theory   DiningPhilosophers
imports Process-norm
begin

```

### 3.2.1 Preliminary lemmas for proof automation

```

lemma Suc-mod:  $n > 1 \implies i \neq \text{Suc } i \text{ mod } n$ 
by (metis One-nat-def mod-Suc mod-if mod-mod-trivial n-not-Suc-n)

```

```

lemmas suc-mods = Suc-mod Suc-mod[symmetric]

```

```

lemma l-suc:  $n > 1 \implies \neg n \leq \text{Suc } 0$ 
by simp

```

```

lemma minus-suc:  $n > 0 \implies n - \text{Suc } 0 \neq n$ 
by linarith

```

```

lemma numeral-4-eq-4:  $4 = \text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0)))$ 
by simp

```

```

lemma numeral-5-eq-5:  $5 = \text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0))))$ 
by simp

```

### 3.2.2 The dining processes definition

```

locale DiningPhilosophers =

```

```

  fixes N::nat
  assumes N-g1[simp] :  $N > 1$ 

```

```

begin

```

```

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

```

**definition** *RPHIL*:: *nat*  $\Rightarrow$  *dining-event process*

**where** *RPHIL* *i* =  $(\mu X. (\text{picks } i \ i \rightarrow (\text{picks } i \ (i-1) \rightarrow (\text{putsdown } i \ (i-1) \rightarrow (\text{putsdown } i \ i \rightarrow X))))))$

**definition** *LPHIL0*:: *dining-event process*

**where** *LPHIL0* =  $(\mu X. (\text{picks } 0 \ (N-1) \rightarrow (\text{picks } 0 \ 0 \rightarrow (\text{putsdown } 0 \ 0 \rightarrow (\text{putsdown } 0 \ (N-1) \rightarrow X))))))$

**definition** *FORK* :: *nat*  $\Rightarrow$  *dining-event process*

**where** *FORK* *i* =  $(\mu X. (\text{picks } i \ i \rightarrow (\text{putsdown } i \ i \rightarrow X)) \square (\text{picks } ((i+1) \ \text{mod } N) \ i \rightarrow (\text{putsdown } ((i+1) \ \text{mod } N) \ i \rightarrow X)))$

**abbreviation** *foldPHILs* *n*  $\equiv$  *fold*  $(\lambda \ i \ P. P \ ||| \ \text{RPHIL } i) \ [1..< \ n] \ (\text{LPHIL0})$

**abbreviation** *foldFORKs* *n*  $\equiv$  *fold*  $(\lambda \ i \ P. P \ ||| \ \text{FORK } i) \ [1..< \ n] \ (\text{FORK } 0)$

**abbreviation** *PHILs*  $\equiv$  *foldPHILs* *N*

**abbreviation** *FORKs*  $\equiv$  *foldFORKs* *N*

**corollary** *FORKs-def2*: *FORKs* = *fold*  $(\lambda \ i \ P. P \ ||| \ \text{FORK } i) \ [0..< \ N] \ \text{SKIP}$

**using** *N-g1* **by** (*subst* (2) *upt-rec*, *auto*) (*metis* (*no-types*, *lifting*) *Inter-commute Inter-SKIP*)

**corollary** *N = 3*  $\implies$  *PHILs* = (*LPHIL0*  $|||$  *RPHIL* 1  $|||$  *RPHIL* 2)

**by** (*subst* *upt-rec*, *auto* *simp* *add:numeral-2-eq-2*) $+$

**definition** *DINING* :: *dining-event process*

**where** *DINING* = (*FORKs*  $||$  *PHILs*)

## Unfolding rules

**lemma** *RPHIL-rec*:

*RPHIL* *i* =  $(\text{picks } i \ i \rightarrow (\text{picks } i \ (i-1) \rightarrow (\text{putsdown } i \ (i-1) \rightarrow (\text{putsdown } i \ i \rightarrow \text{RPHIL } i))))$

**by** (*simp* *add:RPHIL-def* *write0-def*, *subst* *fix-eq*, *simp*)

**lemma** *LPHIL0-rec*:

*LPHIL0* =  $(\text{picks } 0 \ (N-1) \rightarrow (\text{picks } 0 \ 0 \rightarrow (\text{putsdown } 0 \ 0 \rightarrow (\text{putsdown } 0 \ (N-1) \rightarrow \text{LPHIL0}))))$

**by** (*simp* *add:LPHIL0-def* *write0-def*, *subst* *fix-eq*, *simp*)

**lemma** *FORK-rec*: *FORK* *i* =  $(\text{picks } i \ i \rightarrow (\text{putsdown } i \ i \rightarrow (\text{FORK } i)))$

$\square (\text{picks } ((i+1) \ \text{mod } N) \ i \rightarrow (\text{putsdown } ((i+1) \ \text{mod } N) \ i \rightarrow (\text{FORK } i)))$

**by** (*simp* *add:FORK-def* *write0-def*, *subst* *fix-eq*, *simp*)

### 3.2.3 Translation into normal form

**lemma**  $N\text{-pos}[simp]$ :  $N > 0$   
**using**  $N\text{-g1}$   $neq0\text{-conv}$  **by**  $blast$

**lemmas**  $N\text{-pos-simps}[simp] = suc\text{-mods}[OF\ N\text{-g1}]$   $l\text{-suc}[OF\ N\text{-g1}]$   $minus\text{-suc}[OF\ N\text{-pos}]$

The one-fork process

**type-synonym**  $id_{fork} = nat$

**type-synonym**  $\sigma_{fork} = nat$

**definition**  $fork\text{-transitions}:: id_{fork} \Rightarrow \sigma_{fork} \Rightarrow dining\text{-event set } (Tr_f)$   
**where**  $Tr_f\ i\ s =$  (if  $s = 0$  then  $\{picks\ i\ i\} \cup \{picks\ ((i+1)\ mod\ N)\ i\}$   
 else if  $s = 1$  then  $\{putsdown\ i\ i\}$   
 else if  $s = 2$  then  $\{putsdown\ ((i+1)\ mod\ N)\ i\}$   
 else  $\{\}$ )  
**declare**  $Un\text{-insert-right}[simp\ del]$   $Un\text{-insert-left}[simp\ del]$

**lemma**  $ev\text{-}id_{fork}x[simp]$ :  $e \in Tr_f\ i\ s \implies fork\ e = i$   
**by** ( $auto\ simp\ add:fork\text{-transitions-def}\ split:if\text{-splits}$ )

**definition**  $\sigma_{fork}\text{-update}:: id_{fork} \Rightarrow \sigma_{fork} \Rightarrow dining\text{-event} \Rightarrow \sigma_{fork}$  ( $Up_f$ )  
**where**  $Up_f\ i\ s\ e =$  ( if  $e = (picks\ i\ i)$  then  $1$   
 else if  $e = (picks\ ((i+1)\ mod\ N)\ i)$  then  $2$   
 else  $0$  )

**definition**  $FORK_{norm}:: id_{fork} \Rightarrow \sigma_{fork} \Rightarrow dining\text{-event process}$   
**where**  $FORK_{norm}\ i = P_{norm}[[Tr_f\ i, Up_f\ i]]$

**lemma**  $FORK_{norm}\text{-rec}$ :  $FORK_{norm}\ i = (\lambda\ s.\ \square\ e \in (Tr_f\ i\ s) \rightarrow FORK_{norm}\ i$   
 $(Up_f\ i\ s\ e))$   
**using**  $fix\text{-eq}[of\ \Lambda\ X.\ (\lambda\ s.\ Mprefix\ (Tr_f\ i\ s)\ (\lambda\ e.\ X\ (Up_f\ i\ s\ e)))]$   $FORK_{norm}\text{-def}$   
**by**  $simp$

**lemma**  $FORK\text{-refines}\text{-}FORK_{norm}$ :  $FORK_{norm}\ i\ 0 \sqsubseteq_{FD}\ FORK\ i$

**proof**( $unfold\ FORK_{norm}\text{-def}$ ,  
 $induct\ rule:fix\text{-ind}\text{-}k\text{-skip}$ [**where**  $k=2$  **and**  $f=\Lambda\ x.(\lambda\ s.\ Mprefix\ (Tr_f\ i\ s)\ (\lambda\ e.\ x\ (Up_f\ i\ s\ e)))$ ])  
**case**  $lower\text{-bound}$   
**then show**  $?case$  **by**  $simp$   
**next**  
**case**  $admissibility$   
**then show**  $?case$  **by** ( $simp\ add: cont\text{-fun}\ monofunI$ )  
**next**  
**case**  $base\text{-}k\text{-steps}$   
**then show**  $?case$  (**is**  $\forall j < 2.$  ( $iterate\ j.\ ?f.\ \perp$ )  $0 \sqsubseteq_{FD}\ FORK\ i$ )  
**proof** –

```

have less-2:  $\bigwedge j. (j::nat) < 2 = (j = 0 \vee j = 1)$  by linarith
moreover have (iterate 0.?f.⊥) 0  $\sqsubseteq_{FD}$  FORK i by simp
moreover have (iterate 1.?f.⊥) 0  $\sqsubseteq_{FD}$  FORK i
by (subst FORK-rec)
      (simp add: write0-def fork-transitions-def Mprefix-Un-distrib)
ultimately show ?thesis by simp
qed
next
case (step x)
then show ?case (is (iterate 2.?f.x) 0  $\sqsubseteq_{FD}$  FORK i)
by (subst FORK-rec)
      (simp add: write0-def numeral-2-eq-2 fork-transitions-def  $\sigma_{fork}$ -update-def
Mprefix-Un-distrib)
qed

```

```

lemma FORKnorm-refines-FORK: FORK i  $\sqsubseteq_{FD}$  FORKnorm i 0
proof(unfold FORK-def,
      induct rule:fix-ind-k-skip[where k=1])
show (1::nat)  $\leq 1$  by simp
next
show adm ( $\lambda a. a \sqsubseteq_{FD}$  FORKnorm i 0) by (simp add: monofunI)
next
case base-k-steps
show ?case by simp
next
case (step x)
then show ?case (is (iterate 1.?f.x)  $\sqsubseteq_{FD}$  FORKnorm i 0)
apply (subst FORKnorm-rec)
apply (simp add: write0-def fork-transitions-def  $\sigma_{fork}$ -update-def Mprefix-Un-distrib)
by (subst (1 2) FORKnorm-rec)
      (simp add: fork-transitions-def  $\sigma_{fork}$ -update-def Mprefix-Un-distrib)
qed

```

```

lemma FORKnorm-is-FORK: FORK i = FORKnorm i 0
using FORK-refines-FORKnorm FORKnorm-refines-FORK FD-antisym by blast

```

The all-forks process in normal form

```

type-synonym  $\sigma_{forks} = nat\ list$ 

```

```

definition forks-transitions:: nat  $\Rightarrow$   $\sigma_{forks} \Rightarrow$  dining-event set ( $Tr_F$ )
where  $Tr_F\ n\ fs = (\bigcup_{i < n}. Tr_f\ i\ (fs!i))$ 

```

```

lemma forks-transitions-take:  $Tr_F\ n\ fs = Tr_F\ n\ (take\ n\ fs)$ 
by (simp add: forks-transitions-def)

```

```

definition  $\sigma_{forks}$ -update::  $\sigma_{forks} \Rightarrow$  dining-event  $\Rightarrow$   $\sigma_{forks}$  ( $Up_F$ )
where  $Up_F\ fs\ e = (let\ i=(fork\ e)\ in\ fs[i:=(Up_f\ i\ (fs!i)\ e)])$ 

```

**lemma** *forks-update-take*:  $\text{take } n \ (Up_F \ fs \ e) = Up_F \ (\text{take } n \ fs) \ e$   
**unfolding**  $\sigma_{\text{forks-update-def}}$   
**by** (*metis nat-less-le nat-neq-iff nth-take order-refl take-update-cancel take-update-swap*)

**definition**  $FORKs_{norm} :: \text{nat} \Rightarrow \sigma_{\text{forks}} \Rightarrow \text{dining-event process}$   
**where**  $FORKs_{norm} \ n = P_{norm} \llbracket Tr_F \ n, Up_F \rrbracket$

**lemma** *FORKs<sub>norm</sub>-rec*:  $FORKs_{norm} \ n = (\lambda \ fs. \square \ e \in (Tr_F \ n \ fs) \rightarrow FORKs_{norm} \ n \ (Up_F \ fs \ e))$   
**using** *fix-eq*[of  $\Lambda \ X. (\lambda \ fs. Mprefix \ (Tr_F \ n \ fs) \ (\lambda \ e. X \ (Up_F \ fs \ e)))$ ]  
**by** *simp*

**lemma** *FORKs<sub>norm</sub>-0*:  $FORKs_{norm} \ 0 \ fs = STOP$   
**by** (*subst FORKs<sub>norm</sub>-rec, simp add: forks-transitions-def Mprefix-STOP*)

**lemma** *FORKs<sub>norm</sub>-1-dir1*:  $\text{length } fs > 0 \Longrightarrow FORKs_{norm} \ 1 \ fs \sqsubseteq_{FD} (FORK_{norm} \ 0 \ (fs!0))$

**proof**(*unfold FORKs<sub>norm</sub>-def,*  
*induct arbitrary:fs rule:fix-ind-k*[**where**  $k=1$   
**and**  $f=\Lambda \ x. (\lambda \ fs. Mprefix \ (Tr_F \ 1 \ fs) \ (\lambda \ e. x \ (Up_F \ fs \ e)))$ ])  
**case** *admissibility*  
**then show** *?case* **by** (*simp add: cont-fun monofunI*)  
**next**  
**case** *base-k-steps*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*step X*)  
**hence**  $(\bigcup_{i < Suc \ 0. Tr_f \ i \ (fs \ ! \ i)} = Tr_f \ 0 \ (fs \ ! \ 0))$  **by** *auto*  
**with** *step show ?case*  
**apply** (*subst FORK<sub>norm</sub>-rec, simp add:  $\sigma_{\text{forks-update-def}}$  forks-transitions-def*)  
**apply** (*intro mono-Mprefix-FD, safe*)  
**by** (*metis ev-id<sub>fork</sub>x step.prem list-update-nonempty nth-list-update-eq*)  
**qed**

**lemma** *FORKs<sub>norm</sub>-1-dir2*:  $\text{length } fs > 0 \Longrightarrow (FORK_{norm} \ 0 \ (fs!0)) \sqsubseteq_{FD} FORKs_{norm} \ 1 \ fs$

**proof**(*unfold FORK<sub>norm</sub>-def,*  
*induct arbitrary:fs rule:fix-ind-k*[**where**  $k=1$   
**and**  $f=\Lambda \ x. (\lambda \ s. Mprefix \ (Tr_f \ 0 \ s) \ (\lambda \ e. x \ (Up_f \ 0 \ s \ e)))$ ])  
**case** *admissibility*  
**then show** *?case* **by** (*simp add: cont-fun monofunI*)  
**next**  
**case** *base-k-steps*

```

then show ?case by simp
next
  case (step X)
  have ( $\bigcup_{i < \text{Suc } 0}. \text{Tr}_f i (fs ! i) = \text{Tr}_f 0 (fs ! 0)$ ) by auto
  with step show ?case
  apply (subst FORKsnorm-rec, simp add:  $\sigma_{\text{forks-update-def forks-transitions-def}}$ )

  apply (intro mono-Mprefix-FD, safe)
  by (metis ev-idforkx step.premis list-update-nonempty nth-list-update-eq)
qed

```

```

lemma FORKsnorm-1: length fs > 0  $\implies$  (FORKnorm 0 (fs!0)) = FORKsnorm 1
fs
using FORKsnorm-1-dir1 FORKsnorm-1-dir2 FD-antisym by blast

```

```

lemma FORKsnorm-unfold:
0 < n  $\implies$  length fs = Suc n  $\implies$ 
  FORKsnorm (Suc n) fs = (FORKsnorm n (butlast
fs)|||(FORKnorm n (fs!n)))
proof(rule FD-antisym)
  show 0 < n  $\implies$  length fs = Suc n  $\implies$ 
  FORKsnorm (Suc n) fs  $\sqsubseteq_{FD}$  (FORKsnorm n (butlast
fs)|||FORKnorm n (fs!n))
  proof(subst FORKsnorm-def,
    induct arbitrary:fs
    rule:fix-ind-k[where k=1
    and f= $\Lambda x. (\lambda fs. \text{Mprefix} (\text{Tr}_F (\text{Suc } n) fs) (\lambda e. x (\text{Up}_F fs
e)))]$ ))
    case admissibility
    then show ?case by (simp add: cont-fun monofunI)
  next
    case base-k-steps
    then show ?case by simp
  next
    case (step X)
    have indep: $\forall s_1 s_2. \text{Tr}_F n s_1 \cap \text{Tr}_f n s_2 = \{\}$ 
    by (auto simp add:forks-transitions-def fork-transitions-def)
    from step show ?case
    apply (auto simp add:indep dnorm-inter FORKsnorm-def FORKnorm-def)
    apply (subst fix-eq, simp add:forks-transitions-def Un-commute lessThan-Suc
nth-butlast)
    proof(rule mono-Mprefix-FD, safe, goal-cases)
      case (1 e)
      from 1(4) have a:fork e = n
      by (auto simp add:fork-transitions-def split:if-splits)
      show ?case
      using 1(1)[rule-format, of (UpF fs e)]
      apply (simp add: 1 a butlast-list-update  $\sigma_{\text{forks-update-def}}$ )

```

```

    by (metis 1(4) ev-idforkx lessThan-iff less-not-refl)
  next
  case (2 e m)
  hence a:e ∉ Trf n (fs ! n)
    using ev-idforkx by fastforce
  hence c:UpF fs e ! n = fs ! n
    by (metis 2(4) ev-idforkx σforks-update-def nth-list-update-peq)
  have d:UpF (butlast fs) e = butlast (UpF fs e)
    apply (simp add:σforks-update-def)
  by (metis butlast-conv-take σforks-update-def forks-update-take length-list-update)
  from 2 a show ?case
    using 2(1)[rule-format, of (UpF fs e)] c d σforks-update-def by auto
  qed
  qed
  next
  have indep:∀ s1 s2. TrF n s1 ∩ Trf n s2 = {}
    by (auto simp add: forks-transitions-def fork-transitions-def)
  show 0 < n ⇒ length fs = Suc n ⇒
    (FORKsnorm n (butlast fs)|||FORKnorm n (fs!n)) ⊆FD
    FORKsnorm (Suc n) fs
    apply (subst FORKsnorm-def, auto simp add:indep dnorm-inter FORKnorm-def)
  proof (rule fix-ind[where
    P=λa. 0 < n ⇒ (∀ x. length x = Suc n ⇒ a (butlast x, x ! n) ⊆FD
    FORKsnorm (Suc n) x),
    rule-format], simp-all, goal-cases)
    case base:1
  then show ?case by (simp add: cont-fun monofunI)
  next
  case step:(2 X fs)
  then show ?case
    apply (unfold FORKsnorm-def, subst fix-eq, simp add: forks-transitions-def
    Un-commute lessThan-Suc nth-butlast)
  proof (rule mono-Mprefix-FD, safe, goal-cases)
    case (1 e)
  from 1(6) have a:fork e = n
    by (auto simp add: fork-transitions-def split:if-splits)
  show ?case
    using 1(1)[rule-format, of (UpF fs e)]
    apply (simp add: 1 a butlast-list-update σforks-update-def)
    using a ev-idforkx by blast
  next
  case (2 e m)
  have a:UpF (butlast fs) e = butlast (UpF fs e)
    apply (simp add:σforks-update-def)
  by (metis butlast-conv-take σforks-update-def forks-update-take length-list-update)
  from 2 show ?case
    using 2(1)[rule-format, of (UpF fs e)] a σforks-update-def by auto
  qed
  qed

```



qed

**lemma** *ft*:  $0 < n \implies \text{FORKs}_{\text{norm}}\ n\ (\text{replicate}\ n\ 0) = \text{foldFORKs}\ n$

**proof** (*induct* *n*, *simp*)

**case** (*Suc* *n*)

**then show** *?case*

**apply** (*auto simp add: FORKs<sub>norm</sub>-unfold FORK<sub>norm</sub>-is-FORK*)

**apply** (*metis Suc-le-D butlast-snoc replicate-Suc replicate-append-same*)

**by** (*metis FORKs<sub>norm</sub>-1 One-nat-def leI length-replicate less-Suc0 nth-replicate replicate-Suc*)

qed

**corollary** *FORKs-is-FORKs<sub>norm</sub>*:  $\text{FORKs}_{\text{norm}}\ N\ (\text{replicate}\ N\ 0) = \text{FORKs}$

**using** *ft N-pos* **by** *simp*

The one-philosopher process in normal form:

**type-synonym** *phil-id* = *nat*

**type-synonym** *phil-state* = *nat*

**definition** *rphil-transitions*:: *phil-id*  $\Rightarrow$  *phil-state*  $\Rightarrow$  *dining-event set* (*Tr<sub>rp</sub>*)

**where**  $\text{Tr}_{rp}\ i\ s = (\text{if } s = 0 \text{ then } \{\text{picks } i\ i\}$   
 $\text{else if } s = 1 \text{ then } \{\text{picks } i\ (i-1)\}$   
 $\text{else if } s = 2 \text{ then } \{\text{putsdown } i\ (i-1)\}$   
 $\text{else if } s = 3 \text{ then } \{\text{putsdown } i\ i\}$   
 $\text{else } \{\}$ )

**definition** *lphil0-transitions*:: *phil-state*  $\Rightarrow$  *dining-event set* (*Tr<sub>lp</sub>*)

**where**  $\text{Tr}_{lp}\ s = (\text{if } s = 0 \text{ then } \{\text{picks } 0\ (N-1)\}$   
 $\text{else if } s = 1 \text{ then } \{\text{picks } 0\ 0\}$   
 $\text{else if } s = 2 \text{ then } \{\text{putsdown } 0\ 0\}$   
 $\text{else if } s = 3 \text{ then } \{\text{putsdown } 0\ (N-1)\}$   
 $\text{else } \{\}$ )

**corollary** *rphil-phil*:  $e \in \text{Tr}_{rp}\ i\ s \implies \text{phil } e = i$

**and** *lphil0-phil*:  $e \in \text{Tr}_{lp}\ s \implies \text{phil } e = 0$

**by** (*simp-all add:rphil-transitions-def lphil0-transitions-def split:if-splits*)

**definition** *rphil-state-update*:: *id<sub>fork</sub>*  $\Rightarrow$   *$\sigma_{\text{fork}}$*   $\Rightarrow$  *dining-event*  $\Rightarrow$   *$\sigma_{\text{fork}}$*  (*Up<sub>rp</sub>*)

**where**  $\text{Up}_{rp}\ i\ s\ e = (\text{if } e = (\text{picks } i\ i) \text{ then } 1$   
 $\text{else if } e = (\text{picks } i\ (i-1)) \text{ then } 2$   
 $\text{else if } e = (\text{putsdown } i\ (i-1)) \text{ then } 3$   
 $\text{else } 0)$

**definition** *lphil0-state-update*::  *$\sigma_{\text{fork}}$*   $\Rightarrow$  *dining-event*  $\Rightarrow$   *$\sigma_{\text{fork}}$*  (*Up<sub>lp</sub>*)

**where**  $\text{Up}_{lp}\ s\ e = (\text{if } e = (\text{picks } 0\ (N-1)) \text{ then } 1$   
 $\text{else if } e = (\text{picks } 0\ 0) \text{ then } 2$   
 $\text{else if } e = (\text{putsdown } 0\ 0) \text{ then } 3$   
 $\text{else } 0)$

**definition**  $RPHIL_{norm}$ ::  $id_{fork} \Rightarrow \sigma_{fork} \Rightarrow$  dining-event process  
 where  $RPHIL_{norm} \ i = P_{norm} \llbracket Tr_{rp} \ i, Up_{rp} \ i \rrbracket$

**definition**  $LPHILO_{norm}$ ::  $\sigma_{fork} \Rightarrow$  dining-event process  
 where  $LPHILO_{norm} = P_{norm} \llbracket Tr_{lp}, Up_{lp} \rrbracket$

**lemma**  $RPHIL_{norm}$ -rec:  $RPHIL_{norm} \ i = (\lambda \ s. \square \ e \in (Tr_{rp} \ i \ s) \rightarrow RPHIL_{norm} \ i \ (Up_{rp} \ i \ s \ e))$   
 using fix-eq[of  $\Lambda \ X. (\lambda \ s. Mprefix \ (Tr_{rp} \ i \ s) \ (\lambda \ e. X \ (Up_{rp} \ i \ s \ e)))$ ]  $RPHIL_{norm}$ -def  
 by simp

**lemma**  $LPHILO_{norm}$ -rec:  $LPHILO_{norm} = (\lambda \ s. \square \ e \in (Tr_{lp} \ s) \rightarrow LPHILO_{norm} \ (Up_{lp} \ s \ e))$   
 using fix-eq[of  $\Lambda \ X. (\lambda \ s. Mprefix \ (Tr_{lp} \ s) \ (\lambda \ e. X \ (Up_{lp} \ s \ e)))$ ]  $LPHILO_{norm}$ -def  
 by simp

**lemma**  $RPHIL$ -refines- $RPHIL_{norm}$ :

assumes  $i$ -pos:  $i > 0$

shows  $RPHIL_{norm} \ i \ 0 \sqsubseteq_{FD} RPHIL \ i$

**proof**(unfold  $RPHIL_{norm}$ -def,

induct rule:fix-ind-k-skip[where  $k=4$  and  $f=\lambda \ x. (\lambda \ s. Mprefix \ (Tr_{rp} \ i \ s) \ (\lambda \ e. x \ (Up_{rp} \ i \ s \ e)))$ ])

case lower-bound

then show ?case (is  $1 \leq 4$ ) by simp

next

case admissibility

then show ?case (is adm ( $\lambda \ a. a \ 0 \sqsubseteq_{FD} RPHIL \ i$ ))

by (simp add: cont-fun monofunI)

next

case base-k-steps

then show ?case (is  $\forall j < 4. (iterate \ j \cdot ?f \cdot \perp) \ 0 \sqsubseteq_{FD} RPHIL \ i$ )

**proof** –

have less-2: $\bigwedge j. (j::nat) < 4 = (j = 0 \vee j = 1 \vee j = 2 \vee j = 3)$  by linarith

moreover have  $(iterate \ 0 \cdot ?f \cdot \perp) \ 0 \sqsubseteq_{FD} RPHIL \ i$  by simp

moreover have  $(iterate \ 1 \cdot ?f \cdot \perp) \ 0 \sqsubseteq_{FD} RPHIL \ i$

by (subst  $RPHIL$ -rec) (simp add: write0-def rphil-transitions-def)

moreover have  $(iterate \ 2 \cdot ?f \cdot \perp) \ 0 \sqsubseteq_{FD} RPHIL \ i$

by (subst  $RPHIL$ -rec)

(simp add: numeral-2-eq-2 write0-def rphil-transitions-def rphil-state-update-def)

moreover have  $(iterate \ 3 \cdot ?f \cdot \perp) \ 0 \sqsubseteq_{FD} RPHIL \ i$

by (subst  $RPHIL$ -rec) (simp add: numeral-3-eq-3 write0-def rphil-transitions-def

$rphil$ -state-update-def minus-suc[OF  $i$ -pos])

ultimately show ?thesis by simp

qed

next

case (step  $x$ )

then show ?case (is  $(iterate \ 4 \cdot ?f \cdot x) \ 0 \sqsubseteq_{FD} RPHIL \ i$ )

```

apply (subst RPHIL-rec)
apply (simp add: write0-def numeral-4-eq-4 rphil-transitions-def rphil-state-update-def)
apply (rule mono-Mprefix-FD, auto simp: minus-suc[OF i-pos])+
using minus-suc[OF i-pos] by auto
qed

```

```

lemma LPHIL0-refines-LPHIL0norm: LPHIL0norm 0  $\sqsubseteq_{FD}$  LPHIL0
proof(unfold LPHIL0norm-def,
  induct rule:fix-ind-k-skip[where k=4 and f= $\Lambda$  x. ( $\lambda$ s. Mprefix (Trlp s) ( $\lambda$ e.
  x (Ulp s e)))]])
  show (1::nat)  $\leq$  4 by simp
next
  show adm ( $\lambda$ a. a 0  $\sqsubseteq_{FD}$  LPHIL0) by (simp add: cont-fun monofunI)
next
  case base-k-steps
  show ?case (is  $\forall j < 4. (iterate\ j.\ ?f.\ \perp) 0 \sqsubseteq_{FD} LPHIL0$ )
  proof –
    have less-2: $\bigwedge j. (j::nat) < 4 = (j = 0 \vee j = 1 \vee j = 2 \vee j = 3)$  by linarith
    moreover have (iterate 0. ?f.  $\perp$ ) 0  $\sqsubseteq_{FD}$  LPHIL0 by simp
    moreover have (iterate 1. ?f.  $\perp$ ) 0  $\sqsubseteq_{FD}$  LPHIL0
      by (subst LPHIL0-rec) (simp add: write0-def lphil0-transitions-def)
    moreover have (iterate 2. ?f.  $\perp$ ) 0  $\sqsubseteq_{FD}$  LPHIL0
      by (subst LPHIL0-rec) (simp add: numeral-2-eq-2 write0-def lphil0-transitions-def
        lphil0-state-update-def)
    moreover have (iterate 3. ?f.  $\perp$ ) 0  $\sqsubseteq_{FD}$  LPHIL0
      by (subst LPHIL0-rec) (simp add: numeral-3-eq-3 write0-def lphil0-transitions-def
        lphil0-state-update-def)
    ultimately show ?thesis by simp
  qed
next
  case (step x)
  then show ?case (is (iterate 4. ?f. x) 0  $\sqsubseteq_{FD}$  LPHIL0)
    by (subst LPHIL0-rec) (simp add: write0-def numeral-4-eq-4 lphil0-transitions-def
      lphil0-state-update-def)
qed

```

```

lemma RPHILnorm-refines-RPHIL:
  assumes i-pos: i > 0
  shows RPHIL i  $\sqsubseteq_{FD}$  RPHILnorm i 0
proof(unfold RPHIL-def, induct rule:fix-ind-k-skip[where k=1])
  case lower-bound
  then show ?case (is 1  $\leq$  1) by simp
next
  case admissibility
  then show ?case by (simp add: monofunI)

```

```

next
  case base-k-steps
  then show ?case by simp
next
case (step x) then show ?case
  apply (subst RPHILnorm-rec, simp add: write0-def rphil-transitions-def rphil-state-update-def)

  apply (rule mono-Mprefix-FD, simp)
  apply (subst RPHILnorm-rec, simp add: write0-def rphil-transitions-def rphil-state-update-def)
  apply (rule mono-Mprefix-FD, simp add: minus-suc[OF i-pos])
  apply (subst RPHILnorm-rec, simp add: write0-def rphil-transitions-def rphil-state-update-def)
  apply (rule mono-Mprefix-FD, simp add: minus-suc[OF i-pos])
  apply (subst RPHILnorm-rec, simp add: write0-def rphil-transitions-def rphil-state-update-def)
  apply (rule mono-Mprefix-FD, simp add: minus-suc[OF i-pos])
  using minus-suc[OF i-pos] by auto
qed

```

```

lemma LPHILOnorm-refines-LPHILO: LPHILO  $\sqsubseteq_{FD}$  LPHILOnorm 0
proof(unfold LPHILO-def,
  induct rule:fix-ind-k-skip[where k=1])
  show (1::nat)  $\leq$  1 by simp
next
  show adm ( $\lambda a. a \sqsubseteq_{FD} LPHILO_{norm} 0$ ) by (simp add: monofunI)
next
  case base-k-steps
  show ?case by simp
next
  case (step x)
  then show ?case (is iterate 1.?f.x  $\sqsubseteq_{FD} LPHILO_{norm} 0$ )
    apply (subst LPHILOnorm-rec, simp add: write0-def lphil0-transitions-def
lphil0-state-update-def)
    apply (rule mono-Mprefix-FD, simp)
    apply (subst LPHILOnorm-rec, simp add: write0-def lphil0-transitions-def
lphil0-state-update-def)
    apply (rule mono-Mprefix-FD, simp)
    apply (subst LPHILOnorm-rec, simp add: write0-def lphil0-transitions-def
lphil0-state-update-def)
    apply (rule mono-Mprefix-FD, simp)
    by (subst LPHILOnorm-rec, simp add: write0-def lphil0-transitions-def
lphil0-state-update-def)
qed

```

```

lemma RPHILnorm-is-RPHIL:  $i > 0 \implies RPHIL\ i = RPHIL_{norm}\ i\ 0$ 
  using RPHIL-refines-RPHILnorm RPHILnorm-refines-RPHIL FD-antisym by
blast

```

```

lemma LPHILOnorm-is-LPHILO: LPHILO = LPHILOnorm 0
  using LPHILO-refines-LPHILOnorm LPHILOnorm-refines-LPHILO FD-antisym

```

by *blast*

### 3.2.4 The normal form for the global philosopher network

**type-synonym**  $\sigma_{phils} = \text{nat list}$

**definition** *phils-transitions*::  $\text{nat} \Rightarrow \sigma_{phils} \Rightarrow \text{dining-event set } (Tr_P)$   
**where**  $Tr_P \ n \ ps = Tr_{lp} (ps!0) \cup (\bigcup_{i \in \{1..< n\}}. Tr_{rp} \ i \ (ps!i))$

**corollary** *phils-phil*:  $0 < n \implies e \in Tr_P \ n \ s \implies \text{phil } e < n$   
**by** (*auto simp add:phils-transitions-def lphil0-phil rphil-phil*)

**lemma** *phils-transitions-take*:  $0 < n \implies Tr_P \ n \ ps = Tr_P \ n \ (\text{take } n \ ps)$   
**by** (*auto simp add:phils-transitions-def*)

**definition**  $\sigma_{phils\text{-update}}$ ::  $\sigma_{phils} \Rightarrow \text{dining-event} \Rightarrow \sigma_{phils} (Up_P)$   
**where**  $Up_P \ ps \ e = (\text{let } i = (\text{phil } e) \text{ in if } i = 0 \text{ then } ps[i := (Up_{lp} (ps!i) \ e)]$   
else  $ps[i := (Up_{rp} \ i \ (ps!i) \ e)]$ )

**lemma** *phils-update-take*:  $\text{take } n \ (Up_P \ ps \ e) = Up_P \ (\text{take } n \ ps) \ e$   
**by** (*cases e*) (*simp-all add: \sigma\_{phils-update-def lphil0-state-update-def rphil-state-update-def take-update-swap*)

**definition**  $PHILs_{norm}$ ::  $\text{nat} \Rightarrow \sigma_{phils} \Rightarrow \text{dining-event process}$   
**where**  $PHILs_{norm} \ n = P_{norm} \llbracket Tr_P \ n, Up_P \rrbracket$

**lemma**  $PHILs_{norm}\text{-rec}$ :  $PHILs_{norm} \ n = (\lambda \ ps. \ \square \ e \in (Tr_P \ n \ ps) \rightarrow PHILs_{norm} \ n \ (Up_P \ ps \ e))$   
**using** *fix-eq[of  $\Lambda X. (\lambda \ ps. Mprefix (Tr_P \ n \ ps) (\lambda e. X (Up_P \ ps \ e)))$ ]  $PHILs_{norm}\text{-def}$*   
**by** *simp*

**lemma**  $PHILs_{norm}\text{-1-dir1}$ :  $\text{length } ps > 0 \implies PHILs_{norm} \ 1 \ ps \sqsubseteq_{FD} (LPHIL0_{norm} (ps!0))$

**proof**(*unfold  $PHILs_{norm}\text{-def}$ ,*  
*induct arbitrary:ps*  
*rule:fix-ind-k***where**  $k=1$   
**and**  $f = \Lambda x. (\lambda \ ps. Mprefix (Tr_P \ 1 \ ps) (\lambda e. x (Up_P \ ps \ e)))$ )  
**case** *admissibility*  
**then show** *?case* **by** (*simp add: cont-fun monofunI*)  
**next**  
**case** *base-k-steps*  
**then show** *?case* **by** *simp*  
**next**  
**case** (*step X*)  
**then show** *?case*  
**apply** (*subst LPHIL0<sub>norm</sub>-rec, simp add:\sigma\_{phils-update-def phils-transitions-def*)

**proof** (*intro mono-Mprefix-FD, safe, goal-cases*)  
**case** ( $1 \ e$ )

```

with 1(2) show ?case
  using 1(1)[rule-format, of ps[0 := Uplp (ps ! 0) e]]
  by (simp add:lphil0-transitions-def split:if-splits)
qed
qed

lemma PHILsnorm-1-dir2: length ps > 0 ⇒ (LPHIL0norm (ps!0)) ⊆FD PHILsnorm
1 ps
proof(unfold LPHIL0norm-def,
  induct arbitrary:ps rule:fix-ind-k[where k=1
  and f=Λ x. (λs. Mprefix (Trlp s) (λe. x (Uplp s
e))))))
  case admissibility
  then show ?case by (simp add: cont-fun monofunI)
next
  case base-k-steps
  then show ?case by simp
next
  case (step X)
  then show ?case
  apply (subst PHILsnorm-rec, simp add:σphils-update-def phils-transitions-def)
  proof (intro mono-Mprefix-FD, safe, goal-cases)
  case (1 e)
  with 1(2) show ?case
  using 1(1)[rule-format, of ps[0 := Uplp (ps ! 0) e]]
  by (simp add:lphil0-transitions-def split:if-splits)
  qed
qed

lemma PHILsnorm-1: length ps > 0 ⇒ PHILsnorm 1 ps = (LPHIL0norm (ps!0))
using PHILsnorm-1-dir1 PHILsnorm-1-dir2 FD-antisym by blast

lemma PHILsnorm-unfold:
assumes n-pos:0 < n
shows length ps = Suc n ⇒
  PHILsnorm (Suc n) ps = (PHILsnorm n (butlast
ps))|||(RPHILnorm n (ps!n)))
proof(rule FD-antisym)
  show length ps = Suc n ⇒ PHILsnorm (Suc n) ps ⊆FD (PHILsnorm n (butlast
ps))|||(RPHILnorm n (ps!n)))
  proof(subst PHILsnorm-def,
  induct arbitrary:ps
  rule:fix-ind-k[where k=1
  and f=Λ x. (λps. Mprefix (TrP (Suc n) ps) (λe. x (UpP
ps e))))))
  case admissibility
  then show ?case by (simp add: cont-fun monofunI)
next
  case base-k-steps

```

```

then show ?case by simp
next
case (step X)
have indep:  $\forall s_1 s_2. Tr_P n s_1 \cap Tr_{r_p} n s_2 = \{\}$ 
  using phils-phil rphil-phil n-pos by blast
from step have tra:  $(Tr_P (Suc n) ps) = (Tr_P n (butlast ps) \cup Tr_{r_p} n (ps ! n))$ 
  by (auto simp add: n-pos phils-transitions-def nth-butlast Suc-leI
    atLeastLessThanSuc Un-commute Un-assoc)
from step show ?case
  apply (auto simp add: indep dnorm-inter PHILsnorm-def RPHILnorm-def)
  apply (subst fix-eq, auto simp add: tra)
proof(rule mono-Mprefix-FD, safe, goal-cases)
  case (1 e)
  hence c:  $Up_P ps e ! n = ps ! n$ 
    using 1(3) phils-phil  $\sigma_{phils}$ -update-def step n-pos
    by (cases phil e, auto) (metis exists-least-iff nth-list-update-neq)
  have d:  $Up_P (butlast ps) e = butlast (Up_P ps e)$ 
    by (cases phil e, auto simp add:  $\sigma_{phils}$ -update-def butlast-list-update
      lphil0-state-update-def rphil-state-update-def)
  have e:  $length (Up_P ps e) = Suc n$ 
    by (metis (full-types) step(2) length-list-update  $\sigma_{phils}$ -update-def)
  from 1 show ?case
    using 1(1)[rule-format, of  $(Up_P ps e)$ ] c d e by auto
next
case (2 e)
have e:  $length (Up_P ps e) = Suc n$ 
  by (metis (full-types) step(2) length-list-update  $\sigma_{phils}$ -update-def)
from 2 show ?case
  using 2(1)[rule-format, of  $(Up_P ps e)$ , OF e] n-pos
  apply (auto simp add: butlast-list-update rphil-phil  $\sigma_{phils}$ -update-def)
  by (meson disjoint-iff-not-equal indep)
qed
qed
next
have indep:  $\forall s_1 s_2. Tr_P n s_1 \cap Tr_{r_p} n s_2 = \{\}$ 
  using phils-phil rphil-phil using n-pos by blast

show  $length ps = Suc n \implies (PHILs_{norm} n (butlast ps) || RPHIL_{norm} n (ps ! n))$ 
 $\sqsubseteq_{FD} PHILs_{norm} (Suc n) ps$ 
  apply (subst PHILsnorm-def, auto simp add: indep dnorm-inter RPHILnorm-def)
proof(rule fix-ind[where
  P= $\lambda a. \forall x. length x = Suc n \longrightarrow a (butlast x, x ! n)$   $\sqsubseteq_{FD} PHILs_{norm} (Suc$ 
n) x, rule-format],
  simp-all, goal-cases base step)
  case base
  then show ?case by (simp add: cont-fun monofunI)
next
case (step X ps)
hence tra:  $(Tr_P (Suc n) ps) = (Tr_P n (butlast ps) \cup Tr_{r_p} n (ps ! n))$ 

```

```

    by (auto simp add:n-pos phils-transitions-def nth-butlast
        Suc-leI atLeastLessThanSuc Un-commute Un-assoc)
  from step show ?case
    apply (auto simp add:indep dnorm-inter PHILsnorm-def RPHILnorm-def)
    apply (subst fix-eq, auto simp add:tra)
  proof(rule mono-Mprefix-FD, safe, goal-cases)
    case (1 e)
    hence c:UPP ps e ! n = ps ! n
      using 1(3) phils-phil σphils-update-def step n-pos
      by (cases phil e, auto) (metis exists-least-iff nth-list-update-neq)
    have d:UPP (butlast ps) e = butlast (UPP ps e)
      by (cases phil e, auto simp add:σphils-update-def butlast-list-update
          lphil0-state-update-def rphil-state-update-def)
    have e:length (UPP ps e) = Suc n
      by (metis (full-types) step(3) length-list-update σphils-update-def)
    from 1 show ?case
      using 1(2)[rule-format, of (UPP ps e), OF e] c d by auto
  next
    case (2 e)
    have e:length (UPP ps e) = Suc n
      by (metis (full-types) 2(3) length-list-update σphils-update-def)
    from 2 show ?case
      using 2(2)[rule-format, of (UPP ps e), OF e] n-pos
      apply(auto simp add: butlast-list-update rphil-phil σphils-update-def)
      by (meson disjoint-iff-not-equal indep)
  qed
qed
qed

```

lemma *pt*:  $0 < n \implies \text{PHILs}_{\text{norm}} n (\text{replicate } n 0) = \text{foldPHILs } n$

```

proof (induct n, simp)
  case (Suc n)
  then show ?case
    apply (auto simp add: PHILsnorm-unfold LPHIL0norm-is-LPHIL0)
    apply (metis Suc-le-eq butlast.simps(2) butlast-snoc RPHILnorm-is-RPHIL
        nat-neq-iff replicate-append-same replicate-empty)
    by (metis One-nat-def leI length-replicate less-Suc0 PHILsnorm-1 nth-Cons-0
        replicate-Suc)
  qed

```

corollary *PHILs-is-PHILs<sub>norm</sub>*:  $\text{PHILs}_{\text{norm}} N (\text{replicate } N 0) = \text{PHILs}$

```

using pt N-pos by simp

```

### 3.2.5 The complete process system under normal form

**definition** *dining-transitions*::  $\text{nat} \Rightarrow \sigma_{\text{phils}} \times \sigma_{\text{forks}} \Rightarrow \text{dining-event set } (\text{Tr}_D)$   
 where  $\text{Tr}_D n = (\lambda(ps,fs). (\text{Tr}_P n ps) \cap (\text{Tr}_F n fs))$

**definition** *dining-state-update*::



$\sigma_{phil} \times \sigma_{forks} \Rightarrow dining-event \Rightarrow \sigma_{phil} \times \sigma_{forks} (Up_D)$   
**where**  $Up_D = (\lambda(ps,fs) e. (Up_P ps e, Up_F fs e))$

**definition**  $DINING_{norm} :: nat \Rightarrow \sigma_{phil} \times \sigma_{forks} \Rightarrow dining-event\ process$   
**where**  $DINING_{norm} n = P_{norm} [Tr_D n, Up_D]$

**lemma**  $ltsDining-rec$ :  $DINING_{norm} n = (\lambda s. \square e \in (Tr_D n s) \rightarrow DINING_{norm} n (Up_D s e))$   
**using**  $fix-eq[of \Lambda X. (\lambda s. Mprefix (Tr_D n s) (\lambda e. X (Up_D s e)))]$   $DINING_{norm}-def$   
**by**  $simp$

**lemma**  $DINING-is-DINING_{norm}$ :  $DINING = DINING_{norm} N$  ( $replicate\ N\ 0$ ,  $replicate\ N\ 0$ )

**proof** –

**have**  $DINING_{norm} N$  ( $replicate\ N\ 0$ ,  $replicate\ N\ 0$ ) =  
 $(PHILs_{norm} N$  ( $replicate\ N\ 0$ )  $\parallel$   $FORKS_{norm} N$   
 $(replicate\ N\ 0))$   
**unfolding**  $DINING_{norm}-def$   $PHILs_{norm}-def$   $FORKS_{norm}-def$   $dining-transitions-def$

$dining-state-update-def$   $dnorm-par$  **by**  $simp$

**thus**  $?thesis$

**using**  $PHILs-is-PHILs_{norm}$   $FORKS-is-FORKS_{norm}$   $DINING-def$   
**by** ( $simp\ add: Par-commute$ )

**qed**

### 3.2.6 And finally: Philosophers may dine ! Always !

**corollary**  $lphil-states: Up_{lp} r e = 0 \vee Up_{lp} r e = 1 \vee Up_{lp} r e = 2 \vee Up_{lp} r e = 3$

**and**  $rphil-states: Up_{rp} i r e = 0 \vee Up_{rp} i r e = 1 \vee Up_{rp} i r e = 2 \vee Up_{rp} i r e = 3$

**unfolding**  $lphil0-state-update-def$   $rphil-state-update-def$  **by**  $auto$

**lemma**  $dining-events$ :

$e \in Tr_D N s \implies$

$(\exists i \in \{1..<N\}. e = picks\ i\ i \vee e = picks\ i\ (i-1) \vee e = putsdown\ i\ i \vee e = putsdown\ i\ (i-1))$

$\vee (e = picks\ 0\ 0 \vee e = picks\ 0\ (N-1) \vee e = putsdown\ 0\ 0 \vee e = putsdown\ 0\ (N-1))$

**by** ( $auto\ simp\ add: dining-transitions-def\ phil-transitions-def\ rphil-transitions-def$

$lphil0-transitions-def\ split:prod.splits\ if-splits$ )

**definition**  $inv-dining\ ps\ fs \equiv$

$(\forall i. Suc\ i < N \longrightarrow ((fs!(Suc\ i) = 1) \longleftrightarrow ps!Suc\ i \neq 0)) \wedge (fs!(N-1) = 2 \longleftrightarrow ps!0 \neq 0)$

$\wedge (\forall i < N - 1. fs!i = 2 \longleftrightarrow ps!Suc\ i = 2) \wedge (fs!0 = 1 \longleftrightarrow ps!0 = 2)$

$\wedge (\forall i < N. fs!i = 0 \vee fs!i = 1 \vee fs!i = 2)$

$$\begin{aligned} & \wedge (\forall i < N. ps!i = 0 \vee ps!i = 1 \vee ps!i = 2 \vee ps!i = 3) \\ & \wedge \text{length } fs = N \wedge \text{length } ps = N \end{aligned}$$

**lemma** *inv-DINING*:  $s \in \mathfrak{R} (Tr_D N) Up_D (\text{replicate } N 0, \text{replicate } N 0) \implies \text{inv-dining } (fst s) (snd s)$

**proof**(*induct rule:R.induct*)

**case** *rbase*

**show** *?case*

**by** (*simp add: inv-dining-def*)

**next**

**case** (*rstep s e*)

**from** *rstep(2,3)* **show** *?case*

**apply**(*auto simp add: dining-transitions-def phil-transitions-def forks-transitions-def*  
   *lphil0-transitions-def rphil-transitions-def fork-transitions-def*  
   *lphil0-state-update-def rphil-state-update-def sigma\_fork-update-def*  
   *dining-state-update-def sigma\_phil-update-def sigma\_forks-update-def*  
   *split:if-splits prod.split*)

**unfolding** *inv-dining-def*

**proof**(*goal-cases*)

**case** (*1 ps fs*)

**then show** *?case*

**by** (*simp add: nth-list-update*) *force*

**next**

**case** (*2 ps fs*)

**then show** *?case*

**by** (*auto simp add: nth-list-update*)

**next**

**case** (*3 ps fs*)

**then show** *?case*

**using** *N-pos-simps(3)* **by** *force*

**next**

**case** (*4 ps fs*)

**then show** *?case*

**by** (*simp add: nth-list-update*) *force*

**next**

**case** (*5 ps fs*)

**then show** *?case*

**using** *N-g1* **by** *linarith*

**next**

**case** (*6 ps fs*)

**then show** *?case*

**by** (*auto simp add: nth-list-update*)

**next**

**case** (*7 ps fs i*)

**then show** *?case*

**apply** (*simp add: nth-list-update, intro impI conjI, simp-all*)

**by** *auto[1]* (*metis N-pos Suc-pred less-antisym, metis zero-neq-numeral*)

**next**

**case** (*8 ps fs i*)

```

then show ?case
  apply (simp add:nth-list-update, intro impI conjI allI, simp-all)
  by (metis 8(1) zero-neq-one)+
next
  case (9 ps fs i)
  then show ?case
    apply (simp add:nth-list-update, intro impI conjI allI, simp-all)
    by (metis N-pos Suc-pred less-antisym) (metis n-not-Suc-n numeral-2-eq-2)
  next
    case (10 ps fs i)
    then show ?case
      apply (simp add:nth-list-update, intro impI conjI allI, simp-all)
      by (metis 10(1) 10(5) One-nat-def n-not-Suc-n numeral-2-eq-2)+
  qed
qed

lemma inv-implies-DF:inv-dining ps fs  $\implies$   $Tr_D N (ps, fs) \neq \{\}$ 
  unfolding inv-dining-def
  apply(simp add:dining-transitions-def phil-transitions-def forks-transitions-def
    lphil0-transitions-def
    split: if-splits prod.splits)
proof(elim conjE, intro conjI impI, goal-cases)
  case 1
  hence putsdown 0 (N - Suc 0)  $\in$  ( $\bigcup i < N. Tr_f i (fs ! i)$ )
    by (auto simp add:fork-transitions-def)
  then show ?case
    by blast
  next
  case 2
  hence putsdown 0 0  $\in$  ( $\bigcup i < N. Tr_f i (fs ! i)$ )
    by (auto simp add:fork-transitions-def)
  then show ?case
    by (simp add:fork-transitions-def) force
  next
  case 3
  hence a:fs!0 = 0  $\implies$  picks 0 0  $\in$  ( $\bigcup i < N. Tr_f i (fs ! i)$ )
    by (auto simp add:fork-transitions-def)
  from 3 have b1:ps!1 = 2  $\implies$  putsdown 1 0  $\in$  ( $\bigcup x \in \{Suc 0..<N\}. Tr_{r_p} x (ps ! x)$ )
    using N-g1 by (auto simp add:rphil-transitions-def)
  from 3 have b2:fs!0 = 2  $\implies$  putsdown 1 0  $\in$   $Tr_f 0 (fs ! 0)$ 
    using N-g1 by (auto simp add:fork-transitions-def) fastforce
  from 3 have c:fs!0  $\neq$  0  $\implies$  ps!1 = 2
    by (metis N-pos N-pos-simps(3) One-nat-def diff-is-0-eq neq0-conv)
  from 3 have d:fs!0  $\neq$  0  $\implies$  fs!0 = 2
    using N-pos by meson
  then show ?case
    apply(cases fs!0 = 0)
    using a apply (simp add: fork-transitions-def Un-insert-left)

```

```

    using b1[OF c] b2[OF d] N-pos by blast
next
case 4
then show ?case
  using 4(5)[rule-format, of 0, OF N-pos] apply(elim disjE)
proof(goal-cases)
  case 41:1
  then show ?case
  using 4(5)[rule-format, of 1, OF N-g1] apply(elim disjE)
proof(goal-cases)
  case 411:1
  from 411 have a0: ps!1 = 0
    by (metis N-g1 One-nat-def neg0-conv)
  from 411 have a1: picks 1 1 ∈ (⋃ i<N. Trf i (fs ! i))
    apply (auto simp add:fork-transitions-def)
    by (metis (mono-tags, lifting) N-g1 Int-Collect One-nat-def lessThan-iff)
  from 411 have a2: ps!1 = 0 ⇒ picks 1 1 ∈ (⋃ i∈{Suc 0..rp i (ps !
i))
    apply (auto simp add:rphil-transitions-def)
    using N-g1 by linarith
  from 411 show ?case
    using a0 a1 a2 by blast
next
case 412:2
hence ps!1 = 1 ∨ ps!1 = 3
  by (metis N-g1 One-nat-def less-numeral-extra(3) zero-less-diff)
with 412 show ?case
proof(elim disjE, goal-cases)
  case 4121:1
  from 4121 have b1: picks 1 0 ∈ (⋃ i<N. Trf i (fs ! i))
    apply (auto simp add:fork-transitions-def)
    by (metis (full-types) Int-Collect N-g1 N-pos One-nat-def lessThan-iff
mod-less)
  from 4121 have b2: picks 1 0 ∈ (⋃ i∈{Suc 0..rp i (ps ! i))
    apply (auto simp add:rphil-transitions-def)
    using N-g1 by linarith
  from 4121 show ?case
    using b1 b2 by blast
next
case 4122:2
from 4122 have b3: putsdown 1 1 ∈ (⋃ i<N. Trf i (fs ! i))
  apply (auto simp add:fork-transitions-def)
  using N-g1 by linarith
from 4122 have b4: putsdown 1 1 ∈ (⋃ i∈{Suc 0..rp i (ps ! i))
  apply (auto simp add:rphil-transitions-def)
  using N-g1 by linarith
then show ?case
  using b3 b4 by blast
qed

```

```

next
  case 413:3
  then show ?case
  proof(cases N = 2)
    case True
    with 413 show ?thesis by simp
  next
  case False
  from False 413 have c0: ps!2 = 2
    by (metis N-g1 Suc-1 Suc-diff-1 nat-neq-iff not-gr0 zero-less-diff)
  from False 413 have c1: putsdown 2 1 ∈ (⋃ i < N. Trf i (fs ! i))
    apply (auto simp add:fork-transitions-def)
    using N-g1 apply linarith
    using N-g1 by auto
  from False 413 have c2: ps!2 = 2 ⇒ putsdown 2 1 ∈ (⋃ i ∈ {Suc 0..<N}.
Trrp i (ps ! i))
    apply (auto simp add:rphil-transitions-def)
    using N-g1 by linarith
  from 413 False show ?thesis
    using c0 c1 c2 by blast
  qed
  qed
next
  case 42:2
  then show ?case by blast
next
  case 43:3
  from 43 have d0: ps!1 = 2
    by (metis One-nat-def gr0I)
  from 43 have d1: putsdown 1 0 ∈ (⋃ i < N. Trf i (fs ! i))
    by (auto simp add:fork-transitions-def)
  from 43 have d2: ps!1 = 2 ⇒ putsdown 1 0 ∈ (⋃ i ∈ {Suc 0..<N}. Trrp i (ps
! i))
    apply (auto simp add:rphil-transitions-def)
    using N-g1 by linarith
  from 43 show ?case
    using d0 d1 d2 by blast
  qed
next
  case 5
  then show ?case
    using 5(6)[rule-format, of 0] by simp
  qed
corollary DF-DINING: deadlock-free-v2 DINING
  unfolding DINING-is-DININGnorm DININGnorm-def
  using inv-DINING inv-implies-DF by (subst deadlock-free-dnorm) auto
end

```

**end**

## Chapter 4

# Conclusion

We presented a formalisation of the most comprehensive semantic model for CSP, a 'classical' language for the specification and analysis of concurrent systems studied in a rich body of literature. For this purpose, we ported [12] to a modern version of Isabelle, restructured the proofs, and extended the resulting theory of the language substantially. The result HOL-CSP 2 has been submitted to the Isabelle AFP [10], thus a fairly sustainable format accessible to other researchers and tools.

We developed a novel set of deadlock - and livelock inference proof principles based on classical and denotational characterizations. In particular, we formally investigated the relations between different refinement notions in the presence of deadlock - and livelock; an area where traditional CSP literature skates over the nitty-gritty details. Finally, we demonstrated how to exploit these results for deadlock/livelock analysis of protocols.

We put a large body of abstract CSP laws and induction principles together to form concrete verification technologies for generalized classical problems, which have been considered so far from the perspective of data-independence or structural parametricity. The underlying novel principle of "trading rich structure against rich state" allows one to convert processes into classical transition systems for which established invariant techniques become applicable.

Future applications of HOL-CSP 2 could comprise a combination with model checkers, where our theory with its derived rules can be used to certify the output of a model-checker over CSP. In our experience, labelled transition systems generated by model checkers may be used to steer inductions or to construct the normalized processes  $P_{norm}[\tau, \nu]$  automatically, thus combining efficient finite reasoning over finite sub-systems with globally infinite systems in a logically safe way.





# Bibliography

- [1] G. Barrett. Model checking in practice: the t9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, Feb 1995.
- [2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
- [3] 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.
- [4] A. J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer London.
- [5] A. Donovan and B. Kernighan. *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education, 2015.
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [7] Y. Isobe and M. Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.
- [8] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [9] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Berlin, Heidelberg, 1972. Springer.
- [10] S. Taha, L. Ye, and B. Wolff. HOL-CSP Version 2.0. *Archive of Formal Proofs*, Apr. 2019. <http://isa-afp.org/entries/HOL-CSP.html>.

- [11] S. Taha, L. Ye, and B. Wolff. Philosophers may Dine - Definitively! In C. A. Furia, editor, *Integrated Formal Methods (iFM)*, number 12546 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2020.
- [12] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.