

Hoare Logic for Parallel Programs

Leonor Prensa Nieto

December 12, 2021

Abstract

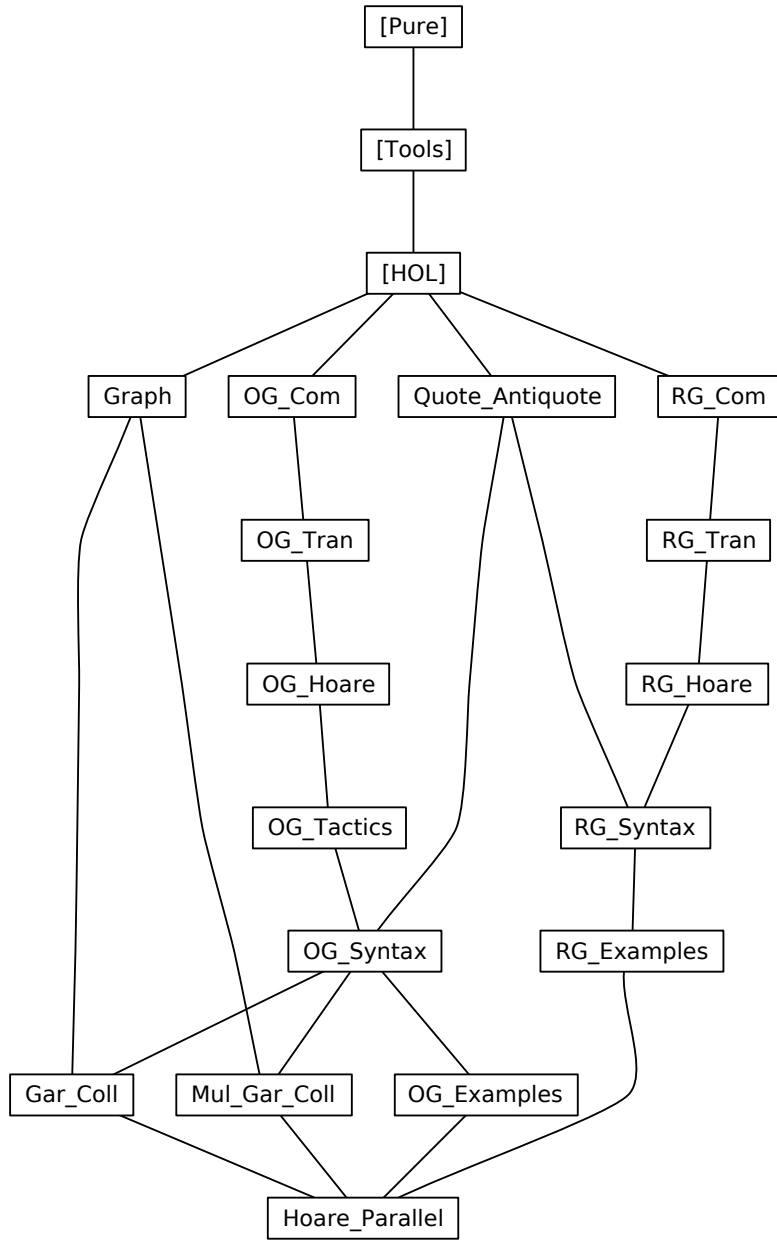
In the following theories a formalization of the Owicki-Gries and the rely-guarantee methods is presented. These methods are widely used for correctness proofs of parallel imperative programs with shared variables. We define syntax, semantics and proof rules in Isabelle/HOL. The proof rules also provide for programs parameterized in the number of parallel components. Their correctness w.r.t. the semantics is proven. Completeness proofs for both methods are extended to the new case of parameterized programs. (These proofs have not been formalized in Isabelle. They can be found in [1].) Using this formalizations we verify several non-trivial examples for parameterized and non-parameterized programs. For the automatic generation of verification conditions with the Owicki-Gries method we define a tactic based on the proof rules. The most involved examples are the verification of two garbage-collection algorithms, the second one parameterized in the number of mutators.

For excellent descriptions of this work see [2, 4, 1, 3].

Contents

1	The Owicki-Gries Method	4
1.1	Abstract Syntax	4
1.2	Operational Semantics	5
1.2.1	The Transition Relation	5
1.2.2	Definition of Semantics	6
1.3	Validity of Correctness Formulas	11
1.4	The Proof System	11
1.5	Soundness	12
1.5.1	Soundness of the System for Atomic Programs	13
1.5.2	Soundness of the System for Component Programs	14
1.5.3	Soundness of the System for Parallel Programs	16
1.6	Generation of Verification Conditions	20
1.7	Concrete Syntax	30
1.8	Examples	33
1.8.1	Mutual Exclusion	33
1.8.2	Parallel Zero Search	38
1.8.3	Producer/Consumer	40
1.8.4	Parameterized Examples	42
2	Case Study: Single and Multi-Mutator Garbage Collection Algorithms	45
2.1	Formalization of the Memory	45
2.1.1	Proofs about Graphs	46
2.2	The Single Mutator Case	53
2.2.1	The Mutator	54
2.2.2	The Collector	55
2.2.3	Interference Freedom	63
2.3	The Multi-Mutator Case	71
2.3.1	The Mutators	71
2.3.2	The Collector	74
2.3.3	Interference Freedom	81

3	The Rely-Guarantee Method	99
3.1	Abstract Syntax	99
3.2	Operational Semantics	99
3.2.1	Semantics of Component Programs	99
3.2.2	Semantics of Parallel Programs	100
3.2.3	Computations	101
3.2.4	Modular Definition of Computation	101
3.2.5	Equivalence of Both Definitions.	102
3.3	Validity of Correctness Formulas	107
3.3.1	Validity for Component Programs.	107
3.3.2	Validity for Parallel Programs.	107
3.3.3	Compositionality of the Semantics	108
3.3.4	The Semantics is Compositional	109
3.4	The Proof System	121
3.4.1	Proof System for Component Programs	121
3.4.2	Proof System for Parallel Programs	122
3.5	Soundness	123
3.5.1	Soundness of the System for Component Programs	127
3.5.2	Soundness of the System for Parallel Programs	144
3.6	Concrete Syntax	150
3.7	Examples	152
3.7.1	Set Elements of an Array to Zero	152
3.7.2	Increment a Variable in Parallel	153
3.7.3	Find Least Element	157



Chapter 1

The Owicki-Gries Method

1.1 Abstract Syntax

theory *OG-Com* **imports** *Main* **begin**

Type abbreviations for boolean expressions and assertions:

type-synonym *'a bexp* = *'a set*

type-synonym *'a assn* = *'a set*

The syntax of commands is defined by two mutually recursive datatypes: *'a ann-com* for annotated commands and *'a com* for non-annotated commands.

datatype *'a ann-com* =
 AnnBasic (*'a assn*) (*'a $\Rightarrow$ 'a*)
 | *AnnSeq* (*'a ann-com*) (*'a ann-com*)
 | *AnnCond1* (*'a assn*) (*'a bexp*) (*'a ann-com*) (*'a ann-com*)
 | *AnnCond2* (*'a assn*) (*'a bexp*) (*'a ann-com*)
 | *AnnWhile* (*'a assn*) (*'a bexp*) (*'a assn*) (*'a ann-com*)
 | *AnnAwait* (*'a assn*) (*'a bexp*) (*'a com*)
and *'a com* =
 Parallel (*'a ann-com option* $\times$ *'a assn*) *list*
 | *Basic* (*'a $\Rightarrow$ 'a*)
 | *Seq* (*'a com*) (*'a com*)
 | *Cond* (*'a bexp*) (*'a com*) (*'a com*)
 | *While* (*'a bexp*) (*'a assn*) (*'a com*)

The function *pre* extracts the precondition of an annotated command:

primrec *pre* :: *'a ann-com $\Rightarrow$ 'a assn* **where**
 pre (*AnnBasic* *r f*) = *r*
 | *pre* (*AnnSeq* *c1 c2*) = *pre c1*
 | *pre* (*AnnCond1* *r b c1 c2*) = *r*
 | *pre* (*AnnCond2* *r b c*) = *r*
 | *pre* (*AnnWhile* *r b i c*) = *r*
 | *pre* (*AnnAwait* *r b c*) = *r*

Well-formedness predicate for atomic programs:

```

primrec atom-com :: 'a com  $\Rightarrow$  bool where
  atom-com (Parallel Ts) = False
| atom-com (Basic f) = True
| atom-com (Seq c1 c2) = (atom-com c1  $\wedge$  atom-com c2)
| atom-com (Cond b c1 c2) = (atom-com c1  $\wedge$  atom-com c2)
| atom-com (While b i c) = atom-com c

end

```

1.2 Operational Semantics

theory OG-Tran **imports** OG-Com **begin**

```

type-synonym 'a ann-com-op = ('a ann-com) option
type-synonym 'a ann-triple-op = ('a ann-com-op  $\times$  'a assn)

```

```

primrec com :: 'a ann-triple-op  $\Rightarrow$  'a ann-com-op where
  com (c, q) = c

```

```

primrec post :: 'a ann-triple-op  $\Rightarrow$  'a assn where
  post (c, q) = q

```

```

definition All-None :: 'a ann-triple-op list  $\Rightarrow$  bool where
  All-None Ts  $\equiv \forall (c, q) \in \text{set } Ts. c = \text{None}$ 

```

1.2.1 The Transition Relation

inductive-set

```

  ann-transition :: (('a ann-com-op  $\times$  'a)  $\times$  ('a ann-com-op  $\times$  'a)) set
and transition :: (('a com  $\times$  'a)  $\times$  ('a com  $\times$  'a)) set
and ann-transition' :: ('a ann-com-op  $\times$  'a)  $\Rightarrow$  ('a ann-com-op  $\times$  'a)  $\Rightarrow$  bool
  (-  $\rightarrow$  -) [81,81] 100
and transition' :: ('a com  $\times$  'a)  $\Rightarrow$  ('a com  $\times$  'a)  $\Rightarrow$  bool
  (-  $\rightarrow$  -) [81,81] 100
and transitions :: ('a com  $\times$  'a)  $\Rightarrow$  ('a com  $\times$  'a)  $\Rightarrow$  bool
  (-  $\rightarrow$  -) [81,81] 100

```

where

```

  con-0  $\rightarrow$  con-1  $\equiv$  (con-0, con-1)  $\in$  ann-transition
| con-0  $\rightarrow$  P1  $\rightarrow$  con-1  $\equiv$  (con-0, con-1)  $\in$  transition
| con-0  $\rightarrow$  P*  $\rightarrow$  con-1  $\equiv$  (con-0, con-1)  $\in$  transition*

| AnnBasic: (Some (AnnBasic r f), s)  $\rightarrow$  (None, f s)

| AnnSeq1: (Some c0, s)  $\rightarrow$  (None, t)  $\implies$ 
  (Some (AnnSeq c0 c1), s)  $\rightarrow$  (Some c1, t)
| AnnSeq2: (Some c0, s)  $\rightarrow$  (Some c2, t)  $\implies$ 
  (Some (AnnSeq c0 c1), s)  $\rightarrow$  (Some (AnnSeq c2 c1), t)

| AnnCond1T: s  $\in$  b  $\implies$  (Some (AnnCond1 r b c1 c2), s)  $\rightarrow$  (Some c1, s)

```

| *AnnCond1F*: $s \notin b \implies (\text{Some } (\text{AnnCond1 } r \ b \ c1 \ c2), s) -1 \rightarrow (\text{Some } c2, s)$

| *AnnCond2T*: $s \in b \implies (\text{Some } (\text{AnnCond2 } r \ b \ c), s) -1 \rightarrow (\text{Some } c, s)$

| *AnnCond2F*: $s \notin b \implies (\text{Some } (\text{AnnCond2 } r \ b \ c), s) -1 \rightarrow (\text{None}, s)$

| *AnnWhileF*: $s \notin b \implies (\text{Some } (\text{AnnWhile } r \ b \ i \ c), s) -1 \rightarrow (\text{None}, s)$

| *AnnWhileT*: $s \in b \implies (\text{Some } (\text{AnnWhile } r \ b \ i \ c), s) -1 \rightarrow$
 $(\text{Some } (\text{AnnSeq } c \ (\text{AnnWhile } i \ b \ i \ c)), s)$

| *AnnAwait*: $\llbracket s \in b; \text{atom-com } c; (c, s) -P* \rightarrow (\text{Parallel } [], t) \rrbracket \implies$
 $(\text{Some } (\text{AnnAwait } r \ b \ c), s) -1 \rightarrow (\text{None}, t)$

| *Parallel*: $\llbracket i < \text{length } Ts; Ts!i = (\text{Some } c, q); (\text{Some } c, s) -1 \rightarrow (r, t) \rrbracket$
 $\implies (\text{Parallel } Ts, s) -P1 \rightarrow (\text{Parallel } (Ts \ [i := (r, q)]), t)$

| *Basic*: $(\text{Basic } f, s) -P1 \rightarrow (\text{Parallel } [], f \ s)$

| *Seq1*: $\text{All-None } Ts \implies (\text{Seq } (\text{Parallel } Ts) \ c, s) -P1 \rightarrow (c, s)$

| *Seq2*: $(c0, s) -P1 \rightarrow (c2, t) \implies (\text{Seq } c0 \ c1, s) -P1 \rightarrow (\text{Seq } c2 \ c1, t)$

| *CondT*: $s \in b \implies (\text{Cond } b \ c1 \ c2, s) -P1 \rightarrow (c1, s)$

| *CondF*: $s \notin b \implies (\text{Cond } b \ c1 \ c2, s) -P1 \rightarrow (c2, s)$

| *WhileF*: $s \notin b \implies (\text{While } b \ i \ c, s) -P1 \rightarrow (\text{Parallel } [], s)$

| *WhileT*: $s \in b \implies (\text{While } b \ i \ c, s) -P1 \rightarrow (\text{Seq } c \ (\text{While } b \ i \ c), s)$

monos *rtrancl-mono*

The corresponding abbreviations are:

abbreviation

$\text{ann-transition-}n :: ('a \ \text{ann-com-op} \times 'a) \Rightarrow \text{nat} \Rightarrow ('a \ \text{ann-com-op} \times 'a)$
 $\implies \text{bool } (- \dashrightarrow \text{--} [81,81] \ 100) \ \mathbf{where}$
 $\text{con-}0 \ -n \rightarrow \text{con-}1 \equiv (\text{con-}0, \text{con-}1) \in \text{ann-transition} \ \widetilde{\sim} \ n$

abbreviation

$\text{ann-transitions} :: ('a \ \text{ann-com-op} \times 'a) \Rightarrow ('a \ \text{ann-com-op} \times 'a) \Rightarrow \text{bool}$
 $(- \dashrightarrow \text{--} [81,81] \ 100) \ \mathbf{where}$
 $\text{con-}0 \ \dashrightarrow \text{con-}1 \equiv (\text{con-}0, \text{con-}1) \in \text{ann-transition}^*$

abbreviation

$\text{transition-}n :: ('a \ \text{com} \times 'a) \Rightarrow \text{nat} \Rightarrow ('a \ \text{com} \times 'a) \Rightarrow \text{bool}$
 $(- \dashrightarrow \text{--} [81,81,81] \ 100) \ \mathbf{where}$
 $\text{con-}0 \ -Pn \rightarrow \text{con-}1 \equiv (\text{con-}0, \text{con-}1) \in \text{transition} \ \widetilde{\sim} \ n$

1.2.2 Definition of Semantics

definition *ann-sem* :: $'a \ \text{ann-com} \Rightarrow 'a \Rightarrow 'a \ \text{set} \ \mathbf{where}$

$\text{ann-sem } c \equiv \lambda s. \{t. (\text{Some } c, s) \dashrightarrow (None, t)\}$

definition *ann-SEM* :: 'a ann-com $\Rightarrow$ 'a set $\Rightarrow$ 'a set **where**
ann-SEM c S $\equiv \bigcup (\text{ann-sem } c \text{ ' } S)$

definition *sem* :: 'a com $\Rightarrow$ 'a $\Rightarrow$ 'a set **where**
sem c $\equiv \lambda s. \{t. \exists Ts. (c, s) -P* \rightarrow (\text{Parallel } Ts, t) \wedge \text{All-None } Ts\}$

definition *SEM* :: 'a com $\Rightarrow$ 'a set $\Rightarrow$ 'a set **where**
SEM c S $\equiv \bigcup (\text{sem } c \text{ ' } S)$

abbreviation *Omega* :: 'a com $(\Omega \ 63)$
where $\Omega \equiv \text{While } \text{UNIV } \text{UNIV } (\text{Basic id})$

primrec *fwhile* :: 'a bexp $\Rightarrow$ 'a com $\Rightarrow$ nat $\Rightarrow$ 'a com **where**
fwhile b c 0 = Ω
| *fwhile* b c (Suc n) = $\text{Cond } b \ (\text{Seq } c \ (\text{fwhile } b \ c \ n)) \ (\text{Basic id})$

Proofs

declare *ann-transition-transition.intros* [intro]

inductive-cases *transition-cases*:

(*Parallel* T,s) $-P1 \rightarrow t$
(*Basic* f, s) $-P1 \rightarrow t$
(*Seq* c1 c2, s) $-P1 \rightarrow t$
(*Cond* b c1 c2, s) $-P1 \rightarrow t$
(*While* b i c, s) $-P1 \rightarrow t$

lemma *Parallel-empty-lemma* [rule-format (no-asm)]:

(*Parallel* [],s) $-Pn \rightarrow (\text{Parallel } Ts,t) \longrightarrow Ts=[] \wedge n=0 \wedge s=t$

apply(*induct* n)

apply(*simp* (no-asm))

apply *clarify*

apply(*drule* *relpow-Suc-D2*)

apply(*force* *elim:transition-cases*)

done

lemma *Parallel-AllNone-lemma* [rule-format (no-asm)]:

All-None Ss $\longrightarrow (\text{Parallel } Ss,s) -Pn \rightarrow (\text{Parallel } Ts,t) \longrightarrow Ts=Ss \wedge n=0 \wedge s=t$

apply(*induct* n)

apply(*simp* (no-asm))

apply *clarify*

apply(*drule* *relpow-Suc-D2*)

apply *clarify*

apply(*erule* *transition-cases,simp-all*)

apply(*force* *dest:nth-mem simp add:All-None-def*)

done

lemma *Parallel-AllNone*: *All-None* Ts $\Longrightarrow (\text{SEM } (\text{Parallel } Ts) \ X) = X$

apply (*unfold SEM-def sem-def*)

apply *auto*

```

apply(drule rtrancl-imp-UN-relpow)
apply clarify
apply(drule Parallel-AllNone-lemma)
apply auto
done

```

```

lemma Parallel-empty:  $Ts = [] \implies (SEM \ (Parallel \ Ts) \ X) = X$ 
apply(rule Parallel-AllNone)
apply(simp add:All-None-def)
done

```

Set of lemmas from Apt and Olderog "Verification of sequential and concurrent programs", page 63.

```

lemma L3-5i:  $X \subseteq Y \implies SEM \ c \ X \subseteq SEM \ c \ Y$ 
apply (unfold SEM-def)
apply force
done

```

```

lemma L3-5ii-lemma1:
   $\llbracket (c1, s1) -P* \rightarrow (Parallel \ Ts, s2); All-None \ Ts;$ 
   $(c2, s2) -P* \rightarrow (Parallel \ Ss, s3); All-None \ Ss \rrbracket$ 
   $\implies (Seq \ c1 \ c2, s1) -P* \rightarrow (Parallel \ Ss, s3)$ 
apply(erule converse-rtrancl-induct2)
apply(force intro:converse-rtrancl-into-rtrancl)+
done

```

```

lemma L3-5ii-lemma2 [rule-format (no-asm)]:
   $\forall c1 \ c2 \ s \ t. (Seq \ c1 \ c2, s) -Pn \rightarrow (Parallel \ Ts, t) \longrightarrow$ 
   $(All-None \ Ts) \longrightarrow (\exists y \ m \ Rs. (c1, s) -P* \rightarrow (Parallel \ Rs, y) \wedge$ 
   $(All-None \ Rs) \wedge (c2, y) -Pm \rightarrow (Parallel \ Ts, t) \wedge m \leq n)$ 
apply(induct n)
apply(force)
apply(safe dest!: relpow-Suc-D2)
apply(erule transition-cases,simp-all)
apply (fast intro!: le-SucI)
apply (fast intro!: le-SucI elim!: relpow-imp-rtrancl converse-rtrancl-into-rtrancl)
done

```

```

lemma L3-5ii-lemma3:
   $\llbracket (Seq \ c1 \ c2, s) -P* \rightarrow (Parallel \ Ts, t); All-None \ Ts \rrbracket \implies$ 
   $(\exists y \ Rs. (c1, s) -P* \rightarrow (Parallel \ Rs, y) \wedge All-None \ Rs$ 
   $\wedge (c2, y) -P* \rightarrow (Parallel \ Ts, t))$ 
apply(drule rtrancl-imp-UN-relpow)
apply(fast dest: L3-5ii-lemma2 relpow-imp-rtrancl)
done

```

```

lemma L3-5ii:  $SEM \ (Seq \ c1 \ c2) \ X = SEM \ c2 \ (SEM \ c1 \ X)$ 
apply (unfold SEM-def sem-def)
apply auto

```

```

  apply(fast dest: L3-5ii-lemma3)
  apply(fast elim: L3-5ii-lemma1)
done

```

```

lemma L3-5iii: SEM (Seq (Seq c1 c2) c3) X = SEM (Seq c1 (Seq c2 c3)) X
  apply (simp (no-asm) add: L3-5ii)
done

```

```

lemma L3-5iv:
  SEM (Cond b c1 c2) X = (SEM c1 (X ∩ b)) Un (SEM c2 (X ∩ (¬b)))
  apply (unfold SEM-def sem-def)
  apply auto
  apply (erule converse-rtranclE)
  prefer 2
  apply (erule transition-cases,simp-all)
  apply (fast intro: converse-rtrancl-into-rtrancl elim: transition-cases)+
done

```

```

lemma L3-5v-lemma1[rule-format]:
  (S,s) -Pn→ (T,t) → S=Ω → (¬(∃ Rs. T=(Parallel Rs) ∧ All-None Rs))
  apply (unfold UNIV-def)
  apply (rule nat-less-induct)
  apply safe
  apply (erule relpow-E2)
  apply simp-all
  apply (erule transition-cases)
  apply simp-all
  apply (erule relpow-E2)
  apply (simp add: Id-def)
  apply (erule transition-cases,simp-all)
  apply clarify
  apply (erule transition-cases,simp-all)
  apply (erule relpow-E2,simp)
  apply clarify
  apply (erule transition-cases)
  apply simp+
  apply clarify
  apply (erule transition-cases)
  apply simp-all
done

```

```

lemma L3-5v-lemma2: [(Ω, s) -P*→ (Parallel Ts, t); All-None Ts] ⇒ False
  apply (fast dest: rtrancl-imp-UN-relpow L3-5v-lemma1)
done

```

```

lemma L3-5v-lemma3: SEM (Ω) S = {}
  apply (unfold SEM-def sem-def)
  apply (fast dest: L3-5v-lemma2)

```

done

lemma *L3-5v-lemma4* [rule-format]:

$\forall s. (\text{While } b \text{ i } c, s) - P \rightarrow (\text{Parallel } Ts, t) \longrightarrow \text{All-None } Ts \longrightarrow$
 $(\exists k. (\text{fwhile } b \text{ c } k, s) - P \rightarrow (\text{Parallel } Ts, t))$

apply(rule nat-less-induct)

apply safe

apply(erule relpow-E2)

apply safe

apply(erule transition-cases,simp-all)

apply (rule-tac $x = 1$ in exI)

apply(force dest: Parallel-empty-lemma intro: converse-rtrancl-into-rtrancl simp
add: Id-def)

apply safe

apply(drule L3-5ii-lemma2)

apply safe

apply(drule le-imp-less-Suc)

apply (erule allE , erule impE,assumption)

apply (erule allE , erule impE, assumption)

apply safe

apply (rule-tac $x = k+1$ in exI)

apply(simp (no-asm))

apply(rule converse-rtrancl-into-rtrancl)

apply fast

apply(fast elim: L3-5ii-lemma1)

done

lemma *L3-5v-lemma5* [rule-format]:

$\forall s. (\text{fwhile } b \text{ c } k, s) - P \rightarrow (\text{Parallel } Ts, t) \longrightarrow \text{All-None } Ts \longrightarrow$
 $(\text{While } b \text{ i } c, s) - P \rightarrow (\text{Parallel } Ts, t)$

apply(induct k)

apply(force dest: L3-5v-lemma2)

apply safe

apply(erule converse-rtranclE)

apply simp-all

apply(erule transition-cases,simp-all)

apply(rule converse-rtrancl-into-rtrancl)

apply(fast)

apply(fast elim!: L3-5ii-lemma1 dest: L3-5ii-lemma3)

apply(drule rtrancl-imp-UN-relpow)

apply clarify

apply(erule relpow-E2)

apply simp-all

apply(erule transition-cases,simp-all)

apply(fast dest: Parallel-empty-lemma)

done

lemma *L3-5v*: $SEM (\text{While } b \text{ i } c) = (\lambda x. (\bigcup k. SEM (\text{fwhile } b \text{ c } k) x))$

apply(rule ext)

```

apply (simp add: SEM-def sem-def)
apply safe
apply(drule rtrancl-imp-UN-relpow,simp)
apply clarify
apply(fast dest:L3-5v-lemma4)
apply(fast intro: L3-5v-lemma5)
done

```

1.3 Validity of Correctness Formulas

definition *com-validity* :: 'a assn $\Rightarrow$ 'a com $\Rightarrow$ 'a assn $\Rightarrow$ bool ((β) $\models$ -// -//-) [90,55,90] 50) **where**
 $\models p \ c \ q \equiv SEM \ c \ p \subseteq q$

definition *ann-com-validity* :: 'a ann-com $\Rightarrow$ 'a assn $\Rightarrow$ bool ($\models$ - - [60,90] 45) **where**
 $\models c \ q \equiv ann-SEM \ c \ (pre \ c) \subseteq q$

end

1.4 The Proof System

theory *OG-Hoare* **imports** *OG-Tran* **begin**

primrec *assertions* :: 'a ann-com $\Rightarrow$ ('a assn) set **where**
 $assertions \ (AnnBasic \ r \ f) = \{r\}$
 $| \ assertions \ (AnnSeq \ c1 \ c2) = assertions \ c1 \cup assertions \ c2$
 $| \ assertions \ (AnnCond1 \ r \ b \ c1 \ c2) = \{r\} \cup assertions \ c1 \cup assertions \ c2$
 $| \ assertions \ (AnnCond2 \ r \ b \ c) = \{r\} \cup assertions \ c$
 $| \ assertions \ (AnnWhile \ r \ b \ i \ c) = \{r, i\} \cup assertions \ c$
 $| \ assertions \ (AnnAwait \ r \ b \ c) = \{r\}$

primrec *atomics* :: 'a ann-com $\Rightarrow$ ('a assn $\times$ 'a com) set **where**
 $atomics \ (AnnBasic \ r \ f) = \{(r, Basic \ f)\}$
 $| \ atomics \ (AnnSeq \ c1 \ c2) = atomics \ c1 \cup atomics \ c2$
 $| \ atomics \ (AnnCond1 \ r \ b \ c1 \ c2) = atomics \ c1 \cup atomics \ c2$
 $| \ atomics \ (AnnCond2 \ r \ b \ c) = atomics \ c$
 $| \ atomics \ (AnnWhile \ r \ b \ i \ c) = atomics \ c$
 $| \ atomics \ (AnnAwait \ r \ b \ c) = \{(r \cap b, c)\}$

primrec *com* :: 'a ann-triple-op $\Rightarrow$ 'a ann-com-op **where**
 $com \ (c, q) = c$

primrec *post* :: 'a ann-triple-op $\Rightarrow$ 'a assn **where**
 $post \ (c, q) = q$

definition *interfree-aux* :: ('a ann-com-op $\times$ 'a assn $\times$ 'a ann-com-op) $\Rightarrow$ bool **where**

$interfree\text{-}aux \equiv \lambda(co, q, co'). co' = None \vee$
 $(\forall (r, a) \in atoms (the\ co'). \models (q \cap r) \ a \ q \wedge$
 $(co = None \vee (\forall p \in assertions (the\ co). \models (p \cap r) \ a \ p)))$

definition $interfree :: (('a\ ann\text{-}triple\text{-}op)\ list) \Rightarrow bool$ **where**
 $interfree\ Ts \equiv \forall i\ j. i < length\ Ts \wedge j < length\ Ts \wedge i \neq j \longrightarrow$
 $interfree\text{-}aux\ (com\ (Ts!i), post\ (Ts!i), com\ (Ts!j))$

inductive

$oghoare :: 'a\ assn \Rightarrow 'a\ com \Rightarrow 'a\ assn \Rightarrow bool$ $((\exists \parallel - \text{ } // - \text{ } // -) [90, 55, 90] 50)$
and $ann\text{-}hoare :: 'a\ ann\text{-}com \Rightarrow 'a\ assn \Rightarrow bool$ $((\exists \vdash - \text{ } // -) [60, 90] 45)$

where

$AnnBasic: r \subseteq \{s. f\ s \in q\} \Longrightarrow \vdash (AnnBasic\ r\ f)\ q$

$| AnnSeq: \llbracket \vdash c0\ pre\ c1; \vdash c1\ q \rrbracket \Longrightarrow \vdash (AnnSeq\ c0\ c1)\ q$

$| AnnCond1: \llbracket r \cap b \subseteq pre\ c1; \vdash c1\ q; r \cap \neg b \subseteq pre\ c2; \vdash c2\ q \rrbracket$
 $\Longrightarrow \vdash (AnnCond1\ r\ b\ c1\ c2)\ q$

$| AnnCond2: \llbracket r \cap b \subseteq pre\ c; \vdash c\ q; r \cap \neg b \subseteq q \rrbracket \Longrightarrow \vdash (AnnCond2\ r\ b\ c)\ q$

$| AnnWhile: \llbracket r \subseteq i; i \cap b \subseteq pre\ c; \vdash c\ i; i \cap \neg b \subseteq q \rrbracket$
 $\Longrightarrow \vdash (AnnWhile\ r\ b\ i\ c)\ q$

$| AnnAwait: \llbracket atom\text{-}com\ c; \parallel - (r \cap b)\ c\ q \rrbracket \Longrightarrow \vdash (AnnAwait\ r\ b\ c)\ q$

$| AnnConseq: \llbracket \vdash c\ q; q \subseteq q' \rrbracket \Longrightarrow \vdash c\ q'$

$| Parallel: \llbracket \forall i < length\ Ts. \exists c\ q. Ts!i = (Some\ c, q) \wedge \vdash c\ q; interfree\ Ts \rrbracket$
 $\Longrightarrow \parallel - (\bigcap_{i \in \{i. i < length\ Ts\}} pre(the(com(Ts!i))))$
 $Parallel\ Ts$
 $(\bigcap_{i \in \{i. i < length\ Ts\}} post(Ts!i))$

$| Basic: \parallel - \{s. f\ s \in q\}\ (Basic\ f)\ q$

$| Seq: \llbracket \parallel - p\ c1\ r; \parallel - r\ c2\ q \rrbracket \Longrightarrow \parallel - p\ (Seq\ c1\ c2)\ q$

$| Cond: \llbracket \parallel - (p \cap b)\ c1\ q; \parallel - (p \cap \neg b)\ c2\ q \rrbracket \Longrightarrow \parallel - p\ (Cond\ b\ c1\ c2)\ q$

$| While: \llbracket \parallel - (p \cap b)\ c\ p \rrbracket \Longrightarrow \parallel - p\ (While\ b\ i\ c)\ (p \cap \neg b)$

$| Conseq: \llbracket p' \subseteq p; \parallel - p\ c\ q; q \subseteq q' \rrbracket \Longrightarrow \parallel - p'\ c\ q'$

1.5 Soundness

lemmas $[cong\ del] = if\text{-}weak\text{-}cong$

lemmas $ann\text{-}hoare\text{-}induct = oghoare\text{-}ann\text{-}hoare.induct\ [THEN\ conjunct2]$

lemmas $oghoare\text{-}induct = oghoare\text{-}ann\text{-}hoare.induct\ [THEN\ conjunct1]$

lemmas *AnnBasic* = *oghoare-ann-hoare.AnnBasic*
lemmas *AnnSeq* = *oghoare-ann-hoare.AnnSeq*
lemmas *AnnCond1* = *oghoare-ann-hoare.AnnCond1*
lemmas *AnnCond2* = *oghoare-ann-hoare.AnnCond2*
lemmas *AnnWhile* = *oghoare-ann-hoare.AnnWhile*
lemmas *AnnAwait* = *oghoare-ann-hoare.AnnAwait*
lemmas *AnnConseq* = *oghoare-ann-hoare.AnnConseq*

lemmas *Parallel* = *oghoare-ann-hoare.Parallel*
lemmas *Basic* = *oghoare-ann-hoare.Basic*
lemmas *Seq* = *oghoare-ann-hoare.Seq*
lemmas *Cond* = *oghoare-ann-hoare.Cond*
lemmas *While* = *oghoare-ann-hoare.While*
lemmas *Conseq* = *oghoare-ann-hoare.Conseq*

1.5.1 Soundness of the System for Atomic Programs

lemma *Basic-ntran* [*rule-format*]:
 $(Basic\ f,\ s) \dashv Pn \rightarrow (Parallel\ Ts,\ t) \longrightarrow All\ None\ Ts \longrightarrow t = f\ s$
apply(*induct n*)
apply(*simp (no-asm)*)
apply(*fast dest: relpow-Suc-D2 Parallel-empty-lemma elim: transition-cases*)
done

lemma *SEM-fwhile*: $SEM\ S\ (p \cap b) \subseteq p \implies SEM\ (fwhile\ b\ S\ k)\ p \subseteq (p \cap \neg b)$
apply (*induct k*)
apply(*simp (no-asm) add: L3-5v-lemma3*)
apply(*simp (no-asm) add: L3-5iv L3-5ii Parallel-empty*)
apply(*rule conjI*)
apply (*blast dest: L3-5i*)
apply(*simp add: SEM-def sem-def id-def*)
apply (*auto dest: Basic-ntran rtrancl-imp-UN-relpow*)
apply *blast*
done

lemma *atom-hoare-sound* [*rule-format*]:
 $\Vdash p\ c\ q \longrightarrow atom-com(c) \longrightarrow \Vdash p\ c\ q$
apply (*unfold com-validity-def*)
apply(*rule oghoare-induct*)
apply *simp-all*
— *Basic*
apply(*simp add: SEM-def sem-def*)
apply(*fast dest: rtrancl-imp-UN-relpow Basic-ntran*)
— *Seq*
apply(*rule impI*)
apply(*rule subset-trans*)
prefer 2 apply simp
apply(*simp add: L3-5ii L3-5i*)

— Cond
 apply(*simp add: L3-5iv*)
 — While
 apply (*force simp add: L3-5v dest: SEM-fwhile*)
 — Conseq
 apply(*force simp add: SEM-def sem-def*)
done

1.5.2 Soundness of the System for Component Programs

inductive-cases *ann-transition-cases*:

$(None, s) \rightarrow (c', s')$
 $(Some (AnnBasic\ r\ f), s) \rightarrow (c', s')$
 $(Some (AnnSeq\ c1\ c2), s) \rightarrow (c', s')$
 $(Some (AnnCond1\ r\ b\ c1\ c2), s) \rightarrow (c', s')$
 $(Some (AnnCond2\ r\ b\ c), s) \rightarrow (c', s')$
 $(Some (AnnWhile\ r\ b\ I\ c), s) \rightarrow (c', s')$
 $(Some (AnnAwait\ r\ b\ c), s) \rightarrow (c', s')$

Strong Soundness for Component Programs:

lemma *ann-hoare-case-analysis* [rule-format]: $\vdash C\ q' \longrightarrow$
 $((\forall r\ f.\ C = AnnBasic\ r\ f \longrightarrow (\exists q.\ r \subseteq \{s.\ f\ s \in q\} \wedge q \subseteq q')) \wedge$
 $(\forall c0\ c1.\ C = AnnSeq\ c0\ c1 \longrightarrow (\exists q.\ q \subseteq q' \wedge \vdash c0\ pre\ c1 \wedge \vdash c1\ q)) \wedge$
 $(\forall r\ b\ c1\ c2.\ C = AnnCond1\ r\ b\ c1\ c2 \longrightarrow (\exists q.\ q \subseteq q' \wedge$
 $r \cap b \subseteq pre\ c1 \wedge \vdash c1\ q \wedge r \cap \neg b \subseteq pre\ c2 \wedge \vdash c2\ q)) \wedge$
 $(\forall r\ b\ c.\ C = AnnCond2\ r\ b\ c \longrightarrow$
 $(\exists q.\ q \subseteq q' \wedge r \cap b \subseteq pre\ c \wedge \vdash c\ q \wedge r \cap \neg b \subseteq q)) \wedge$
 $(\forall r\ i\ b\ c.\ C = AnnWhile\ r\ b\ i\ c \longrightarrow$
 $(\exists q.\ q \subseteq q' \wedge r \subseteq i \wedge i \cap b \subseteq pre\ c \wedge \vdash c\ i \wedge i \cap \neg b \subseteq q)) \wedge$
 $(\forall r\ b\ c.\ C = AnnAwait\ r\ b\ c \longrightarrow (\exists q.\ q \subseteq q' \wedge \Vdash (r \cap b)\ c\ q)))$
apply(rule *ann-hoare-induct*)
apply *simp-all*
apply(rule-tac *x=q in exI, simp*)+
apply(rule *conjI, clarify, simp, clarify, rule-tac x=qa in exI, fast*)+
apply(*clarify, simp, clarify, rule-tac x=qa in exI, fast*)
done

lemma *Help*: $(transition \cap \{(x, y). \ True\}) = (transition)$
apply *force*
done

lemma *Strong-Soundness-aux-aux* [rule-format]:
 $(co, s) \rightarrow (co', t) \longrightarrow (\forall c.\ co = Some\ c \longrightarrow s \in pre\ c \longrightarrow$
 $(\forall q.\ \vdash c\ q \longrightarrow (if\ co' = None\ then\ t \in q\ else\ t \in pre(the\ co') \wedge \vdash (the\ co')\ q)))$
apply(rule *ann-transition-transition.induct [THEN conjunct1]*)
apply *simp-all*
 — Basic
 apply *clarify*
 apply(rule *ann-hoare-case-analysis*)

```

      apply force
— Seq
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply(fast intro: AnnConseq)
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply clarify
  apply(rule conjI)
  apply force
  apply(rule AnnSeq,simp)
  apply(fast intro: AnnConseq)
— Cond1
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply(fast intro: AnnConseq)
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply(fast intro: AnnConseq)
— Cond2
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply(fast intro: AnnConseq)
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply(fast intro: AnnConseq)
— While
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply force
  apply clarify
  apply(frule ann-hoare-case-analysis,simp)
  apply auto
  apply(rule AnnSeq)
  apply simp
  apply(rule AnnWhile)
  apply simp-all
— Await
  apply(frule ann-hoare-case-analysis,simp)
  apply clarify
  apply(drule atom-hoare-sound)
  apply simp
  apply(simp add: com-validity-def SEM-def sem-def)
  apply(simp add: Help All-None-def)
  apply force
done

```

lemma *Strong-Soundness-aux*: $\llbracket (\text{Some } c, s) \multimap (co, t); s \in \text{pre } c; \vdash c \ q \rrbracket$
 $\implies$ if $co = \text{None}$ then $t \in q$ else $t \in \text{pre } (the \ co) \wedge \vdash (the \ co) \ q$

```

apply(erule rtrancl-induct2)
apply simp
apply(case-tac a)
apply(fast elim: ann-transition-cases)
apply(erule Strong-Soundness-aux-aux)
apply simp
apply simp-all
done

```

```

lemma Strong-Soundness:  $\llbracket (\text{Some } c, s) \multimap (co, t); s \in \text{pre } c; \vdash c \ q \rrbracket$ 
 $\implies$  if  $co = \text{None}$  then  $t \in q$  else  $t \in \text{pre } (the \ co)$ 
apply(force dest:Strong-Soundness-aux)
done

```

```

lemma ann-hoare-sound:  $\vdash c \ q \implies \models c \ q$ 
apply (unfold ann-com-validity-def ann-SEM-def ann-sem-def)
apply clarify
apply(erule Strong-Soundness)
apply simp-all
done

```

1.5.3 Soundness of the System for Parallel Programs

```

lemma Parallel-length-post-P1:  $(\text{Parallel } Ts, s) \multimap P1 \rightarrow (R', t) \implies$ 
 $(\exists Rs. R' = (\text{Parallel } Rs) \wedge (\text{length } Rs) = (\text{length } Ts) \wedge$ 
 $(\forall i. i < \text{length } Ts \longrightarrow \text{post}(Rs \ ! \ i) = \text{post}(Ts \ ! \ i)))$ 
apply(erule transition-cases)
apply simp
apply clarify
apply(case-tac i=ia)
apply simp+
done

```

```

lemma Parallel-length-post-PStar:  $(\text{Parallel } Ts, s) \multimap P^* \rightarrow (R', t) \implies$ 
 $(\exists Rs. R' = (\text{Parallel } Rs) \wedge (\text{length } Rs) = (\text{length } Ts) \wedge$ 
 $(\forall i. i < \text{length } Ts \longrightarrow \text{post}(Ts \ ! \ i) = \text{post}(Rs \ ! \ i)))$ 
apply(erule rtrancl-induct2)
apply(simp-all)
apply clarify
apply simp
apply(erule Parallel-length-post-P1)
apply auto
done

```

```

lemma assertions-lemma:  $\text{pre } c \in \text{assertions } c$ 
apply(rule ann-com-com.induct [THEN conjunct1])
apply auto
done

```

```

lemma interfree-aux1 [rule-format]:
   $(c, s) -1 \rightarrow (r, t) \longrightarrow (\text{interfree-aux}(c1, q1, c) \longrightarrow \text{interfree-aux}(c1, q1, r))$ 
apply (rule ann-transition-transition.induct [THEN conjunct1])
apply (safe)
prefer 13
apply (rule TrueI)
apply (simp-all add: interfree-aux-def)
apply (force+)
done

```

```

lemma interfree-aux2 [rule-format]:
   $(c, s) -1 \rightarrow (r, t) \longrightarrow (\text{interfree-aux}(c, q, a) \longrightarrow \text{interfree-aux}(r, q, a))$ 
apply (rule ann-transition-transition.induct [THEN conjunct1])
apply (force simp add: interfree-aux-def) +
done

```

```

lemma interfree-lemma:  $\llbracket (\text{Some } c, s) -1 \rightarrow (r, t); \text{interfree } Ts ; i < \text{length } Ts; \\ Ts!i = (\text{Some } c, q) \rrbracket \Longrightarrow \text{interfree } (Ts[i := (r, q)])$ 
apply (simp add: interfree-def)
apply (clarify)
apply (case-tac i=j)
  apply (drule-tac t = ia in not-sym)
  apply (simp-all)
apply (force elim: interfree-aux1)
apply (force elim: interfree-aux2 simp add: nth-list-update)
done

```

Strong Soundness Theorem for Parallel Programs:

```

lemma Parallel-Strong-Soundness-Seq-aux:
   $\llbracket \text{interfree } Ts; i < \text{length } Ts; \text{com}(Ts!i) = \text{Some}(\text{AnnSeq } c0 \ c1) \rrbracket \\ \Longrightarrow \text{interfree } (Ts[i := (\text{Some } c0, \text{pre } c1)])$ 
apply (simp add: interfree-def)
apply (clarify)
apply (case-tac i=j)
  apply (force simp add: nth-list-update interfree-aux-def)
apply (case-tac i=ia)
  apply (erule-tac x=ia in allE)
  apply (force simp add: interfree-aux-def assertions-lemma)
apply (simp)
done

```

```

lemma Parallel-Strong-Soundness-Seq [rule-format (no-asm)]:
   $\llbracket \forall i < \text{length } Ts. (\text{if } \text{com}(Ts!i) = \text{None} \text{ then } b \in \text{post}(Ts!i) \\ \text{else } b \in \text{pre}(\text{the}(\text{com}(Ts!i))) \wedge \vdash \text{the}(\text{com}(Ts!i)) \text{ post}(Ts!i)); \\ \text{com}(Ts!i) = \text{Some}(\text{AnnSeq } c0 \ c1); i < \text{length } Ts; \text{interfree } Ts \rrbracket \Longrightarrow \\ (\forall ia < \text{length } Ts. (\text{if } \text{com}(Ts[i := (\text{Some } c0, \text{pre } c1)]! ia) = \text{None} \\ \text{then } b \in \text{post}(Ts[i := (\text{Some } c0, \text{pre } c1)]! ia) \\ \text{else } b \in \text{pre}(\text{the}(\text{com}(Ts[i := (\text{Some } c0, \text{pre } c1)]! ia))) \wedge \\ \vdash \text{the}(\text{com}(Ts[i := (\text{Some } c0, \text{pre } c1)]! ia)) \text{ post}(Ts[i := (\text{Some } c0, \text{pre } c1)]! ia)))$ 

```

```

     $\wedge \text{interfree } (Ts[i := (Some\ c0, \text{pre}\ c1)])$ 
  apply(rule conjI)
  apply safe
  apply(case-tac i=ia)
  apply simp
  apply(force dest: ann-hoare-case-analysis)
  apply simp
  apply(fast elim: Parallel-Strong-Soundness-Seq-aux)
done

lemma Parallel-Strong-Soundness-aux-aux [rule-format]:
  (Some c, b)  $\rightarrow$  (co, t)  $\rightarrow$ 
  ( $\forall Ts. i < \text{length } Ts \rightarrow \text{com}(Ts ! i) = \text{Some } c \rightarrow$ 
   ( $\forall i < \text{length } Ts. (\text{if } \text{com}(Ts ! i) = \text{None} \text{ then } b \in \text{post}(Ts ! i)$ 
    else  $b \in \text{pre}(\text{the}(\text{com}(Ts ! i))) \wedge \vdash \text{the}(\text{com}(Ts ! i)) \text{ post}(Ts ! i)) \rightarrow$ 
    $\text{interfree } Ts \rightarrow$ 
   ( $\forall j. j < \text{length } Ts \wedge i \neq j \rightarrow (\text{if } \text{com}(Ts ! j) = \text{None} \text{ then } t \in \text{post}(Ts ! j)$ 
    else  $t \in \text{pre}(\text{the}(\text{com}(Ts ! j))) \wedge \vdash \text{the}(\text{com}(Ts ! j)) \text{ post}(Ts ! j))$  )
  )
  apply(rule ann-transition-transition.induct [THEN conjunct1])
  apply safe
  prefer 11
  apply(rule TrueI)
  apply simp-all
  — Basic
    apply(erule-tac x = i in all-dupE, erule (1) notE impE)
    apply(erule-tac x = j in allE , erule (1) notE impE)
    apply(simp add: interfree-def)
    apply(erule-tac x = j in allE,simp)
    apply(erule-tac x = i in allE,simp)
    apply(drule-tac t = i in not-sym)
    apply(case-tac com(Ts ! j)=None)
    apply(force intro: converse-rtrancl-into-rtrancl
      simp add: interfree-aux-def com-validity-def SEM-def sem-def All-None-def)
    apply(simp add:interfree-aux-def)
    apply clarify
    apply simp
    apply(erule-tac x=pre y in ballE)
    apply(force intro: converse-rtrancl-into-rtrancl
      simp add: com-validity-def SEM-def sem-def All-None-def)
    apply(simp add:assertions-lemma)
  — Seqs
    apply(erule-tac x = Ts[i:=(Some c0, pre c1)] in allE)
    apply(drule Parallel-Strong-Soundness-Seq,simp+)
    apply(erule-tac x = Ts[i:=(Some c0, pre c1)] in allE)
    apply(drule Parallel-Strong-Soundness-Seq,simp+)
  — Await
    apply(rule-tac x = i in allE , assumption , erule (1) notE impE)
    apply(erule-tac x = j in allE , erule (1) notE impE)
    apply(simp add: interfree-def)

```

```

apply(erule-tac  $x = j$  in allE,simp)
apply(erule-tac  $x = i$  in allE,simp)
apply(drule-tac  $t = i$  in not-sym)
apply(case-tac com( $Ts ! j$ )=None)
  apply(force intro: converse-rtrancl-into-rtrancl simp add: interfree-aux-def
    com-validity-def SEM-def sem-def All-None-def Help)
apply(simp add:interfree-aux-def)
apply clarify
apply simp
apply(erule-tac  $x=pre\ y$  in ballE)
  apply(force intro: converse-rtrancl-into-rtrancl
    simp add: com-validity-def SEM-def sem-def All-None-def Help)
apply(simp add:assertions-lemma)
done

```

lemma *Parallel-Strong-Soundness-aux* [rule-format]:

$$\begin{aligned} & \llbracket (Ts', s) - P^* \rightarrow (Rs', t); \quad Ts' = (Parallel\ Ts); \quad interfree\ Ts; \\ & \forall i. i < length\ Ts \rightarrow (\exists c\ q. (Ts ! i) = (Some\ c, q) \wedge s \in (pre\ c) \wedge \vdash c\ q) \rrbracket \implies \\ & \forall Rs. Rs' = (Parallel\ Rs) \rightarrow (\forall j. j < length\ Rs \rightarrow \\ & \quad (if\ com(Rs ! j) = None\ then\ t \in post(Ts ! j) \\ & \quad \quad else\ t \in pre(the(com(Rs ! j))) \wedge \vdash the(com(Rs ! j))\ post(Ts ! j))) \wedge interfree\ Rs \end{aligned}$$

```

apply(erule rtrancl-induct2)
apply clarify
— Base
  apply force
— Induction step
  apply clarify
  apply(drule Parallel-length-post-PStar)
  apply clarify
  apply(ind-cases (Parallel Ts, s) - P1  $\rightarrow$  (Parallel Rs, t) for Ts s Rs t)
  apply(rule conjI)
  apply clarify
  apply(case-tac  $i=j$ )
  apply(simp split del:if-split)
  apply(erule Strong-Soundness-aux-aux,simp+)
  apply force
  apply force
  apply(simp split del: if-split)
  apply(erule Parallel-Strong-Soundness-aux-aux)
  apply(simp-all add: split del:if-split)
  apply force
  apply(rule interfree-lemma)
  apply simp-all
done

```

lemma *Parallel-Strong-Soundness*:

$$\begin{aligned} & \llbracket (Parallel\ Ts, s) - P^* \rightarrow (Parallel\ Rs, t); \quad interfree\ Ts; \quad j < length\ Rs; \\ & \forall i. i < length\ Ts \rightarrow (\exists c\ q. Ts ! i = (Some\ c, q) \wedge s \in pre\ c \wedge \vdash c\ q) \rrbracket \implies \\ & if\ com(Rs ! j) = None\ then\ t \in post(Ts ! j)\ else\ t \in pre\ (the(com(Rs ! j))) \end{aligned}$$

```

apply(drule Parallel-Strong-Soundness-aux)
apply simp+
done

lemma oghoare-sound [rule-format]:  $\|- p \ c \ q \longrightarrow \|= p \ c \ q$ 
apply (unfold com-validity-def)
apply(rule oghoare-induct)
apply(rule TrueI)+
— Parallel
  apply(simp add: SEM-def sem-def)
  apply(clarify, rename-tac x y i Ts')
  apply(frule Parallel-length-post-PStar)
  apply clarify
  apply(drule-tac j=i in Parallel-Strong-Soundness)
  apply clarify
  apply simp
  apply force
  apply simp
  apply(erule-tac V =  $\forall i. P \ i$  for  $P$  in thin-rl)
  apply(drule-tac s = length Rs in sym)
  apply(erule allE, erule impE, assumption)
  apply(force dest: nth-mem simp add: All-None-def)
— Basic
  apply(simp add: SEM-def sem-def)
  apply(force dest: rtrancl-imp-UN-relpow Basic-ntran)
— Seq
  apply(rule subset-trans)
  prefer 2 apply assumption
  apply(simp add: L3-5ii L3-5i)
— Cond
  apply(simp add: L3-5iv)
— While
  apply(simp add: L3-5v)
  apply (blast dest: SEM-fwhile)
— Conseq
apply(auto simp add: SEM-def sem-def)
done

end

```

1.6 Generation of Verification Conditions

```

theory OG-Tactics
imports OG-Hoare
begin

```

```

lemmas ann-hoare-intros=AnnBasic AnnSeq AnnCond1 AnnCond2 AnnWhile AnnAwait AnnConseq
lemmas oghoare-intros=Parallel Basic Seq Cond While Conseq

```

lemma *ParallelConseqRule*:

$$\begin{aligned} & \llbracket p \subseteq (\bigcap i \in \{i. i < \text{length } Ts\}. \text{pre}(\text{the}(\text{com}(Ts ! i))))); \\ & \quad \llbracket - (\bigcap i \in \{i. i < \text{length } Ts\}. \text{pre}(\text{the}(\text{com}(Ts ! i)))) \\ & \quad (\text{Parallel } Ts) \\ & \quad (\bigcap i \in \{i. i < \text{length } Ts\}. \text{post}(Ts ! i)); \\ & \quad (\bigcap i \in \{i. i < \text{length } Ts\}. \text{post}(Ts ! i)) \subseteq q \rrbracket \\ & \implies \llbracket - p (\text{Parallel } Ts) q \rrbracket \end{aligned}$$

apply (*rule Conseq*)
prefer 2
apply *fast*
apply *assumption* +
done

lemma *SkipRule*: $p \subseteq q \implies \llbracket - p (\text{Basic id}) q \rrbracket$

apply(*rule oghoare-intros*)
prefer 2 **apply**(*rule Basic*)
prefer 2 **apply**(*rule subset-refl*)
apply(*simp add:Id-def*)
done

lemma *BasicRule*: $p \subseteq \{s. (f s) \in q\} \implies \llbracket - p (\text{Basic } f) q \rrbracket$

apply(*rule oghoare-intros*)
prefer 2 **apply**(*rule oghoare-intros*)
prefer 2 **apply**(*rule subset-refl*)
apply *assumption*
done

lemma *SeqRule*: $\llbracket \llbracket - p \ c1 \ r; \llbracket - r \ c2 \ q \rrbracket \rrbracket \implies \llbracket - p (\text{Seq } c1 \ c2) q \rrbracket$

apply(*rule Seq*)
apply *fast* +
done

lemma *CondRule*:

$$\begin{aligned} & \llbracket p \subseteq \{s. (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}; \llbracket - w \ c1 \ q; \llbracket - w' \ c2 \ q \rrbracket \\ & \implies \llbracket - p (\text{Cond } b \ c1 \ c2) q \rrbracket \end{aligned}$$

apply(*rule Cond*)
apply(*rule Conseq*)
prefer 4 **apply**(*rule Conseq*)
apply *simp-all*
apply *force* +
done

lemma *WhileRule*: $\llbracket p \subseteq i; \llbracket - (i \cap b) \ c \ i; (i \cap (-b)) \subseteq q \rrbracket \implies \llbracket - p (\text{While } b \ i \ c) q \rrbracket$

apply(*rule Conseq*)
prefer 2 **apply**(*rule While*)
apply *assumption* +
done

Three new proof rules for special instances of the *AnnBasic* and the *AnnAwait* commands when the transformation performed on the state is the identity, and for an *AnnAwait* command where the boolean condition is $\{s. \text{True}\}$:

lemma *AnnatomRule*:

$\llbracket \text{atom-com}(c); \llbracket - \ r \ c \ q \rrbracket \implies \vdash (\text{AnnAwait } r \ \{s. \text{True}\} \ c) \ q$
apply(rule *AnnAwait*)
apply *simp-all*
done

lemma *AnnskipRule*:

$r \subseteq q \implies \vdash (\text{AnnBasic } r \ \text{id}) \ q$
apply(rule *AnnBasic*)
apply *simp*
done

lemma *AnnwaitRule*:

$\llbracket (r \cap b) \subseteq q \rrbracket \implies \vdash (\text{AnnAwait } r \ b \ (\text{Basic } \text{id})) \ q$
apply(rule *AnnAwait*)
apply *simp*
apply(rule *BasicRule*)
apply *simp*
done

Lemmata to avoid using the definition of *map-ann-hoare*, *interfree-aux*, *interfree-swap* and *interfree* by splitting it into different cases:

lemma *interfree-aux-rule1*: *interfree-aux*(*co*, *q*, *None*)
by(*simp add:interfree-aux-def*)

lemma *interfree-aux-rule2*:

$\forall (R, r) \in (\text{atomics } a). \llbracket - \ (q \cap R) \ r \ q \implies \text{interfree-aux}(\text{None}, q, \text{Some } a)$
apply(*simp add:interfree-aux-def*)
apply(*force elim:oghoare-sound*)
done

lemma *interfree-aux-rule3*:

$(\forall (R, r) \in (\text{atomics } a). \llbracket - \ (q \cap R) \ r \ q \wedge (\forall p \in (\text{assertions } c). \llbracket - \ (p \cap R) \ r \ p) \implies \text{interfree-aux}(\text{Some } c, q, \text{Some } a)$
apply(*simp add:interfree-aux-def*)
apply(*force elim:oghoare-sound*)
done

lemma *AnnBasic-assertions*:

$\llbracket \text{interfree-aux}(\text{None}, r, \text{Some } a); \text{interfree-aux}(\text{None}, q, \text{Some } a) \rrbracket \implies \text{interfree-aux}(\text{Some } (\text{AnnBasic } r \ f), q, \text{Some } a)$
apply(*simp add: interfree-aux-def*)
by *force*

lemma *AnnSeq-assertions*:
 $\llbracket \text{interfree-aux}(\text{Some } c1, q, \text{Some } a); \text{interfree-aux}(\text{Some } c2, q, \text{Some } a) \rrbracket \implies$
 $\text{interfree-aux}(\text{Some } (\text{AnnSeq } c1 \ c2), q, \text{Some } a)$
apply(simp add: interfree-aux-def)
by force

lemma *AnnCond1-assertions*:
 $\llbracket \text{interfree-aux}(\text{None}, r, \text{Some } a); \text{interfree-aux}(\text{Some } c1, q, \text{Some } a);$
 $\text{interfree-aux}(\text{Some } c2, q, \text{Some } a) \rrbracket \implies$
 $\text{interfree-aux}(\text{Some } (\text{AnnCond1 } r \ b \ c1 \ c2), q, \text{Some } a)$
apply(simp add: interfree-aux-def)
by force

lemma *AnnCond2-assertions*:
 $\llbracket \text{interfree-aux}(\text{None}, r, \text{Some } a); \text{interfree-aux}(\text{Some } c, q, \text{Some } a) \rrbracket \implies$
 $\text{interfree-aux}(\text{Some } (\text{AnnCond2 } r \ b \ c), q, \text{Some } a)$
apply(simp add: interfree-aux-def)
by force

lemma *AnnWhile-assertions*:
 $\llbracket \text{interfree-aux}(\text{None}, r, \text{Some } a); \text{interfree-aux}(\text{None}, i, \text{Some } a);$
 $\text{interfree-aux}(\text{Some } c, q, \text{Some } a) \rrbracket \implies$
 $\text{interfree-aux}(\text{Some } (\text{AnnWhile } r \ b \ i \ c), q, \text{Some } a)$
apply(simp add: interfree-aux-def)
by force

lemma *AnnAwait-assertions*:
 $\llbracket \text{interfree-aux}(\text{None}, r, \text{Some } a); \text{interfree-aux}(\text{None}, q, \text{Some } a) \rrbracket \implies$
 $\text{interfree-aux}(\text{Some } (\text{AnnAwait } r \ b \ c), q, \text{Some } a)$
apply(simp add: interfree-aux-def)
by force

lemma *AnnBasic-atomics*:
 $\llbracket - \ (q \cap r) \ (\text{Basic } f) \ q \implies \text{interfree-aux}(\text{None}, q, \text{Some } (\text{AnnBasic } r \ f)) \rrbracket$
by(simp add: interfree-aux-def oghoare-sound)

lemma *AnnSeq-atomics*:
 $\llbracket \text{interfree-aux}(\text{Any}, q, \text{Some } a1); \text{interfree-aux}(\text{Any}, q, \text{Some } a2) \rrbracket \implies$
 $\text{interfree-aux}(\text{Any}, q, \text{Some } (\text{AnnSeq } a1 \ a2))$
apply(simp add: interfree-aux-def)
by force

lemma *AnnCond1-atomics*:
 $\llbracket \text{interfree-aux}(\text{Any}, q, \text{Some } a1); \text{interfree-aux}(\text{Any}, q, \text{Some } a2) \rrbracket \implies$
 $\text{interfree-aux}(\text{Any}, q, \text{Some } (\text{AnnCond1 } r \ b \ a1 \ a2))$
apply(simp add: interfree-aux-def)
by force

lemma *AnnCond2-atomics*:

$interfree_aux \ (Any, q, Some \ a) \implies interfree_aux (Any, q, Some \ (AnnCond2 \ r \ b \ a))$

by(simp add: interfree-aux-def)

lemma *AnnWhile-atomics*: $interfree_aux \ (Any, q, Some \ a)$

$\implies interfree_aux (Any, q, Some \ (AnnWhile \ r \ b \ i \ a))$

by(simp add: interfree-aux-def)

lemma *Annatom-atomics*:

$\parallel - \ (q \cap r) \ a \ q \implies interfree_aux \ (None, q, Some \ (AnnAwait \ r \ \{x. \ True\} \ a))$

by(simp add: interfree-aux-def oghoare-sound)

lemma *AnnAwait-atomics*:

$\parallel - \ (q \cap (r \cap b)) \ a \ q \implies interfree_aux \ (None, q, Some \ (AnnAwait \ r \ b \ a))$

by(simp add: interfree-aux-def oghoare-sound)

definition *interfree-swap* :: $('a \ ann_triple_op * ('a \ ann_triple_op) \ list) \Rightarrow bool$ **where**

$interfree_swap == \lambda(x, xs). \forall y \in set \ xs. interfree_aux \ (com \ x, post \ x, com \ y)$

$\wedge interfree_aux (com \ y, post \ y, com \ x)$

lemma *interfree-swap-Empty*: $interfree_swap \ (x, [])$

by(simp add: interfree-swap-def)

lemma *interfree-swap-List*:

$\parallel interfree_aux \ (com \ x, post \ x, com \ y);$

$interfree_aux \ (com \ y, post \ y, com \ x); interfree_swap \ (x, xs) \parallel$

$\implies interfree_swap \ (x, y \# xs)$

by(simp add: interfree-swap-def)

lemma *interfree-swap-Map*: $\forall k. i \leq k \wedge k < j \longrightarrow interfree_aux \ (com \ x, post \ x, c \ k)$

$\wedge interfree_aux \ (c \ k, Q \ k, com \ x)$

$\implies interfree_swap \ (x, map \ (\lambda k. (c \ k, Q \ k)) \ [i..<j])$

by(force simp add: interfree-swap-def less-diff-conv)

lemma *interfree-Empty*: $interfree \ []$

by(simp add: interfree-def)

lemma *interfree-List*:

$\parallel interfree_swap(x, xs); interfree \ xs \parallel \implies interfree \ (x \# xs)$

apply(simp add: interfree-def interfree-swap-def)

apply clarify

apply(case-tac i)

apply(case-tac j)

apply simp-all

apply(case-tac j, simp+)

done

lemma *interfree-Map*:

$(\forall i \ j. a \leq i \wedge i < b \wedge a \leq j \wedge j < b \wedge i \neq j \longrightarrow interfree_aux \ (c \ i, Q \ i, c \ j))$

$\Rightarrow \text{interfree } (\text{map } (\lambda k. (c\ k, Q\ k))\ [a..<b])$
by(*force simp add: interfree-def less-diff-conv*)

definition *map-ann-hoare* :: $((\text{'a ann-com-op} * \text{'a assn})\ \text{list}) \Rightarrow \text{bool}$ ($[\vdash] - [0]$ 45)
where
 $[\vdash]\ Ts == (\forall i < \text{length}\ Ts. \exists c\ q. Ts!i = (\text{Some } c, q) \wedge \vdash\ c\ q)$

lemma *MapAnnEmpty*: $[\vdash]\ []$
by(*simp add: map-ann-hoare-def*)

lemma *MapAnnList*: $\llbracket \vdash\ c\ q ; [\vdash]\ xs \rrbracket \Rightarrow [\vdash]\ (\text{Some } c, q) \# xs$
apply(*simp add: map-ann-hoare-def*)
apply *clarify*
apply(*case-tac i, simp+*)
done

lemma *MapAnnMap*:
 $\forall k. i \leq k \wedge k < j \longrightarrow \vdash\ (c\ k)\ (Q\ k) \Rightarrow [\vdash]\ \text{map } (\lambda k. (\text{Some } (c\ k), Q\ k))\ [i..<j]$
apply(*simp add: map-ann-hoare-def less-diff-conv*)
done

lemma *ParallelRule*: $\llbracket [\vdash]\ Ts ; \text{interfree } Ts \rrbracket$
 $\Rightarrow \llbracket -\ (\bigcap i \in \{i. i < \text{length}\ Ts\}. \text{pre}(\text{the}(\text{com}(Ts!i))))$
 $\text{Parallel } Ts$
 $(\bigcap i \in \{i. i < \text{length}\ Ts\}. \text{post}(Ts!i))$
apply(*rule Parallel*)
apply(*simp add: map-ann-hoare-def*)
apply *simp*
done

The following are some useful lemmas and simplification tactics to control which theorems are used to simplify at each moment, so that the original input does not suffer any unexpected transformation.

lemma *Compl-Collect*: $\neg(\text{Collect } b) = \{x. \neg(b\ x)\}$
by *fast*

lemma *list-length*: $\text{length}\ [] = 0$ $\text{length}\ (x \# xs) = \text{Suc}(\text{length}\ xs)$
by *simp-all*

lemma *list-lemmas*: $\text{length}\ [] = 0$ $\text{length}\ (x \# xs) = \text{Suc}(\text{length}\ xs)$
 $(x \# xs) ! 0 = x$ $(x \# xs) ! \text{Suc } n = xs ! n$
by *simp-all*

lemma *le-Suc-eq-insert*: $\{i. i < \text{Suc } n\} = \text{insert } n\ \{i. i < n\}$
by *auto*

lemmas *primrecdef-list* = *pre.simps assertions.simps atomics.simps atom-com.simps*

lemmas *my-simp-list* = *list-lemmas fst-conv snd-conv*
not-less0 refl le-Suc-eq-insert Suc-not-Zero Zero-not-Suc nat.inject
Collect-mem-eq ball-simps option.simps primrecdef-list

lemmas *ParallelConseq-list* = *INTER-eq Collect-conj-eq length-map length-upt length-append*

```

ML <
fun before-interfree-simp-tac ctxt =
  simp-tac (put-simpset HOL-basic-ss ctxt addsimps [@{thm com.simps}, @{thm
post.simps}])

fun interfree-simp-tac ctxt =
  asm-simp-tac (put-simpset HOL-ss ctxt
  addsimps [@{thm split}, @{thm ball-Un}, @{thm ball-empty}] @ @{thms my-simp-list})

fun ParallelConseq ctxt =
  simp-tac (put-simpset HOL-basic-ss ctxt
  addsimps @ {thms ParallelConseq-list} @ @ {thms my-simp-list})
>

```

The following tactic applies *tac* to each conjunct in a subgoal of the form $A \implies a1 \wedge a2 \wedge \dots \wedge an$ returning n subgoals, one for each conjunct:

```

ML <
fun conjI-Tac ctxt tac i st = st |>
  ( (EVERY [resolve-tac ctxt [conjI] i,
    conjI-Tac ctxt tac (i+1),
    tac i]) ORELSE (tac i) )
>

```

Tactic for the generation of the verification conditions

The tactic basically uses two subtactics:

HoareRuleTac is called at the level of parallel programs, it uses the **ParallelTac** to solve parallel composition of programs. This verification has two parts, namely, (1) all component programs are correct and (2) they are interference free. *HoareRuleTac* is also called at the level of atomic regions, i.e. $\langle \rangle$ and *AWAIT* b *THEN* - *END*, and at each interference freedom test.

AnnHoareRuleTac is for component programs which are annotated programs and so, there are not unknown assertions (no need to use the parameter *precond*, see NOTE).

NOTE: *precond*(::bool) informs if the subgoal has the form $\parallel - ?p \ c \ q$, in this case we have *precond*=False and the generated verification condition would have the form $?p \subseteq \dots$ which can be solved by *rtac subset-refl*, if True we proceed to simplify it using the simplification tactics above.

```

ML <

fun WlpTac ctxt i = resolve-tac ctxt @ {thms SeqRule} i THEN HoareRuleTac ctxt
false (i + 1)

```

and HoareRuleTac ctxt precondition i st = st |>
 ((WlpTac ctxt i THEN HoareRuleTac ctxt precondition i)
 ORELSE
 (FIRST[resolve-tac ctxt @{thms SkipRule} i,
 resolve-tac ctxt @{thms BasicRule} i,
 EVERY[resolve-tac ctxt @{thms ParallelConseqRule} i,
 ParallelConseq ctxt (i+2),
 ParallelTac ctxt (i+1),
 ParallelConseq ctxt i],
 EVERY[resolve-tac ctxt @{thms CondRule} i,
 HoareRuleTac ctxt false (i+2),
 HoareRuleTac ctxt false (i+1)],
 EVERY[resolve-tac ctxt @{thms WhileRule} i,
 HoareRuleTac ctxt true (i+1)],
 K all-tac i]
 THEN (if precondition then (K all-tac i) else resolve-tac ctxt @{thms subset-refl}
 i)))

and AnnWlpTac ctxt i = resolve-tac ctxt @{thms AnnSeq} i THEN AnnHoareRule-
Tac ctxt (i + 1)

and AnnHoareRuleTac ctxt i st = st |>
 ((AnnWlpTac ctxt i THEN AnnHoareRuleTac ctxt i)
 ORELSE
 (FIRST[(resolve-tac ctxt @{thms AnnskipRule} i),
 EVERY[resolve-tac ctxt @{thms AnnatomRule} i,
 HoareRuleTac ctxt true (i+1)],
 (resolve-tac ctxt @{thms AnnwaitRule} i),
 resolve-tac ctxt @{thms AnnBasic} i,
 EVERY[resolve-tac ctxt @{thms AnnCond1} i,
 AnnHoareRuleTac ctxt (i+3),
 AnnHoareRuleTac ctxt (i+1)],
 EVERY[resolve-tac ctxt @{thms AnnCond2} i,
 AnnHoareRuleTac ctxt (i+1)],
 EVERY[resolve-tac ctxt @{thms AnnWhile} i,
 AnnHoareRuleTac ctxt (i+2)],
 EVERY[resolve-tac ctxt @{thms AnnAwait} i,
 HoareRuleTac ctxt true (i+1)],
 K all-tac i]))

and ParallelTac ctxt i = EVERY[resolve-tac ctxt @{thms ParallelRule} i,
interfree-Tac ctxt (i+1),
MapAnn-Tac ctxt i]

and MapAnn-Tac ctxt i st = st |>
 (FIRST[resolve-tac ctxt @{thms MapAnnEmpty} i,
 EVERY[resolve-tac ctxt @{thms MapAnnList} i,
 MapAnn-Tac ctxt (i+1),
 AnnHoareRuleTac ctxt i],
 EVERY[resolve-tac ctxt @{thms MapAnnMap} i,

$\text{resolve-tac } \text{ctxt} \ @\{\text{thms allI}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms impI}\} \ i,$
 $\text{AnnHoareRuleTac } \text{ctxt} \ i])$

$\text{and } \text{interfree-swap-Tac } \text{ctxt } i \ st = st \ |>$
 $(\text{FIRST}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-swap-Empty}\} \ i,$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-swap-List}\} \ i,$
 $\text{interfree-swap-Tac } \text{ctxt} \ (i+2),$
 $\text{interfree-aux-Tac } \text{ctxt} \ (i+1),$
 $\text{interfree-aux-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-swap-Map}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms allI}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms impI}\} \ i,$
 $\text{conjI-Tac } \text{ctxt} \ (\text{interfree-aux-Tac } \text{ctxt} \ i)])$

$\text{and } \text{interfree-Tac } \text{ctxt } i \ st = st \ |>$
 $(\text{FIRST}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-Empty}\} \ i,$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-List}\} \ i,$
 $\text{interfree-Tac } \text{ctxt} \ (i+1),$
 $\text{interfree-swap-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-Map}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms allI}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms allI}\} \ i,$
 $\text{resolve-tac } \text{ctxt} \ @\{\text{thms impI}\} \ i,$
 $\text{interfree-aux-Tac } \text{ctxt} \ i]])$

$\text{and } \text{interfree-aux-Tac } \text{ctxt } i = (\text{before-interfree-simp-tac } \text{ctxt} \ i) \ \text{THEN}$
 $(\text{FIRST}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms interfree-aux-rule1}\} \ i,$
 $\text{dest-assertions-Tac } \text{ctxt} \ i])$

$\text{and } \text{dest-assertions-Tac } \text{ctxt } i \ st = st \ |>$
 $(\text{FIRST}[\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnBasic-assertions}\} \ i,$
 $\text{dest-atomics-Tac } \text{ctxt} \ (i+1),$
 $\text{dest-atomics-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnSeq-assertions}\} \ i,$
 $\text{dest-assertions-Tac } \text{ctxt} \ (i+1),$
 $\text{dest-assertions-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnCond1-assertions}\} \ i,$
 $\text{dest-assertions-Tac } \text{ctxt} \ (i+2),$
 $\text{dest-assertions-Tac } \text{ctxt} \ (i+1),$
 $\text{dest-atomics-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnCond2-assertions}\} \ i,$
 $\text{dest-assertions-Tac } \text{ctxt} \ (i+1),$
 $\text{dest-atomics-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnWhile-assertions}\} \ i,$
 $\text{dest-assertions-Tac } \text{ctxt} \ (i+2),$
 $\text{dest-atomics-Tac } \text{ctxt} \ (i+1),$
 $\text{dest-atomics-Tac } \text{ctxt} \ i],$
 $\text{EVERY}[\text{resolve-tac } \text{ctxt} \ @\{\text{thms AnnAwait-assertions}\} \ i,$

```

      dest-atomics-Tac ctxt (i+1),
      dest-atomics-Tac ctxt i],
    dest-atomics-Tac ctxt i])
and dest-atomics-Tac ctxt i st = st |>
  (FIRST[EVERY[resolve-tac ctxt @{thms AnnBasic-atomics} i,
    HoareRuleTac ctxt true i],
    EVERY[resolve-tac ctxt @{thms AnnSeq-atomics} i,
      dest-atomics-Tac ctxt (i+1),
      dest-atomics-Tac ctxt i],
    EVERY[resolve-tac ctxt @{thms AnnCond1-atomics} i,
      dest-atomics-Tac ctxt (i+1),
      dest-atomics-Tac ctxt i],
    EVERY[resolve-tac ctxt @{thms AnnCond2-atomics} i,
      dest-atomics-Tac ctxt i],
    EVERY[resolve-tac ctxt @{thms AnnWhile-atomics} i,
      dest-atomics-Tac ctxt i],
    EVERY[resolve-tac ctxt @{thms Annatom-atomics} i,
      HoareRuleTac ctxt true i],
    EVERY[resolve-tac ctxt @{thms AnnAwait-atomics} i,
      HoareRuleTac ctxt true i],
    K all-tac i])
>

```

The final tactic is given the name *oghoare*:

```

ML <
fun oghoare-tac ctxt = SUBGOAL (fn (-, i) => HoareRuleTac ctxt true i)
>

```

Notice that the tactic for parallel programs *oghoare-tac* is initially invoked with the value *true* for the parameter *precond*.

Parts of the tactic can be also individually used to generate the verification conditions for annotated sequential programs and to generate verification conditions out of interference freedom tests:

```

ML <
fun annhoare-tac ctxt = SUBGOAL (fn (-, i) => AnnHoareRuleTac ctxt i)

fun interfree-aux-tac ctxt = SUBGOAL (fn (-, i) => interfree-aux-Tac ctxt i)
>

```

The so defined ML tactics are then “exported” to be used in Isabelle proofs.

```

method-setup oghoare = <
  Scan.succeed (SIMPLE-METHOD' o oghoare-tac)>
  verification condition generator for the oghoare logic

method-setup annhoare = <
  Scan.succeed (SIMPLE-METHOD' o annhoare-tac)>
  verification condition generator for the ann-hoare logic

```

```

method-setup interfree-aux = ⟨
  Scan.succeed (SIMPLE-METHOD' o interfree-aux-tac)
  verification condition generator for interference freedom tests

```

Tactics useful for dealing with the generated verification conditions:

```

method-setup conjI-tac = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (conjI-Tac ctxt (K all-tac)))
  verification condition generator for interference freedom tests

```

```

ML ⟨
  fun disjE-Tac ctxt tac i st = st |>
    ( (EVERY [eresolve-tac ctxt [disjE] i,
      disjE-Tac ctxt tac (i+1),
      tac i] ) ORELSE (tac i) )
  ⟩

```

```

method-setup disjE-tac = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (disjE-Tac ctxt (K all-tac)))
  verification condition generator for interference freedom tests

```

```

end

```

1.7 Concrete Syntax

```

theory Quote-Antiquote imports Main begin

```

```

syntax
  -quote      :: 'b ⇒ ('a ⇒ 'b)                ((«-») [0] 1000)
  -antiquote  :: ('a ⇒ 'b) ⇒ 'b                  ('- [1000] 1000)
  -Assert     :: 'a ⇒ 'a set                      (({ }-} [0] 1000)

```

```

translations
  {b} → CONST Collect «b»

```

```

parse-translation ⟨
  let
    fun quote-tr [t] = Syntax-Trans.quote-tr syntax-const ⟨-antiquote⟩ t
      | quote-tr ts = raise TERM (quote-tr, ts);
  in [(syntax-const ⟨-quote⟩, K quote-tr)] end
  ⟩

```

```

end
theory OG-Syntax
imports OG-Tactics Quote-Antiquote
begin

```

Syntax for commands and for assertions and boolean expressions in commands *com* and annotated commands *ann-com*.

abbreviation $Skip :: 'a\ com\ (SKIP\ 63)$
where $Skip \equiv Basic\ id$

abbreviation $AnnSkip :: 'a\ assn \Rightarrow 'a\ ann-com\ (-//SKIP\ [90]\ 63)$
where $r\ SKIP \equiv AnnBasic\ r\ id$

notation

$Seq\ (-, / - [55, 56]\ 55)$ **and**
 $AnnSeq\ (-; / - [60, 61]\ 60)$

syntax

$-Assign :: idt \Rightarrow 'b \Rightarrow 'a\ com\ ((\ ' - := / -) [70, 65]\ 61)$
 $-AnnAssign :: 'a\ assn \Rightarrow idt \Rightarrow 'b \Rightarrow 'a\ com\ ((\ ' - := / -) [90, 70, 65]\ 61)$

translations

$'x := a \rightarrow CONST\ Basic\ \langle\langle\ '(-update-name\ x\ (\lambda-. a)) \rangle\rangle$
 $r\ 'x := a \rightarrow CONST\ AnnBasic\ r\ \langle\langle\ '(-update-name\ x\ (\lambda-. a)) \rangle\rangle$

syntax

$-AnnCond1 :: 'a\ assn \Rightarrow 'a\ bexp \Rightarrow 'a\ ann-com \Rightarrow 'a\ ann-com \Rightarrow 'a\ ann-com$
 $(- // IF - / THEN - / ELSE - / FI [90, 0, 0, 0]\ 61)$
 $-AnnCond2 :: 'a\ assn \Rightarrow 'a\ bexp \Rightarrow 'a\ ann-com \Rightarrow 'a\ ann-com$
 $(- // IF - / THEN - / FI [90, 0, 0]\ 61)$
 $-AnnWhile :: 'a\ assn \Rightarrow 'a\ bexp \Rightarrow 'a\ assn \Rightarrow 'a\ ann-com \Rightarrow 'a\ ann-com$
 $(- // WHILE - / INV - // DO - // OD [90, 0, 0, 0]\ 61)$
 $-AnnAwait :: 'a\ assn \Rightarrow 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ ann-com$
 $(- // AWAIT - / THEN - / END [90, 0, 0]\ 61)$
 $-AnnAtom :: 'a\ assn \Rightarrow 'a\ com \Rightarrow 'a\ ann-com\ (-//\langle\ \rangle\ [90, 0]\ 61)$
 $-AnnWait :: 'a\ assn \Rightarrow 'a\ bexp \Rightarrow 'a\ ann-com\ (-//WAIT - END [90, 0]\ 61)$

 $-Cond :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ com \Rightarrow 'a\ com$
 $((0IF - / THEN - / ELSE - / FI) [0, 0, 0]\ 61)$
 $-Cond2 :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ com\ (IF - THEN - FI [0, 0]\ 56)$
 $-While-inv :: 'a\ bexp \Rightarrow 'a\ assn \Rightarrow 'a\ com \Rightarrow 'a\ com$
 $((0WHILE - / INV - // DO - / OD) [0, 0, 0]\ 61)$
 $-While :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ com$
 $((0WHILE - // DO - / OD) [0, 0]\ 61)$

translations

$IF\ b\ THEN\ c1\ ELSE\ c2\ FI \rightarrow CONST\ Cond\ \{\!\{b\}\!\}\ c1\ c2$
 $IF\ b\ THEN\ c\ FI \rightleftharpoons IF\ b\ THEN\ c\ ELSE\ SKIP\ FI$
 $WHILE\ b\ INV\ i\ DO\ c\ OD \rightarrow CONST\ While\ \{\!\{b\}\!\}\ i\ c$
 $WHILE\ b\ DO\ c\ OD \rightleftharpoons WHILE\ b\ INV\ CONST\ undefined\ DO\ c\ OD$

$r\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI \rightarrow CONST\ AnnCond1\ r\ \{\!\{b\}\!\}\ c1\ c2$
 $r\ IF\ b\ THEN\ c\ FI \rightarrow CONST\ AnnCond2\ r\ \{\!\{b\}\!\}\ c$
 $r\ WHILE\ b\ INV\ i\ DO\ c\ OD \rightarrow CONST\ AnnWhile\ r\ \{\!\{b\}\!\}\ i\ c$
 $r\ AWAIT\ b\ THEN\ c\ END \rightarrow CONST\ AnnAwait\ r\ \{\!\{b\}\!\}\ c$
 $r\ \langle c \rangle \rightleftharpoons r\ AWAIT\ CONST\ True\ THEN\ c\ END$

$r \text{ WAIT } b \text{ END} \Rightarrow r \text{ AWAIT } b \text{ THEN SKIP END}$

nonterminal *prgs*

syntax

$\text{-PAR} :: \text{prgs} \Rightarrow 'a \quad (\text{COBEGIN} // - // \text{COEND} [57] \ 56)$

$\text{-prg} :: ['a, 'a] \Rightarrow \text{prgs} \quad (- // - [60, 90] \ 57)$

$\text{-prgs} :: ['a, 'a, \text{prgs}] \Rightarrow \text{prgs} \quad (- // - // || // - [60, 90, 57] \ 57)$

$\text{-prg-scheme} :: ['a, 'a, 'a, 'a, 'a] \Rightarrow \text{prgs}$
 $(\text{SCHEME} [- \leq - < -] - // - [0, 0, 0, 60, 90] \ 57)$

translations

$\text{-prg } c \ q \Rightarrow [(\text{CONST } \text{Some } c, \ q)]$

$\text{-prgs } c \ q \ ps \Rightarrow (\text{CONST } \text{Some } c, \ q) \# \ ps$

$\text{-PAR } ps \Rightarrow \text{CONST Parallel } ps$

$\text{-prg-scheme } j \ i \ k \ c \ q \Rightarrow \text{CONST map } (\lambda i. (\text{CONST } \text{Some } c, \ q)) [j..<k]$

print-translation $\langle$

let

$\text{fun quote-tr}' f (t :: ts) =$

$\text{Term.list-comb } (f \ \$ \ \text{Syntax-Trans.quote-tr}' \ \textbf{syntax-const} \langle \text{-antiquote} \rangle \ t,$

$ts)$

$\mid \text{quote-tr}' - - = \text{raise Match};$

$\text{fun annquote-tr}' f (r :: t :: ts) =$

$\text{Term.list-comb } (f \ \$ \ r \ \$ \ \text{Syntax-Trans.quote-tr}' \ \textbf{syntax-const} \langle \text{-antiquote} \rangle$

$t, ts)$

$\mid \text{annquote-tr}' - - = \text{raise Match};$

$\text{val assert-tr}' = \text{quote-tr}' (\text{Syntax.const } \textbf{syntax-const} \langle \text{-Assert} \rangle);$

$\text{fun berp-tr}' \text{ name } ((\text{Const } (\textbf{const-syntax} \langle \text{Collect} \rangle, -) \ \$ \ t) :: ts) =$

$\text{quote-tr}' (\text{Syntax.const name}) (t :: ts)$

$\mid \text{berp-tr}' - - = \text{raise Match};$

$\text{fun annberp-tr}' \text{ name } (r :: (\text{Const } (\textbf{const-syntax} \langle \text{Collect} \rangle, -) \ \$ \ t) :: ts) =$

$\text{annquote-tr}' (\text{Syntax.const name}) (r :: t :: ts)$

$\mid \text{annberp-tr}' - - = \text{raise Match};$

$\text{fun assign-tr}' (\text{Abs } (x, -, f \ \$ \ k \ \$ \ \text{Bound } 0) :: ts) =$

$\text{quote-tr}' (\text{Syntax.const } \textbf{syntax-const} \langle \text{-Assign} \rangle \ \$ \ \text{Syntax-Trans.update-name-tr}'$

$f)$

$(\text{Abs } (x, \text{dummyT}, \text{Syntax-Trans.const-abs-tr}' k) :: ts)$

$\mid \text{assign-tr}' - - = \text{raise Match};$

$\text{fun annassign-tr}' (r :: \text{Abs } (x, -, f \ \$ \ k \ \$ \ \text{Bound } 0) :: ts) =$

$\text{quote-tr}' (\text{Syntax.const } \textbf{syntax-const} \langle \text{-AnnAssign} \rangle \ \$ \ r \ \$ \ \text{Syntax-Trans.update-name-tr}'$

```

f)
  (Abs (x, dummyT, Syntax-Trans.const-abs-tr' k) :: ts)
  | annassign-tr' - = raise Match;

fun Parallel-PAR [(Const (const-syntax ⟨Cons⟩, -) $
  (Const (const-syntax ⟨Pair⟩, -) $ (Const (const-syntax ⟨Some⟩, -) $ t1
) $ t2) $
  Const (const-syntax ⟨Nil⟩, -))] = Syntax.const syntax-const ⟨-prg⟩ $
t1 $ t2
  | Parallel-PAR [(Const (const-syntax ⟨Cons⟩, -) $
  (Const (const-syntax ⟨Pair⟩, -) $ (Const (const-syntax ⟨Some⟩, -) $
t1) $ t2) $ ts)] =
  Syntax.const syntax-const ⟨-prgs⟩ $ t1 $ t2 $ Parallel-PAR [ts]
  | Parallel-PAR - = raise Match;

fun Parallel-tr' ts = Syntax.const syntax-const ⟨-PAR⟩ $ Parallel-PAR ts;
in
  [(const-syntax ⟨Collect⟩, K assert-tr'),
  (const-syntax ⟨Basic⟩, K assign-tr'),
  (const-syntax ⟨Cond⟩, K (bexp-tr' syntax-const ⟨-Cond⟩)),
  (const-syntax ⟨While⟩, K (bexp-tr' syntax-const ⟨-While-inv⟩)),
  (const-syntax ⟨AnnBasic⟩, K annassign-tr'),
  (const-syntax ⟨AnnWhile⟩, K (annbexp-tr' syntax-const ⟨-AnnWhile⟩)),
  (const-syntax ⟨AnnAwait⟩, K (annbexp-tr' syntax-const ⟨-AnnAwait⟩)),
  (const-syntax ⟨AnnCond1⟩, K (annbexp-tr' syntax-const ⟨-AnnCond1⟩)),
  (const-syntax ⟨AnnCond2⟩, K (annbexp-tr' syntax-const ⟨-AnnCond2⟩))]
end

```

end

1.8 Examples

theory *OG-Examples* imports *OG-Syntax* begin

1.8.1 Mutual Exclusion

Peterson's Algorithm I

Eike Best. "Semantics of Sequential and Parallel Programs", page 217.

record *Petersons-mutex-1* =

```

pr1 :: nat
pr2 :: nat
in1 :: bool
in2 :: bool
hold :: nat

```

lemma *Petersons-mutex-1*:

$\| - \{ \text{'pr1=0} \wedge \neg \text{'in1} \wedge \text{'pr2=0} \wedge \neg \text{'in2} \}$

```

COBEGIN  $\{\text{'pr1}=0 \wedge \neg \text{'in1}\}$ 
WHILE True INV  $\{\text{'pr1}=0 \wedge \neg \text{'in1}\}$ 
DO
 $\{\text{'pr1}=0 \wedge \neg \text{'in1}\} \langle \text{'in1}:=\text{True},, \text{'pr1}:=1 \rangle;;$ 
 $\{\text{'pr1}=1 \wedge \text{'in1}\} \langle \text{'hold}:=1,, \text{'pr1}:=2 \rangle;;$ 
 $\{\text{'pr1}=2 \wedge \text{'in1} \wedge (\text{'hold}=1 \vee \text{'hold}=2 \wedge \text{'pr2}=2)\}$ 
AWAIT  $(\neg \text{'in2} \vee \neg(\text{'hold}=1))$  THEN  $\text{'pr1}:=3$  END;;
 $\{\text{'pr1}=3 \wedge \text{'in1} \wedge (\text{'hold}=1 \vee \text{'hold}=2 \wedge \text{'pr2}=2)\}$ 
 $\langle \text{'in1}:=\text{False},, \text{'pr1}:=0 \rangle$ 
OD  $\{\text{'pr1}=0 \wedge \neg \text{'in1}\}$ 
||
 $\{\text{'pr2}=0 \wedge \neg \text{'in2}\}$ 
WHILE True INV  $\{\text{'pr2}=0 \wedge \neg \text{'in2}\}$ 
DO
 $\{\text{'pr2}=0 \wedge \neg \text{'in2}\} \langle \text{'in2}:=\text{True},, \text{'pr2}:=1 \rangle;;$ 
 $\{\text{'pr2}=1 \wedge \text{'in2}\} \langle \text{'hold}:=2,, \text{'pr2}:=2 \rangle;;$ 
 $\{\text{'pr2}=2 \wedge \text{'in2} \wedge (\text{'hold}=2 \vee (\text{'hold}=1 \wedge \text{'pr1}=2))\}$ 
AWAIT  $(\neg \text{'in1} \vee \neg(\text{'hold}=2))$  THEN  $\text{'pr2}:=3$  END;;
 $\{\text{'pr2}=3 \wedge \text{'in2} \wedge (\text{'hold}=2 \vee (\text{'hold}=1 \wedge \text{'pr1}=2))\}$ 
 $\langle \text{'in2}:=\text{False},, \text{'pr2}:=0 \rangle$ 
OD  $\{\text{'pr2}=0 \wedge \neg \text{'in2}\}$ 
COEND
 $\{\text{'pr1}=0 \wedge \neg \text{'in1} \wedge \text{'pr2}=0 \wedge \neg \text{'in2}\}$ 
apply oghoare
— 104 verification conditions.
apply auto
done

```

Peterson's Algorithm II: A Busy Wait Solution

Apt and Olderog. "Verification of sequential and concurrent Programs", page 282.

record *Busy-wait-mutex* =

flag1 :: bool

flag2 :: bool

turn :: nat

after1 :: bool

after2 :: bool

lemma *Busy-wait-mutex*:

$\parallel - \{\text{True}\}$

$\text{'flag1}:=\text{False},, \text{'flag2}:=\text{False},,$

COBEGIN $\{\neg \text{'flag1}\}$

WHILE True

INV $\{\neg \text{'flag1}\}$

DO $\{\neg \text{'flag1}\} \langle \text{'flag1}:=\text{True},, \text{'after1}:=\text{False} \rangle;;$

$\{\text{'flag1} \wedge \neg \text{'after1}\} \langle \text{'turn}:=1,, \text{'after1}:=\text{True} \rangle;;$

$\{\text{'flag1} \wedge \text{'after1} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$

WHILE $\neg(\text{'flag2} \longrightarrow \text{'turn}=2)$

```

      INV  $\{\text{'flag1} \wedge \text{'after1} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$ 
      DO  $\{\text{'flag1} \wedge \text{'after1} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$  SKIP OD;;
       $\{\text{'flag1} \wedge \text{'after1} \wedge (\text{'flag2} \wedge \text{'after2} \longrightarrow \text{'turn}=2)\}$ 
       $\text{'flag1}:=\text{False}$ 
    OD
     $\{\text{False}\}$ 
  ||
   $\{\neg \text{'flag2}\}$ 
  WHILE True
  INV  $\{\neg \text{'flag2}\}$ 
  DO  $\{\neg \text{'flag2}\} \langle \text{'flag2}:=\text{True},, \text{'after2}:=\text{False} \rangle;;$ 
     $\{\text{'flag2} \wedge \neg \text{'after2}\} \langle \text{'turn}:=2,, \text{'after2}:=\text{True} \rangle;;$ 
     $\{\text{'flag2} \wedge \text{'after2} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$ 
    WHILE  $\neg(\text{'flag1} \longrightarrow \text{'turn}=1)$ 
    INV  $\{\text{'flag2} \wedge \text{'after2} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$ 
    DO  $\{\text{'flag2} \wedge \text{'after2} \wedge (\text{'turn}=1 \vee \text{'turn}=2)\}$  SKIP OD;;
     $\{\text{'flag2} \wedge \text{'after2} \wedge (\text{'flag1} \wedge \text{'after1} \longrightarrow \text{'turn}=1)\}$ 
     $\text{'flag2}:=\text{False}$ 
  OD
   $\{\text{False}\}$ 
COEND
 $\{\text{False}\}$ 
apply oghoare
— 122 vc
apply auto
done

```

Peterson's Algorithm III: A Solution using Semaphores

record *Semaphores-mutex* =

out :: bool

who :: nat

lemma *Semaphores-mutex*:

```

||-  $\{i \neq j\}$ 
 $\text{'out}:=\text{True} \text{ ,,}$ 
COBEGIN  $\{i \neq j\}$ 
  WHILE True INV  $\{i \neq j\}$ 
  DO  $\{i \neq j\}$  AWAIT  $\text{'out}$  THEN  $\text{'out}:=\text{False},, \text{'who}:=i$  END;;
     $\{\neg \text{'out} \wedge \text{'who}=i \wedge i \neq j\} \text{'out}:=\text{True}$  OD
   $\{\text{False}\}$ 
||
   $\{i \neq j\}$ 
  WHILE True INV  $\{i \neq j\}$ 
  DO  $\{i \neq j\}$  AWAIT  $\text{'out}$  THEN  $\text{'out}:=\text{False},, \text{'who}:=j$  END;;
     $\{\neg \text{'out} \wedge \text{'who}=j \wedge i \neq j\} \text{'out}:=\text{True}$  OD
   $\{\text{False}\}$ 
COEND
 $\{\text{False}\}$ 

```

apply *oghoare*
 — 38 vc
apply *auto*
done

Peterson's Algorithm III: Parameterized version:

lemma *Semaphores-parameterized-mutex*:
 $0 < n \implies \parallel - \{ \text{True} \}$
 $\text{'out} := \text{True} \text{ , ,}$
COBEGIN
 SCHEME $[0 \leq i < n]$
 $\{ \text{True} \}$
 WHILE *True* **INV** $\{ \text{True} \}$
 DO $\{ \text{True} \}$ **AWAIT** $\text{'out} := \text{False} \text{ , , 'who} := i \text{ END} ; ;$
 $\{ \neg \text{'out} \wedge \text{'who} = i \}$ $\text{'out} := \text{True} \text{ OD}$
 $\{ \text{False} \}$
 COEND
 $\{ \text{False} \}$
apply *oghoare*
 — 20 vc
apply *auto*
done

The Ticket Algorithm

record *Ticket-mutex* =
 num :: nat
 nextv :: nat
 turn :: nat list
 index :: nat

lemma *Ticket-mutex*:
 $\llbracket 0 < n ; I = \langle n = \text{length } \text{'turn} \wedge 0 < \text{'nextv} \wedge (\forall k \ l. k < n \wedge l < n \wedge k \neq l \implies \text{'turn}!k < \text{'num} \wedge (\text{'turn}!k = 0 \vee \text{'turn}!k \neq \text{'turn}!l)) \rangle \rrbracket$
 $\implies \parallel - \{ n = \text{length } \text{'turn} \}$
 $\text{'index} := 0 \text{ , ,}$
WHILE $\text{'index} < n$ **INV** $\{ n = \text{length } \text{'turn} \wedge (\forall i < \text{'index}. \text{'turn}!i = 0) \}$
 DO $\text{'turn} := \text{'turn}[\text{'index} := 0] \text{ , , 'index} := \text{'index} + 1 \text{ OD} \text{ , ,}$
 $\text{'num} := 1 \text{ , , 'nextv} := 1 \text{ , ,}$
COBEGIN
 SCHEME $[0 \leq i < n]$
 $\{ \text{'I} \}$
 WHILE *True* **INV** $\{ \text{'I} \}$
 DO $\{ \text{'I} \}$ $\langle \text{'turn} := \text{'turn}[i := \text{'num}] \text{ , , 'num} := \text{'num} + 1 \rangle ; ;$
 $\{ \text{'I} \}$ **WAIT** $\text{'turn}!i = \text{'nextv} \text{ END} ; ;$
 $\{ \text{'I} \wedge \text{'turn}!i = \text{'nextv} \}$ $\text{'nextv} := \text{'nextv} + 1$
 OD
 $\{ \text{False} \}$
COEND

```

  {False}
apply oghoare
— 35 vc
apply simp-all
— 16 vc
apply(tactic ‹ALLGOALS (clarify-tac context)›)
— 11 vc
apply simp-all
apply(tactic ‹ALLGOALS (clarify-tac context)›)
— 10 subgoals left
apply(erule less-SucE)
  apply simp
apply simp
— 9 subgoals left
apply(case-tac i=k)
  apply force
apply simp
apply(case-tac i=l)
  apply force
apply force
— 8 subgoals left
prefer 8
apply force
apply force
— 6 subgoals left
prefer 6
apply(erule-tac x=j in allE)
apply fastforce
— 5 subgoals left
prefer 5
apply(case-tac [!] j=k)
— 10 subgoals left
apply simp-all
apply(erule-tac x=k in allE)
apply force
— 9 subgoals left
apply(case-tac j=l)
  apply simp
  apply(erule-tac x=k in allE)
  apply(erule-tac x=k in allE)
  apply(erule-tac x=l in allE)
  apply force
apply(erule-tac x=k in allE)
apply(erule-tac x=k in allE)
apply(erule-tac x=l in allE)
apply force
— 8 subgoals left
apply force
apply(case-tac j=l)

```

```

  apply simp
  apply (erule-tac x=k in allE)
  apply (erule-tac x=l in allE)
  apply force
  apply force
  apply force
  — 5 subgoals left
  apply (erule-tac x=k in allE)
  apply (erule-tac x=l in allE)
  apply (case-tac j=l)
  apply force
  apply force
  apply force
  — 3 subgoals left
  apply (erule-tac x=k in allE)
  apply (erule-tac x=l in allE)
  apply (case-tac j=l)
  apply force
  apply force
  apply force
  — 1 subgoals left
  apply (erule-tac x=k in allE)
  apply (erule-tac x=l in allE)
  apply (case-tac j=l)
  apply force
  apply force
done

```

1.8.2 Parallel Zero Search

Synchronized Zero Search. Zero-6

Apt and Olderog. "Verification of sequential and concurrent Programs" page 294:

record *Zero-search* =

```

  turn :: nat
  found :: bool
  x :: nat
  y :: nat

```

lemma *Zero-search*:

$$\begin{aligned}
& \llbracket I1 = \llbracket a \leq 'x \wedge ('found \longrightarrow (a < 'x \wedge f('x) = 0) \vee ('y \leq a \wedge f('y) = 0)) \\
& \quad \wedge (\neg 'found \wedge a < 'x \longrightarrow f('x) \neq 0) \rrbracket ; \\
& \quad I2 = \llbracket 'y \leq a+1 \wedge ('found \longrightarrow (a < 'x \wedge f('x) = 0) \vee ('y \leq a \wedge f('y) = 0)) \\
& \quad \wedge (\neg 'found \wedge 'y \leq a \longrightarrow f('y) \neq 0) \rrbracket \rrbracket \Longrightarrow \\
& \llbracket - \{ \exists u. f(u) = 0 \} \\
& \quad 'turn := 1, 'found := False, \\
& \quad 'x := a, 'y := a+1, \\
& \quad COBEGIN \{ I1 \}
\end{aligned}$$

```

    WHILE  $\neg 'found$ 
    INV  $\{ 'I1 \}$ 
    DO  $\{ a \leq 'x \wedge ('found \longrightarrow 'y \leq a \wedge f('y)=0) \wedge (a < 'x \longrightarrow f('x) \neq 0) \}$ 
      WAIT  $'turn=1$  END;;
       $\{ a \leq 'x \wedge ('found \longrightarrow 'y \leq a \wedge f('y)=0) \wedge (a < 'x \longrightarrow f('x) \neq 0) \}$ 
       $'turn:=2$ ;;
       $\{ a \leq 'x \wedge ('found \longrightarrow 'y \leq a \wedge f('y)=0) \wedge (a < 'x \longrightarrow f('x) \neq 0) \}$ 
       $\langle 'x:='x+1, ,$ 
      IF  $f('x)=0$  THEN  $'found:=True$  ELSE SKIP FI
    OD;;
     $\{ 'I1 \wedge 'found \}$ 
     $'turn:=2$ 
     $\{ 'I1 \wedge 'found \}$ 
  ||
   $\{ 'I2 \}$ 
  WHILE  $\neg 'found$ 
  INV  $\{ 'I2 \}$ 
  DO  $\{ 'y \leq a+1 \wedge ('found \longrightarrow a < 'x \wedge f('x)=0) \wedge ('y \leq a \longrightarrow f('y) \neq 0) \}$ 
    WAIT  $'turn=2$  END;;
     $\{ 'y \leq a+1 \wedge ('found \longrightarrow a < 'x \wedge f('x)=0) \wedge ('y \leq a \longrightarrow f('y) \neq 0) \}$ 
     $'turn:=1$ ;;
     $\{ 'y \leq a+1 \wedge ('found \longrightarrow a < 'x \wedge f('x)=0) \wedge ('y \leq a \longrightarrow f('y) \neq 0) \}$ 
     $\langle 'y:=( 'y - 1), ,$ 
    IF  $f('y)=0$  THEN  $'found:=True$  ELSE SKIP FI
  OD;;
   $\{ 'I2 \wedge 'found \}$ 
   $'turn:=1$ 
   $\{ 'I2 \wedge 'found \}$ 
COEND
 $\{ f('x)=0 \vee f('y)=0 \}$ 
apply oghoare
— 98 verification conditions
apply auto
— auto takes about 3 minutes !!
done

```

Easier Version: without AWAIT. Apt and Olderog. page 256:

lemma *Zero-Search-2*:

```

 $\llbracket I1 = \langle a \leq 'x \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$ 
 $\wedge (\neg 'found \wedge a < 'x \longrightarrow f('x) \neq 0) \rangle \rrbracket$ ;
 $I2 = \langle 'y \leq a+1 \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$ 
 $\wedge (\neg 'found \wedge 'y \leq a \longrightarrow f('y) \neq 0) \rangle \rrbracket \implies$ 
 $\llbracket - \{ \exists u. f(u)=0 \}$ 
 $'found:=False, ,$ 
 $'x:=a, , 'y:=a+1, ,$ 
COBEGIN  $\{ 'I1 \}$ 
  WHILE  $\neg 'found$ 
  INV  $\{ 'I1 \}$ 
  DO  $\{ a \leq 'x \wedge ('found \longrightarrow 'y \leq a \wedge f('y)=0) \wedge (a < 'x \longrightarrow f('x) \neq 0) \}$ 

```

```

      < 'x:='x+1,,IF f('x)=0 THEN 'found:=True ELSE SKIP FI>
    OD
    { 'I1 ∧ 'found }
  ||
  { 'I2 }
  WHILE ¬'found
  INV { 'I2 }
  DO { 'y≤a+1 ∧ ('found → a<'x ∧ f('x)=0) ∧ ('y≤a → f('y)≠0) }
    < 'y:=('y - 1),,IF f('y)=0 THEN 'found:=True ELSE SKIP FI>
  OD
  { 'I2 ∧ 'found }
COEND
{ f('x)=0 ∨ f('y)=0 }
apply oghoare
— 20 vc
apply auto
— auto takes approx. 2 minutes.
done

```

1.8.3 Producer/Consumer

Previous lemmas

lemma *nat-lemma2*: $\llbracket b = m*(n::nat) + t; a = s*n + u; t=u; b-a < n \rrbracket \implies m \leq s$

proof —

```

  assume b = m*(n::nat) + t a = s*n + u t=u
  hence (m - s) * n = b - a by (simp add: diff-mult-distrib)
  also assume ... < n
  finally have m - s < 1 by simp
  thus ?thesis by arith

```

qed

lemma *mod-lemma*: $\llbracket (c::nat) \leq a; a < b; b - c < n \rrbracket \implies b \bmod n \neq a \bmod n$

apply(*subgoal-tac b=b div n*n + b mod n*)

prefer 2 **apply** (*simp add: div-mult-mod-eq [symmetric]*)

apply(*subgoal-tac a=a div n*n + a mod n*)

prefer 2

apply(*simp add: div-mult-mod-eq [symmetric]*)

apply(*subgoal-tac b - a ≤ b - c*)

prefer 2 **apply** *arith*

apply(*drule le-less-trans*)

back

apply *assumption*

apply(*frule less-not-refl2*)

apply(*drule less-imp-le*)

apply (*drule-tac m = a and k = n in div-le-mono*)

apply(*safe*)

apply(*frule-tac b = b and a = a and n = n in nat-lemma2, assumption, assumption*)

```

apply assumption
apply(drule order-antisym, assumption)
apply(rotate-tac -3)
apply(simp)
done

```

Producer/Consumer Algorithm

record *Producer-consumer* =

```

  ins :: nat
  outs :: nat
  li :: nat
  lj :: nat
  vx :: nat
  vy :: nat
  buffer :: nat list
  b :: nat list

```

The whole proof takes aprox. 4 minutes.

lemma *Producer-consumer*:

```

  [[INIT = « $0 < \text{length } a \wedge 0 < \text{length } 'buffer \wedge \text{length } 'b = \text{length } a$ » ;
    I = « $(\forall k < 'ins. 'outs \leq k \longrightarrow (a ! k) = 'buffer ! (k \bmod (\text{length } 'buffer))) \wedge$ 
      ' $outs \leq 'ins \wedge 'ins - 'outs \leq \text{length } 'buffer$ » ;
    I1 = « $'I \wedge 'li \leq \text{length } a$ » ;
    p1 = « $'I1 \wedge 'li = 'ins$ » ;
    I2 = « $'I \wedge (\forall k < 'lj. (a ! k) = ('b ! k)) \wedge 'lj \leq \text{length } a$ » ;
    p2 = « $'I2 \wedge 'lj = 'outs$ » ]]  $\implies$ 
  || - { 'INIT }
  'ins := 0,, 'outs := 0,, 'li := 0,, 'lj := 0,,
  COBEGIN { 'p1  $\wedge$  'INIT }
    WHILE 'li < length a
      INV { 'p1  $\wedge$  'INIT }
      DO { 'p1  $\wedge$  'INIT  $\wedge$  'li < length a }
        'vx := (a ! 'li);;
        { 'p1  $\wedge$  'INIT  $\wedge$  'li < length a  $\wedge$  'vx = (a ! 'li) }
        WAIT 'ins - 'outs < length 'buffer END;;
        { 'p1  $\wedge$  'INIT  $\wedge$  'li < length a  $\wedge$  'vx = (a ! 'li)
           $\wedge$  'ins - 'outs < length 'buffer }
        'buffer := (list-update 'buffer ('ins mod (length 'buffer)) 'vx);;
        { 'p1  $\wedge$  'INIT  $\wedge$  'li < length a
           $\wedge$  (a ! 'li) = ('buffer ! ('ins mod (length 'buffer)))
           $\wedge$  'ins - 'outs < length 'buffer }
        'ins := 'ins + 1;;
        { 'I1  $\wedge$  'INIT  $\wedge$  ('li + 1) = 'ins  $\wedge$  'li < length a }
        'li := 'li + 1
      OD
    { 'p1  $\wedge$  'INIT  $\wedge$  'li = length a }
  ||
  { 'p2  $\wedge$  'INIT }

```

```

    WHILE 'lj < length a
      INV { 'p2 ∧ 'INIT }
    DO { 'p2 ∧ 'lj < length a ∧ 'INIT }
      WAIT 'outs < 'ins END;;
      { 'p2 ∧ 'lj < length a ∧ 'outs < 'ins ∧ 'INIT }
      'vy := ('buffer ! ('outs mod (length 'buffer)));;
      { 'p2 ∧ 'lj < length a ∧ 'outs < 'ins ∧ 'vy = (a ! 'lj) ∧ 'INIT }
      'outs := 'outs + 1;;
      { 'I2 ∧ ('lj + 1) = 'outs ∧ 'lj < length a ∧ 'vy = (a ! 'lj) ∧ 'INIT }
      'b := (list-update 'b 'lj 'vy);;
      { 'I2 ∧ ('lj + 1) = 'outs ∧ 'lj < length a ∧ (a ! 'lj) = ('b ! 'lj) ∧ 'INIT }
      'lj := 'lj + 1
    OD
    { 'p2 ∧ 'lj = length a ∧ 'INIT }
  COEND
  { ∀ k < length a. (a ! k) = ('b ! k) }
apply oghoare
— 138 vc
apply(tactic (ALLGOALS (clarify-tac context)))
— 112 subgoals left
apply(simp-all (no-asm))
— 43 subgoals left
apply(tactic (ALLGOALS (conjI-Tac context (K all-tac))))
— 419 subgoals left
apply(tactic (ALLGOALS (clarify-tac context)))
— 99 subgoals left
apply(simp-all only:length-0-conv [ THEN sym ])
— 20 subgoals left
apply (simp-all del:length-0-conv length-greater-0-conv add: nth-list-update mod-lemma)
— 9 subgoals left
apply (force simp add:less-Suc-eq)
apply (hypsubst-thin, drule sym)
apply (force simp add:less-Suc-eq) +
done

```

1.8.4 Parameterized Examples

Set Elements of an Array to Zero

record *Example1* =

a :: nat ⇒ nat

lemma *Example1*:

```

  ||- { True }
  COBEGIN SCHEME [0 ≤ i < n] { True } 'a := 'a (i := 0) { 'a i = 0 } COEND
  { ∀ i < n. 'a i = 0 }
apply oghoare
apply simp-all
done

```

Same example with lists as auxiliary variables.

```

record Example1-list =
  A :: nat list
lemma Example1-list:
  ||- {n < length 'A}
  COBEGIN
    SCHEME [0 ≤ i < n] {n < length 'A} 'A := 'A[i:=0] { 'A!i=0 }
  COEND
  {∀ i < n. 'A!i = 0}
apply oghoare
apply force+
done

```

Increment a Variable in Parallel

First some lemmas about summation properties.

```

lemma Example2-lemma2-aux: !!b. j < n ==>
  (∑ i=0.. $n$ . (b i::nat)) =
  (∑ i=0.. $j$ . b i) + b j + (∑ i=0.. $n-(\text{Suc } j)$ . b (Suc j + i))
apply(induct n)
apply simp-all
apply(simp add:less-Suc-eq)
apply(auto)
apply(subgoal-tac n - j = Suc(n - Suc j))
apply simp
apply arith
done

```

```

lemma Example2-lemma2-aux2:
  !!b. j ≤ s ==> (∑ i::nat=0.. $j$ . (b (s:=t)) i) = (∑ i=0.. $j$ . b i)
apply(induct j)
apply simp-all
done

```

```

lemma Example2-lemma2:
  !!b. [j < n; b j = 0] ==> Suc (∑ i::nat=0.. $n$ . b i) = (∑ i=0.. $n$ . (b (j := Suc 0)) i)
apply(frule-tac b=(b (j:=(Suc 0))) in Example2-lemma2-aux)
apply(erule-tac t=sum (b(j := (Suc 0))) {0.. $n$ } in ssubst)
apply(frule-tac b=b in Example2-lemma2-aux)
apply(erule-tac t=sum b {0.. $n$ } in ssubst)
apply(subgoal-tac Suc (sum b {0.. $j$ } + b j + (∑ i=0.. $n - \text{Suc } j$ . b (Suc j + i))) = (sum b {0.. $j$ } + Suc (b j) + (∑ i=0.. $n - \text{Suc } j$ . b (Suc j + i))))
apply(rotate-tac -1)
apply(erule ssubst)
apply(subgoal-tac j ≤ j)
apply(drule-tac b=b and t=(Suc 0) in Example2-lemma2-aux2)
apply(rotate-tac -1)

```

```

apply(erule ssubst)
apply simp-all
done

```

```

record Example2 =
  c :: nat  $\Rightarrow$  nat
  x :: nat

```

```

lemma Example2:  $0 < n \implies$ 
 $\| - \{ 'x = 0 \wedge (\sum i = 0..<n. 'c\ i) = 0 \}$ 
  COBEGIN
    SCHEME [ $0 \leq i < n$ ]
     $\{ 'x = (\sum i = 0..<n. 'c\ i) \wedge 'c\ i = 0 \}$ 
     $\langle 'x := 'x + (Suc\ 0),, 'c := 'c\ (i := (Suc\ 0)) \rangle$ 
     $\{ 'x = (\sum i = 0..<n. 'c\ i) \wedge 'c\ i = (Suc\ 0) \}$ 
  COEND
 $\{ 'x = n \}$ 
apply oghoare
apply (simp-all cong del: sum.cong-simp)
apply (tactic  $\langle ALLGOALS\ (clarify-tac\ \textit{context}) \rangle$ )
apply (simp-all cong del: sum.cong-simp)
  apply(erule (1) Example2-lemma2)
  apply(erule (1) Example2-lemma2)
  apply(erule (1) Example2-lemma2)
apply(simp)
done

end

```

Chapter 2

Case Study: Single and Multi-Mutator Garbage Collection Algorithms

2.1 Formalization of the Memory

theory *Graph* imports *Main* begin

datatype *node* = *Black* | *White*

type-synonym *nodes* = *node list*

type-synonym *edge* = *nat* $\times$ *nat*

type-synonym *edges* = *edge list*

consts *Roots* :: *nat set*

definition *Proper-Roots* :: *nodes* $\Rightarrow$ *bool* **where**

$Proper-Roots\ M \equiv Roots \neq \{\} \wedge Roots \subseteq \{i. i < length\ M\}$

definition *Proper-Edges* :: (*nodes* $\times$ *edges*) $\Rightarrow$ *bool* **where**

$Proper-Edges \equiv (\lambda(M, E). \forall i < length\ E. fst(E!i) < length\ M \wedge snd(E!i) < length\ M)$

definition *BtoW* :: (*edge* $\times$ *nodes*) $\Rightarrow$ *bool* **where**

$BtoW \equiv (\lambda(e, M). (M!fst\ e) = Black \wedge (M!snd\ e) \neq Black)$

definition *Blacks* :: *nodes* $\Rightarrow$ *nat set* **where**

$Blacks\ M \equiv \{i. i < length\ M \wedge M!i = Black\}$

definition *Reach* :: *edges* $\Rightarrow$ *nat set* **where**

$Reach\ E \equiv \{x. (\exists path. 1 < length\ path \wedge path!(length\ path - 1) \in Roots \wedge x = path!0 \wedge (\forall i < length\ path - 1. (\exists j < length\ E. E!j = (path!(i+1), path!i)))) \vee x \in Roots\}$

Reach: the set of reachable nodes is the set of Roots together with the nodes reachable from some Root by a path represented by a list of nodes (at least two since we traverse at least one edge), where two consecutive nodes correspond to an edge in E.

2.1.1 Proofs about Graphs

lemmas *Graph-defs* = *Blacks-def Proper-Roots-def Proper-Edges-def BtoW-def*
declare *Graph-defs* [simp]

Graph 1

lemma *Graph1-aux* [rule-format]:

$$\begin{aligned} & \llbracket \text{Roots} \subseteq \text{Blacks } M; \forall i < \text{length } E. \neg \text{BtoW}(E!i, M) \rrbracket \\ & \implies 1 < \text{length } \text{path} \longrightarrow (\text{path}!(\text{length } \text{path} - 1)) \in \text{Roots} \longrightarrow \\ & (\forall i < \text{length } \text{path} - 1. (\exists j. j < \text{length } E \wedge E!j = (\text{path}!(\text{Suc } i), \text{path}!i))) \\ & \longrightarrow M!(\text{path}!0) = \text{Black} \end{aligned}$$

apply (*induct-tac path*)
apply *force*
apply *clarify*
apply *simp*
apply (*case-tac list*)
apply *force*
apply *simp*
apply (*rename-tac lista*)
apply (*rotate-tac -2*)
apply (*erule-tac x = 0 in all-dupE*)
apply *simp*
apply *clarify*
apply (*erule allE , erule (1) notE impE*)
apply *simp*
apply (*erule mp*)
apply (*case-tac lista*)
apply *force*
apply *simp*
apply (*erule mp*)
apply *clarify*
apply (*erule-tac x = Suc i in allE*)
apply *force*
done

lemma *Graph1*:

$$\begin{aligned} & \llbracket \text{Roots} \subseteq \text{Blacks } M; \text{Proper-Edges}(M, E); \forall i < \text{length } E. \neg \text{BtoW}(E!i, M) \rrbracket \\ & \implies \text{Reach } E \subseteq \text{Blacks } M \end{aligned}$$

apply (*unfold Reach-def*)
apply *simp*
apply *clarify*
apply (*erule disjE*)
apply *clarify*

```

apply(rule conjI)
apply(subgoal-tac 0 < length path - Suc 0)
apply(erule allE , erule (1) notE impE)
apply force
apply simp
apply(rule Graph1-aux)
apply auto
done

```

Graph 2

```

lemma Ex-first-occurrence [rule-format]:
   $P (n::nat) \longrightarrow (\exists m. P m \wedge (\forall i. i < m \longrightarrow \neg P i))$ 
apply(rule nat-less-induct)
apply clarify
apply(case-tac  $\forall m. m < n \longrightarrow \neg P m$ )
apply auto
done

```

```

lemma Compl-lemma:  $(n::nat) \leq l \implies (\exists m. m \leq l \wedge n = l - m)$ 
apply(rule-tac  $x = l - n$  in exI)
apply arith
done

```

```

lemma Ex-last-occurrence:
   $\llbracket P (n::nat); n \leq l \rrbracket \implies (\exists m. P (l - m) \wedge (\forall i. i < m \longrightarrow \neg P (l - i)))$ 
apply(drule Compl-lemma)
apply clarify
apply(erule Ex-first-occurrence)
done

```

```

lemma Graph2:
   $\llbracket T \in \text{Reach } E; R < \text{length } E \rrbracket \implies T \in \text{Reach } (E[R := (\text{fst}(E!R), T)])$ 
apply (unfold Reach-def)
apply clarify
apply simp
apply(case-tac  $\forall z < \text{length } \text{path}. \text{fst}(E!R) \neq \text{path}!z$ )
apply(rule-tac  $x = \text{path}$  in exI)
apply simp
apply clarify
apply(erule allE , erule (1) notE impE)
apply clarify
apply(rule-tac  $x = j$  in exI)
apply(case-tac  $j = R$ )
apply(erule-tac  $x = \text{Suc } i$  in allE)
apply simp
apply (force simp add: nth-list-update)
apply simp
apply(erule exE)

```

```

apply(subgoal-tac  $z \leq \text{length path} - \text{Suc } 0$ )
  prefer 2 apply arith
apply(drule-tac  $P = \lambda m. m < \text{length path} \wedge \text{fst}(E!R) = \text{path}!m$  in Ex-last-occurrence)
  apply assumption
apply clarify
apply simp
apply(rule-tac  $x = (\text{path}!0) \# (\text{drop } (\text{length path} - \text{Suc } m) \text{ path})$  in exI)
apply simp
apply(case-tac  $\text{length path} - (\text{length path} - \text{Suc } m)$ )
  apply arith
apply simp
apply(subgoal-tac  $(\text{length path} - \text{Suc } m) + \text{nat} \leq \text{length path}$ )
  prefer 2 apply arith
apply(subgoal-tac  $\text{length path} - \text{Suc } m + \text{nat} = \text{length path} - \text{Suc } 0$ )
  prefer 2 apply arith
apply clarify
apply(case-tac i)
  apply(force simp add: nth-list-update)
apply simp
apply(subgoal-tac  $(\text{length path} - \text{Suc } m) + \text{nat} \leq \text{length path}$ )
  prefer 2 apply arith
apply(subgoal-tac  $(\text{length path} - \text{Suc } m) + (\text{Suc } \text{nat}) \leq \text{length path}$ )
  prefer 2 apply arith
apply simp
apply(erule-tac  $x = \text{length path} - \text{Suc } m + \text{nat}$  in allE)
apply simp
apply clarify
apply(rule-tac  $x = j$  in exI)
apply(case-tac  $R=j$ )
  prefer 2 apply force
apply simp
apply(drule-tac  $t = \text{path} ! (\text{length path} - \text{Suc } m)$  in sym)
apply simp
apply(case-tac  $\text{length path} - \text{Suc } 0 < m$ )
  apply(subgoal-tac  $(\text{length path} - \text{Suc } m) = 0$ )
    prefer 2 apply arith
  apply(simp del: diff-is-0-eq)
  apply(subgoal-tac  $\text{Suc } \text{nat} \leq \text{nat}$ )
  prefer 2 apply arith
  apply(drule-tac  $n = \text{Suc } \text{nat}$  in Compl-lemma)
  apply clarify
  subgoal using  $[[\text{linarith-split-limit} = 0]]$  by force
apply(drule leI)
apply(subgoal-tac  $\text{Suc } (\text{length path} - \text{Suc } m + \text{nat}) = (\text{length path} - \text{Suc } 0) - (m - \text{Suc } \text{nat})$ )
  apply(erule-tac  $x = m - (\text{Suc } \text{nat})$  in allE)
  apply(case-tac m)
    apply simp
  apply simp

```

apply *simp*
done

Graph 3

declare *min.absorb1* [*simp*] *min.absorb2* [*simp*]

lemma *Graph3*:

$\llbracket T \in \text{Reach } E; R < \text{length } E \rrbracket \implies \text{Reach}(E[R := (\text{fst}(E!R), T)]) \subseteq \text{Reach } E$
apply (*unfold Reach-def*)
apply *clarify*
apply *simp*
apply (*case-tac* $\exists i < \text{length path} - 1. (\text{fst}(E!R), T) = (\text{path}!(\text{Suc } i), \text{path}!i)$)
— the changed edge is part of the path
apply (*erule exE*)
apply (*drule-tac* $P = \lambda i. i < \text{length path} - 1 \wedge (\text{fst}(E!R), T) = (\text{path}!\text{Suc } i, \text{path}!i)$)
in *Ex-first-occurrence*)
apply *clarify*
apply (*erule disjE*)
— T is NOT a root
apply *clarify*
apply (*rule-tac* $x = (\text{take } m \text{ path})@patha$ **in** *exI*)
apply (*subgoal-tac* $\neg(\text{length path} \leq m)$)
prefer 2 **apply** *arith*
apply (*simp*)
apply (*rule conjI*)
apply (*subgoal-tac* $\neg(m + \text{length patha} - 1 < m)$)
prefer 2 **apply** *arith*
apply (*simp add: nth-append*)
apply (*rule conjI*)
apply (*case-tac m*)
apply *force*
apply (*case-tac path*)
apply *force*
apply *force*
apply *clarify*
apply (*case-tac* $\text{Suc } i \leq m$)
apply (*erule-tac* $x = i$ **in** *allE*)
apply *simp*
apply *clarify*
apply (*rule-tac* $x = j$ **in** *exI*)
apply (*case-tac* $\text{Suc } i < m$)
apply (*simp add: nth-append*)
apply (*case-tac* $R=j$)
apply (*simp add: nth-list-update*)
apply (*case-tac* $i=m$)
apply *force*
apply (*erule-tac* $x = i$ **in** *allE*)
apply *force*

```

    apply(force simp add: nth-list-update)
  apply(simp add: nth-append)
  apply(subgoal-tac i=m - 1)
  prefer 2 apply arith
  apply(case-tac R=j)
  apply(erule-tac x = m - 1 in allE)
  apply(simp add: nth-list-update)
  apply(force simp add: nth-list-update)
  apply(simp add: nth-append)
  apply(rotate-tac -4)
  apply(erule-tac x = i - m in allE)
  apply(subgoal-tac Suc (i - m)=(Suc i - m) )
  prefer 2 apply arith
  apply simp
— T is a root
  apply(case-tac m=0)
  apply force
  apply(rule-tac x = take (Suc m) path in exI)
  apply(subgoal-tac ¬(length path≤Suc m) )
  prefer 2 apply arith
  apply clarsimp
  apply(erule-tac x = i in allE)
  apply simp
  apply clarify
  apply(case-tac R=j)
  apply(force simp add: nth-list-update)
  apply(force simp add: nth-list-update)
— the changed edge is not part of the path
  apply(rule-tac x = path in exI)
  apply simp
  apply clarify
  apply(erule-tac x = i in allE)
  apply clarify
  apply(case-tac R=j)
  apply(erule-tac x = i in allE)
  apply simp
  apply(force simp add: nth-list-update)
done

```

Graph 4

lemma *Graph4*:

```

  [[ T ∈ Reach E; Roots⊆Blacks M; I≤length E; T<length M; R<length E;
    ∀ i<I. ¬BtoW(E!i,M); R<I; M!fst(E!R)=Black; M!T≠Black]] ⇒
  (∃ r. I≤r ∧ r<length E ∧ BtoW(E[R:=(fst(E!R),T)]!r,M))
  apply (unfold Reach-def)
  apply simp
  apply(erule disjE)
  prefer 2 apply force

```

```

apply clarify
— there exist a black node in the path to T
apply(case-tac  $\exists m < \text{length } \text{path}. M!(\text{path}!m) = \text{Black}$ )
apply(erule exE)
apply(drule-tac  $P = \lambda m. m < \text{length } \text{path} \wedge M!(\text{path}!m) = \text{Black}$  in Ex-first-occurrence)
apply clarify
apply(case-tac ma)
apply force
apply simp
apply(case-tac  $\text{length } \text{path}$ )
apply force
apply simp
apply(erule-tac  $P = \lambda i. i < \text{nat} \longrightarrow P\ i$  and  $x = \text{nat}$  for  $P$  in allE)
apply simp
apply clarify
apply(erule-tac  $P = \lambda i. i < \text{Suc } \text{nat} \longrightarrow P\ i$  and  $x = \text{nat}$  for  $P$  in allE)
apply simp
apply(case-tac  $j < I$ )
apply(erule-tac  $x = j$  in allE)
apply force
apply(rule-tac  $x = j$  in exI)
apply(force simp add: nth-list-update)
apply simp
apply(rotate-tac  $-1$ )
apply(erule-tac  $x = \text{length } \text{path} - 1$  in allE)
apply(case-tac  $\text{length } \text{path}$ )
apply force
apply force
done

declare min.absorb1 [simp del] min.absorb2 [simp del]

```

Graph 5

lemma *Graph5*:

```

  [|  $T \in \text{Reach } E$  ;  $\text{Roots} \subseteq \text{Blacks } M$  ;  $\forall i < R. \neg \text{BtoW}(E!i, M)$  ;  $T < \text{length } M$  ;
     $R < \text{length } E$  ;  $M!\text{fst}(E!R) = \text{Black}$  ;  $M!\text{snd}(E!R) = \text{Black}$  ;  $M!T \neq \text{Black}$  |]
   $\implies (\exists r. R < r \wedge r < \text{length } E \wedge \text{BtoW}(E[R := (\text{fst}(E!R), T)]!r, M))$ 

```

apply (*unfold* *Reach-def*)

apply *simp*

apply(*erule* *disjE*)

prefer 2 **apply** *force*

apply *clarify*

— there exist a black node in the path to T

apply(*case-tac* $\exists m < \text{length } \text{path}. M!(\text{path}!m) = \text{Black}$)

apply(*erule* *exE*)

apply(*drule-tac* $P = \lambda m. m < \text{length } \text{path} \wedge M!(\text{path}!m) = \text{Black}$ **in** *Ex-first-occurrence*)

apply *clarify*

apply(*case-tac* *ma*)

```

  apply force
  apply simp
  apply(case-tac length path)
  apply force
  apply simp
  apply(erule-tac  $P = \lambda i. i < \text{nat} \longrightarrow P\ i$  and  $x = \text{nat}$  for  $P$  in  $\text{all}E$ )
  apply simp
  apply clarify
  apply(erule-tac  $P = \lambda i. i < \text{Suc nat} \longrightarrow P\ i$  and  $x = \text{nat}$  for  $P$  in  $\text{all}E$ )
  apply simp
  apply(case-tac  $j \leq R$ )
  apply(drule le-imp-less-or-eq [of -  $R$ ])
  apply(erule disjE)
  apply(erule allE , erule (1) notE impE)
  apply force
  apply force
  apply(rule-tac  $x = j$  in  $\text{exI}$ )
  apply(force simp add: nth-list-update)
  apply simp
  apply(rotate-tac -1)
  apply(erule-tac  $x = \text{length path} - 1$  in  $\text{all}E$ )
  apply(case-tac length path)
  apply force
  apply force
done

```

Other lemmas about graphs

lemma *Graph6*:

```

[[Proper-Edges( $M, E$ );  $R < \text{length } E$  ;  $T < \text{length } M$ ]]  $\implies$  Proper-Edges( $M, E[R := (\text{fst}(E!R), T)]$ )
apply (unfold Proper-Edges-def)
apply(force simp add: nth-list-update)
done

```

lemma *Graph7*:

```

[[Proper-Edges( $M, E$ )]  $\implies$  Proper-Edges( $M[T := a], E$ )
apply (unfold Proper-Edges-def)
apply force
done

```

lemma *Graph8*:

```

[[Proper-Roots( $M$ )]  $\implies$  Proper-Roots( $M[T := a]$ )
apply (unfold Proper-Roots-def)
apply force
done

```

Some specific lemmata for the verification of garbage collection algorithms.

```

lemma Graph9:  $j < \text{length } M \implies \text{Blacks } M \subseteq \text{Blacks } (M[j := \text{Black}])$ 
apply (unfold Blacks-def)

```

```

  apply(force simp add: nth-list-update)
done

```

```

lemma Graph10 [rule-format (no-asm)]:  $\forall i. M!i=a \longrightarrow M[i:=a]=M$ 
  apply(induct-tac M)
  apply auto
  apply(case-tac i)
  apply auto
done

```

```

lemma Graph11 [rule-format (no-asm)]:
   $\llbracket M!j \neq \text{Black}; j < \text{length } M \rrbracket \implies \text{Blacks } M \subset \text{Blacks } (M[j := \text{Black}])$ 
  apply (unfold Blacks-def)
  apply(rule psubsetI)
  apply(force simp add: nth-list-update)
  apply safe
  apply(erule-tac c = j in equalityCE)
  apply auto
done

```

```

lemma Graph12:  $\llbracket a \subseteq \text{Blacks } M; j < \text{length } M \rrbracket \implies a \subseteq \text{Blacks } (M[j := \text{Black}])$ 
  apply (unfold Blacks-def)
  apply(force simp add: nth-list-update)
done

```

```

lemma Graph13:  $\llbracket a \subset \text{Blacks } M; j < \text{length } M \rrbracket \implies a \subset \text{Blacks } (M[j := \text{Black}])$ 
  apply (unfold Blacks-def)
  apply(erule psubset-subset-trans)
  apply(force simp add: nth-list-update)
done

```

```

declare Graph-defs [simp del]

```

```

end

```

2.2 The Single Mutator Case

```

theory Gar-Coll imports Graph OG-Syntax begin

```

```

declare psubsetE [rule del]

```

Declaration of variables:

```

record gar-coll-state =
  M :: nodes
  E :: edges
  bc :: nat set
  obc :: nat set
  Ma :: nodes
  ind :: nat

```

$k :: \text{nat}$
 $z :: \text{bool}$

2.2.1 The Mutator

The mutator first redirects an arbitrary edge R from an arbitrary accessible node towards an arbitrary accessible node T . It then colors the new target T black.

We declare the arbitrarily selected node and edge as constants:

consts $R :: \text{nat}$ $T :: \text{nat}$

The following predicate states, given a list of nodes m and a list of edges e , the conditions under which the selected edge R and node T are valid:

definition $\text{Mut-init} :: \text{gar-coll-state} \Rightarrow \text{bool}$ **where**

$\text{Mut-init} \equiv \llbracket T \in \text{Reach } 'E \wedge R < \text{length } 'E \wedge T < \text{length } 'M \rrbracket$

For the mutator we consider two modules, one for each action. An auxiliary variable $'z$ is set to false if the mutator has already redirected an edge but has not yet colored the new target.

definition $\text{Redirect-Edge} :: \text{gar-coll-state} \text{ ann-com}$ **where**

$\text{Redirect-Edge} \equiv \llbracket ' \text{Mut-init} \wedge 'z \rrbracket \langle 'E := 'E[R := (\text{fst } 'E!R), T] \rrbracket, 'z := (\neg 'z) \rangle$

definition $\text{Color-Target} :: \text{gar-coll-state} \text{ ann-com}$ **where**

$\text{Color-Target} \equiv \llbracket ' \text{Mut-init} \wedge \neg 'z \rrbracket \langle 'M := 'M[T := \text{Black}] \rrbracket, 'z := (\neg 'z) \rangle$

definition $\text{Mutator} :: \text{gar-coll-state} \text{ ann-com}$ **where**

$\text{Mutator} \equiv$
 $\llbracket ' \text{Mut-init} \wedge 'z \rrbracket$
 $\text{WHILE True INV } \llbracket ' \text{Mut-init} \wedge 'z \rrbracket$
 $\text{DO } \text{Redirect-Edge} ;; \text{Color-Target } \text{OD}$

Correctness of the mutator

lemmas $\text{mutator-defs} = \text{Mut-init-def } \text{Redirect-Edge-def } \text{Color-Target-def}$

lemma Redirect-Edge :

$\vdash \text{Redirect-Edge } \text{pre}(\text{Color-Target})$

apply $(\text{unfold mutator-defs})$

apply annhoare

apply (simp-all)

apply $(\text{force elim:Graph2})$

done

lemma Color-Target :

$\vdash \text{Color-Target } \llbracket ' \text{Mut-init} \wedge 'z \rrbracket$

apply $(\text{unfold mutator-defs})$

apply annhoare

```

apply(simp-all)
done

```

```

lemma Mutator:
   $\vdash \text{Mutator } \{\!\!| \text{False} |\!\!\}$ 
apply(unfold Mutator-def)
apply annhoare
apply(simp-all add:Redirect-Edge Color-Target)
apply(simp add:mutator-defs)
done

```

2.2.2 The Collector

A constant $M\text{-init}$ is used to give $\text{'}Ma$ a suitable first value, defined as a list of nodes where only the $Roots$ are black.

```

consts M-init :: nodes

```

```

definition Proper-M-init :: gar-coll-state  $\Rightarrow$  bool where
  Proper-M-init  $\equiv$  « Blacks M-init=Roots  $\wedge$  length M-init=length  $\text{'}M$  »

```

```

definition Proper :: gar-coll-state  $\Rightarrow$  bool where
  Proper  $\equiv$  « Proper-Roots  $\text{'}M \wedge$  Proper-Edges( $\text{'}M$ ,  $\text{'}E$ )  $\wedge$   $\text{'}Proper\text{-}M\text{-init}$  »

```

```

definition Safe :: gar-coll-state  $\Rightarrow$  bool where
  Safe  $\equiv$  « Reach  $\text{'}E \subseteq$  Blacks  $\text{'}M$  »

```

```

lemmas collector-defs = Proper-M-init-def Proper-def Safe-def

```

Blackening the roots

```

definition Blacken-Roots :: gar-coll-state ann-com where
  Blacken-Roots  $\equiv$ 
     $\{\!\!| \text{'}Proper$ 
     $\text{'}ind:=0;;$ 
     $\{\!\!| \text{'}Proper \wedge \text{'}ind=0$ 
    WHILE  $\text{'}ind < \text{length } \text{'}M$ 
      INV  $\{\!\!| \text{'}Proper \wedge (\forall i < \text{'}ind. i \in \text{Roots} \longrightarrow \text{'}M[i]=\text{Black}) \wedge \text{'}ind \leq \text{length } \text{'}M$ 
      DO  $\{\!\!| \text{'}Proper \wedge (\forall i < \text{'}ind. i \in \text{Roots} \longrightarrow \text{'}M[i]=\text{Black}) \wedge \text{'}ind < \text{length } \text{'}M$ 
      IF  $\text{'}ind \in \text{Roots}$  THEN
         $\{\!\!| \text{'}Proper \wedge (\forall i < \text{'}ind. i \in \text{Roots} \longrightarrow \text{'}M[i]=\text{Black}) \wedge \text{'}ind < \text{length } \text{'}M \wedge$ 
 $\text{'}ind \in \text{Roots}$ 
         $\text{'}M := \text{'}M[\text{'}ind:=\text{Black}] \text{ FI};;$ 
         $\{\!\!| \text{'}Proper \wedge (\forall i < \text{'}ind+1. i \in \text{Roots} \longrightarrow \text{'}M[i]=\text{Black}) \wedge \text{'}ind < \text{length } \text{'}M$ 
         $\text{'}ind := \text{'}ind+1$ 
      OD
     $\}\!\!\}$ 

```

```

lemma Blacken-Roots:
   $\vdash \text{Blacken-Roots } \{\!\!| \text{'}Proper \wedge \text{Roots} \subseteq \text{Blacks } \text{'}M |\!\!\}$ 
apply (unfold Blacken-Roots-def)

```

```

apply annhoare
apply(simp-all add:collector-defs Graph-defs)
apply safe
apply(simp-all add:nth-list-update)
  apply (erule less-SucE)
  apply simp+
  apply force
apply force
done

```

Propagating black

definition *PBInv* :: *gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**

$$PBInv \equiv \llbracket \lambda ind. 'obc < Blacks\ 'M \vee (\forall i < ind. \neg BtoW\ ('E!i, 'M) \vee (\neg 'z \wedge i=R \wedge (snd('E!R)) = T \wedge (\exists r. ind \leq r \wedge r < length\ 'E \wedge BtoW('E!r, 'M)))) \rrbracket$$

definition *Propagate-Black-aux* :: *gar-coll-state ann-com* **where**

$$Propagate-Black-aux \equiv$$

$$\llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M \rrbracket$$

$$'ind := 0;;$$

$$\llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M \wedge 'ind = 0 \rrbracket$$

$$WHILE\ 'ind < length\ 'E$$

$$INV\ \llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$$

$$\wedge 'PBInv\ 'ind \wedge 'ind \leq length\ 'E \rrbracket$$

$$DO\ \llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$$

$$\wedge 'PBInv\ 'ind \wedge 'ind < length\ 'E \rrbracket$$

$$IF\ 'M!(fst\ ('E!'ind)) = Black\ THEN$$

$$\llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$$

$$\wedge 'PBInv\ 'ind \wedge 'ind < length\ 'E \wedge 'M!fst('E!'ind) = Black \rrbracket$$

$$'M := 'M[snd('E!'ind) := Black];;$$

$$\llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$$

$$\wedge 'PBInv\ ('ind + 1) \wedge 'ind < length\ 'E \rrbracket$$

$$'ind := 'ind + 1$$

$$FI$$

$$OD$$

lemma *Propagate-Black-aux*:

$\vdash Propagate-Black-aux$

$$\llbracket 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$$

$$\wedge ('obc < Blacks\ 'M \vee 'Safe) \rrbracket$$

```

apply (unfold Propagate-Black-aux-def PBInv-def collector-defs)
apply annhoare
apply(simp-all add:Graph6 Graph7 Graph8 Graph12)
  apply force
  apply force
  apply force
— 4 subgoals left
apply clarify
apply(simp add:Proper-Edges-def Proper-Roots-def Graph6 Graph7 Graph8 Graph12)

```

```

apply (erule disjE)
apply(rule disjI1)
apply(erule Graph13)
apply force
apply (case-tac M x ! snd (E x ! ind x)=Black)
apply (simp add: Graph10 BtoW-def)
apply (rule disjI2)
apply clarify
apply (erule less-SucE)
apply (erule-tac x=i in allE , erule (1) notE impE)
apply simp
apply clarify
apply (drule-tac y = r in le-imp-less-or-eq)
apply (erule disjE)
apply (subgoal-tac Suc (ind x)≤r)
apply fast
apply arith
apply fast
apply fast
apply fast
apply(rule disjI1)
apply(erule subset-psubset-trans)
apply(erule Graph11)
apply fast
— 3 subgoals left
apply force
apply force
— last
apply clarify
apply simp
apply(subgoal-tac ind x = length (E x))
apply (simp)
apply(drule Graph1)
apply simp
apply clarify
apply(erule allE, erule impE, assumption)
apply force
apply force
apply arith
done

```

Refining propagating black

definition Auxk :: gar-coll-state $\Rightarrow$ bool **where**

$$\begin{aligned}
 \text{Auxk} \equiv & \ll 'k < \text{length } 'M \wedge ('M! 'k \neq \text{Black} \vee \neg \text{BtoW}('E! 'ind, 'M) \vee \\
 & 'obc < \text{Blacks } 'M \vee (\neg 'z \wedge 'ind = R \wedge \text{snd}('E! R) = T \\
 & \wedge (\exists r. 'ind < r \wedge r < \text{length } 'E \wedge \text{BtoW}('E! r, 'M))) \gg
 \end{aligned}$$

definition Propagate-Black :: gar-coll-state $\Rightarrow$ bool **where**

$$\text{Propagate-Black} \equiv$$

```

 $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M \}$ 
 $\text{'ind}:=0;;$ 
 $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M \wedge \text{'ind}=0 \}$ 
WHILE  $\text{'ind} < \text{length 'E}$ 
  INV  $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge \text{'PBI}nv \text{ 'ind} \wedge \text{'ind} \leq \text{length 'E} \}$ 
  DO  $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge \text{'PBI}nv \text{ 'ind} \wedge \text{'ind} < \text{length 'E} \}$ 
  IF  $(\text{'M}!(fst(\text{'E}!\text{'ind})))=Black$  THEN
     $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge \text{'PBI}nv \text{ 'ind} \wedge \text{'ind} < \text{length 'E} \wedge (\text{'M}!(fst(\text{'E}!\text{'ind})))=Black \}$ 
 $\text{'k}:=(snd(\text{'E}!\text{'ind}));;$ 
 $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge \text{'PBI}nv \text{ 'ind} \wedge \text{'ind} < \text{length 'E} \wedge (\text{'M}!(fst(\text{'E}!\text{'ind})))=Black$ 
 $\wedge \text{'Aux}k \}$ 
 $\langle \text{'M}:=\text{'M}[\text{'k}:=Black],, \text{'ind}:=\text{'ind}+1 \rangle$ 
  ELSE  $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge \text{'PBI}nv \text{ 'ind} \wedge \text{'ind} < \text{length 'E} \}$ 
 $\langle IF(\text{'M}!(fst(\text{'E}!\text{'ind}))) \neq Black THEN \text{'ind}:=\text{'ind}+1 FI \rangle$ 
FI
OD

```

lemma *Propagate-Black*:

```

 $\vdash \text{Propagate-Black}$ 
 $\{\{ 'Proper \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$ 
 $\wedge (\text{'obc} < Blacks \text{ ' } M \vee \text{'Safe}) \}$ 
apply (unfold Propagate-Black-def PBI}nv-def Auxk-def collector-defs)
apply annhoare
apply (simp-all add: Graph6 Graph7 Graph8 Graph12)
  apply force
  apply force
  apply force
— 5 subgoals left
apply clarify
apply (simp add:BtoW-def Proper-Edges-def)
— 4 subgoals left
apply clarify
apply (simp add:Proper-Edges-def Graph6 Graph7 Graph8 Graph12)
apply (erule disjE)
apply (rule disjI1)
apply (erule psubset-subset-trans)
apply (erule Graph9)
apply (case-tac M x!k x=Black)
apply (case-tac M x ! snd (E x ! ind x)=Black)
apply (simp add: Graph10 BtoW-def)
apply (rule disjI2)
apply clarify
apply (erule less-SucE)
apply (erule-tac x=i in allE , erule (1) notE impE)

```

```

apply simp
apply clarify
apply (drule-tac  $y = r$  in le-imp-less-or-eq)
apply (erule disjE)
  apply (subgoal-tac Suc ( $\text{ind } x \leq r$ )
    apply fast
    apply arith
    apply fast
  apply fast
apply (simp add: Graph10 BtoW-def)
apply (erule disjE)
apply (erule disjI1)
apply clarify
apply (erule less-SucE)
  apply force
apply simp
apply (subgoal-tac Suc  $R \leq r$ )
  apply fast
apply arith
apply(rule disjI1)
apply(erule subset-psubset-trans)
apply(erule Graph11)
apply fast
— 2 subgoals left
apply clarify
apply(simp add: Proper-Edges-def Graph6 Graph7 Graph8 Graph12)
apply (erule disjE)
  apply fast
apply clarify
apply (erule less-SucE)
apply (erule-tac  $x=i$  in allE , erule (1) notE impE)
apply simp
apply clarify
apply (drule-tac  $y = r$  in le-imp-less-or-eq)
apply (erule disjE)
  apply (subgoal-tac Suc ( $\text{ind } x \leq r$ )
    apply fast
    apply arith
  apply (simp add: BtoW-def)
apply (simp add: BtoW-def)
— last
apply clarify
apply simp
apply(subgoal-tac  $\text{ind } x = \text{length } (E \ x)$ )
apply (simp)
apply(drule Graph1)
  apply simp
apply clarify
apply(erule allE, erule impE, assumption)

```

apply force
 apply force
 apply arith
 done

Counting black nodes

definition *CountInv* :: *gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
CountInv $\equiv$ « $\lambda ind. \{i. i < ind \wedge 'Ma!i=Black\} \subseteq 'bc$ »

definition *Count* :: *gar-coll-state* *ann-com* **where**

Count $\equiv$
 $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'bc = \{ \} \}$
 $'ind := 0;;$
 $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'bc = \{ \}$
 $\wedge 'ind = 0 \}$
WHILE $'ind < length \ 'M$
INV $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge 'CountInv \ 'ind$
 $\wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'ind \leq length \ 'M \}$
DO $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge 'CountInv \ 'ind$
 $\wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'ind < length \ 'M \}$
IF $'M! 'ind = Black$
THEN $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge 'CountInv \ 'ind$
 $\wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'ind < length \ 'M \wedge 'M! 'ind = Black \}$
 $'bc := insert \ 'ind \ 'bc$
FI;;
 $\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge 'CountInv \ ('ind + 1)$
 $\wedge ('obc < Blacks \ 'Ma \vee 'Safe) \wedge 'ind < length \ 'M \}$
 $'ind := 'ind + 1$
OD

lemma *Count*:

$\vdash Count$

$\{ \{ 'Proper \wedge Roots \subseteq Blacks \ 'M$
 $\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq 'bc \wedge 'bc \subseteq Blacks \ 'M \wedge length \ 'Ma = length$
 $\ 'M$
 $\wedge ('obc < Blacks \ 'Ma \vee 'Safe) \}$

```

apply(unfold Count-def)
apply annhoare
apply(simp-all add:CountInv-def Graph6 Graph7 Graph8 Graph12 Blacks-def col-
lector-defs)
  apply force
  apply force
  apply force
  apply clarify
  apply simp
  apply(fast elim:less-SucE)
  apply clarify
  apply simp
  apply(fast elim:less-SucE)
  apply force
apply force
done

```

Appending garbage nodes to the free list

axiomatization *Append-to-free* :: $\text{nat} \times \text{edges} \Rightarrow \text{edges}$

where

Append-to-free0: $\text{length} (\text{Append-to-free} (i, e)) = \text{length } e$ **and**

Append-to-free1: $\text{Proper-Edges} (m, e)$

$\implies \text{Proper-Edges} (m, \text{Append-to-free}(i, e))$ **and**

Append-to-free2: $i \notin \text{Reach } e$

$\implies n \in \text{Reach} (\text{Append-to-free}(i, e)) = (n = i \vee n \in \text{Reach } e)$

definition *AppendInv* :: $\text{gar-coll-state} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

AppendInv $\equiv \llbracket \lambda \text{ind}. \forall i < \text{length } 'M. \text{ind} \leq i \longrightarrow i \in \text{Reach } 'E \longrightarrow 'M!i = \text{Black} \rrbracket$

definition *Append* :: $\text{gar-coll-state} \text{ ann-com}$ **where**

Append $\equiv$

$\llbracket 'Proper \wedge \text{Roots} \subseteq \text{Blacks } 'M \wedge 'Safe \rrbracket$

$'ind := 0;$

$\llbracket 'Proper \wedge \text{Roots} \subseteq \text{Blacks } 'M \wedge 'Safe \wedge 'ind = 0 \rrbracket$

WHILE $'ind < \text{length } 'M$

INV $\llbracket 'Proper \wedge 'AppendInv 'ind \wedge 'ind \leq \text{length } 'M \rrbracket$

DO $\llbracket 'Proper \wedge 'AppendInv 'ind \wedge 'ind < \text{length } 'M \rrbracket$

IF $'M!'ind = \text{Black}$ *THEN*

$\llbracket 'Proper \wedge 'AppendInv 'ind \wedge 'ind < \text{length } 'M \wedge 'M!'ind = \text{Black} \rrbracket$

$'M := 'M['ind := \text{White}]$

ELSE $\llbracket 'Proper \wedge 'AppendInv 'ind \wedge 'ind < \text{length } 'M \wedge 'ind \notin \text{Reach } 'E \rrbracket$

$'E := \text{Append-to-free}('ind, 'E)$

FI;;

$\llbracket 'Proper \wedge 'AppendInv ('ind + 1) \wedge 'ind < \text{length } 'M \rrbracket$

$'ind := 'ind + 1$

OD

lemma *Append*:

```

  ⊢ Append { ' Proper }
apply(unfold Append-def AppendInv-def)
apply annhoare
apply(simp-all add:collector-defs Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1
Graph12)
  apply(force simp:Blacks-def nth-list-update)
  apply force
  apply force
  apply(force simp add:Graph-defs)
  apply force
apply clarify
apply simp
apply(rule conjI)
  apply (erule Append-to-free1)
apply clarify
apply (drule-tac n = i in Append-to-free2)
apply force
apply force
apply force
done

```

Correctness of the Collector

definition *Collector* :: *gar-coll-state ann-com* **where**

```

Collector ≡
{ ' Proper }
WHILE True INV { ' Proper }
DO
  Blacken-Roots;;
  { ' Proper ∧ Roots ⊆ Blacks ' M }
  ' obc := { };;
  { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' obc = { } }
  ' bc := Roots;;
  { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' obc = { } ∧ ' bc = Roots }
  ' Ma := M-init;;
  { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' obc = { } ∧ ' bc = Roots ∧ ' Ma = M-init }
  WHILE ' obc ≠ ' bc
    INV { ' Proper ∧ Roots ⊆ Blacks ' M
      ∧ ' obc ⊆ Blacks ' Ma ∧ Blacks ' Ma ⊆ ' bc ∧ ' bc ⊆ Blacks ' M
      ∧ length ' Ma = length ' M ∧ ( ' obc < Blacks ' Ma ∨ ' Safe ) }
    DO { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' bc ⊆ Blacks ' M }
      ' obc := ' bc;;
      Propagate-Black;;
      { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' obc ⊆ Blacks ' M ∧ ' bc ⊆ Blacks ' M
        ∧ ( ' obc < Blacks ' M ∨ ' Safe ) }
      ' Ma := ' M;;
      { ' Proper ∧ Roots ⊆ Blacks ' M ∧ ' obc ⊆ Blacks ' Ma
        ∧ Blacks ' Ma ⊆ Blacks ' M ∧ ' bc ⊆ Blacks ' M ∧ length ' Ma = length ' M
        ∧ ( ' obc < Blacks ' Ma ∨ ' Safe ) }

```

```

      'bc:={};;
      Count
    OD;;
    Append
  OD

```

```

lemma Collector:
  ⊢ Collector ⌊False⌋
apply(unfold Collector-def)
apply annhoare
apply(simp-all add: Blacken-Roots Propagate-Black Count Append)
apply(simp-all add:Blacken-Roots-def Propagate-Black-def Count-def Append-def
  collector-defs)
  apply (force simp add: Proper-Roots-def)
  apply force
  apply force
apply clarify
apply (erule disjE)
apply(simp add:psubsetI)
  apply(force dest:subset-antisym)
done

```

2.2.3 Interference Freedom

```

lemmas modules = Redirect-Edge-def Color-Target-def Blacken-Roots-def
  Propagate-Black-def Count-def Append-def
lemmas Invariants = PBIInv-def Auxk-def CountInv-def AppendInv-def
lemmas abbrev = collector-defs mutator-defs Invariants

```

```

lemma interfree-Blacken-Roots-Redirect-Edge:
  interfree-aux (Some Blacken-Roots, {}, Some Redirect-Edge)
apply (unfold modules)
apply interfree-aux
apply safe
apply (simp-all add:Graph6 Graph12 abbrev)
done

```

```

lemma interfree-Redirect-Edge-Blacken-Roots:
  interfree-aux (Some Redirect-Edge, {}, Some Blacken-Roots)
apply (unfold modules)
apply interfree-aux
apply safe
apply(simp add:abbrev)+
done

```

```

lemma interfree-Blacken-Roots-Color-Target:
  interfree-aux (Some Blacken-Roots, {}, Some Color-Target)
apply (unfold modules)
apply interfree-aux

```

```

apply safe
apply(simp-all add:Graph7 Graph8 nth-list-update abbrev)
done

```

```

lemma interfree-Color-Target-Blacken-Roots:
  interfree-aux (Some Color-Target, {}, Some Blacken-Roots)
apply (unfold modules)
apply interfree-aux
apply safe
apply(simp add:abbrev)+
done

```

```

lemma interfree-Propagate-Black-Redirect-Edge:
  interfree-aux (Some Propagate-Black, {}, Some Redirect-Edge)
apply (unfold modules)
apply interfree-aux
— 11 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(erule conjE)+
apply(erule disjE, erule disjI1, rule disjI2, rule allI, (rule impI)+, case-tac R=i,
rule conjI, erule sym)
apply(erule Graph4)
  apply(simp)+
  apply (simp add:BtoW-def)
  apply (simp add:BtoW-def)
apply(rule conjI)
  apply (force simp add:BtoW-def)
apply(erule Graph4)
  apply simp+
— 7 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(erule conjE)+
apply(erule disjE, erule disjI1, rule disjI2, rule allI, (rule impI)+, case-tac R=i,
rule conjI, erule sym)
apply(erule Graph4)
  apply(simp)+
  apply (simp add:BtoW-def)
  apply (simp add:BtoW-def)
apply(rule conjI)
  apply (force simp add:BtoW-def)
apply(erule Graph4)
  apply simp+
— 6 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(erule conjE)+
apply(rule conjI)

```

```

apply(erule disjE, erule disjI1, rule disjI2, rule allI, (rule impI)+, case-tac R=i,
rule conjI, erule sym)
  apply(erule Graph4)
    apply(simp)+
    apply (simp add:BtoW-def)
    apply (simp add:BtoW-def)
  apply(rule conjI)
  apply (force simp add:BtoW-def)
apply(erule Graph4)
  apply simp+
apply(simp add:BtoW-def nth-list-update)
apply force
— 5 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
— 4 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(rule conjI)
apply(erule disjE, erule disjI1, rule disjI2, rule allI, (rule impI)+, case-tac R=i,
rule conjI, erule sym)
  apply(erule Graph4)
    apply(simp)+
    apply (simp add:BtoW-def)
    apply (simp add:BtoW-def)
  apply(rule conjI)
  apply (force simp add:BtoW-def)
apply(erule Graph4)
  apply simp+
apply(rule conjI)
  apply(simp add:nth-list-update)
  apply force
apply(rule impI, rule impI, erule disjE, erule disjI1, case-tac R = (ind x) ,case-tac
M x ! T = Black)
  apply(force simp add:BtoW-def)
  apply(case-tac M x !snd (E x ! ind x)=Black)
  apply(rule disjI2)
  apply simp
  apply (erule Graph5)
  apply simp+
  apply(force simp add:BtoW-def)
apply(force simp add:BtoW-def)
— 3 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
— 2 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12)
apply(erule disjE, erule disjI1, rule disjI2, rule allI, (rule impI)+, case-tac R=i,
rule conjI, erule sym)
  apply clarify
  apply(erule Graph4)
  apply(simp)+

```

```

    apply (simp add:BtoW-def)
  apply (simp add:BtoW-def)
  apply(rule conjI)
  apply (force simp add:BtoW-def)
  apply(erule Graph4)
  apply simp+
done

```

```

lemma interfree-Redirect-Edge-Propagate-Black:
  interfree-aux (Some Redirect-Edge, {}, Some Propagate-Black)
  apply (unfold modules )
  apply interfree-aux
  apply(clarify, simp add:abbrev)+
done

```

```

lemma interfree-Propagate-Black-Color-Target:
  interfree-aux (Some Propagate-Black, {}, Some Color-Target)
  apply (unfold modules )
  apply interfree-aux
  — 11 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)+
  apply(erule conjE)+
  apply(erule disjE,rule disjI1,erule psubset-subset-trans,erule Graph9,
    case-tac M x!T=Black, rule disjI2,rotate-tac -1, simp add: Graph10, clarify,
    erule allE, erule impE, assumption, erule impE, assumption,
    simp add:BtoW-def, rule disjI1, erule subset-psubset-trans, erule Graph11,
    force)
  — 7 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
  apply(erule conjE)+
  apply(erule disjE,rule disjI1,erule psubset-subset-trans,erule Graph9,
    case-tac M x!T=Black, rule disjI2,rotate-tac -1, simp add: Graph10, clarify,
    erule allE, erule impE, assumption, erule impE, assumption,
    simp add:BtoW-def, rule disjI1, erule subset-psubset-trans, erule Graph11,
    force)
  — 6 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
  apply clarify
  apply (rule conjI)
  apply(erule disjE,rule disjI1,erule psubset-subset-trans,erule Graph9,
    case-tac M x!T=Black, rule disjI2,rotate-tac -1, simp add: Graph10, clarify,
    erule allE, erule impE, assumption, erule impE, assumption,
    simp add:BtoW-def, rule disjI1, erule subset-psubset-trans, erule Graph11,
    force)
  apply(simp add:nth-list-update)
  — 5 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
  — 4 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)

```

```

apply (rule conjI)
  apply(erule disjE,rule disjI1,erule psubset-subset-trans,erule Graph9,
    case-tac M x!T=Black, rule disjI2,rotate-tac -1, simp add: Graph10, clarify,
    erule allE, erule impE, assumption, erule impE, assumption,
    simp add:BtoW-def, rule disjI1, erule subset-psubset-trans, erule Graph11,
    force)
  apply(rule conjI)
  apply(simp add:nth-list-update)
  apply(rule impI,rule impI, case-tac M x!T=Black,rotate-tac -1, force simp add:
    BtoW-def Graph10,
    erule subset-psubset-trans, erule Graph11, force)
  — 3 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
  — 2 subgoals left
  apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
  apply(erule disjE,rule disjI1,erule psubset-subset-trans,erule Graph9,
    case-tac M x!T=Black, rule disjI2,rotate-tac -1, simp add: Graph10, clarify,
    erule allE, erule impE, assumption, erule impE, assumption,
    simp add:BtoW-def, rule disjI1, erule subset-psubset-trans, erule Graph11,
    force)
  — 3 subgoals left
  apply(simp add:abbrev)
  done

```

```

lemma interfree-Color-Target-Propagate-Black:
  interfree-aux (Some Color-Target, {}, Some Propagate-Black)
apply (unfold modules )
apply interfree-aux
apply(clarify, simp add:abbrev)+
done

```

```

lemma interfree-Count-Redirect-Edge:
  interfree-aux (Some Count, {}, Some Redirect-Edge)
apply (unfold modules)
apply interfree-aux
  — 9 subgoals left
apply(simp-all add:abbrev Graph6 Graph12)
  — 6 subgoals left
apply(clarify, simp add:abbrev Graph6 Graph12,
  erule disjE,erule disjI1,rule disjI2,rule subset-trans, erule Graph3,force,force)+
done

```

```

lemma interfree-Redirect-Edge-Count:
  interfree-aux (Some Redirect-Edge, {}, Some Count)
apply (unfold modules )
apply interfree-aux
apply(clarify,simp add:abbrev)+
apply(simp add:abbrev)
done

```

lemma *interfree-Count-Color-Target:*
interfree-aux (Some Count, {}, Some Color-Target)
apply (unfold modules)
apply *interfree-aux*
— 9 subgoals left
apply(simp-all add:abbrev Graph7 Graph8 Graph12)
— 6 subgoals left
apply(clarify,simp add:abbrev Graph7 Graph8 Graph12,
erule disjE, erule disjI1, rule disjI2,erule subset-trans, erule Graph9)+
— 2 subgoals left
apply(clarify, simp add:abbrev Graph7 Graph8 Graph12)
apply(rule conjI)
apply(erule disjE, erule disjI1, rule disjI2,erule subset-trans, erule Graph9)
apply(simp add:nth-list-update)
— 1 subgoal left
apply(clarify, simp add:abbrev Graph7 Graph8 Graph12,
erule disjE, erule disjI1, rule disjI2,erule subset-trans, erule Graph9)
done

lemma *interfree-Color-Target-Count:*
interfree-aux (Some Color-Target, {}, Some Count)
apply (unfold modules)
apply *interfree-aux*
apply(clarify, simp add:abbrev)+
apply(simp add:abbrev)
done

lemma *interfree-Append-Redirect-Edge:*
interfree-aux (Some Append, {}, Some Redirect-Edge)
apply (unfold modules)
apply *interfree-aux*
apply(simp-all add:abbrev Graph6 Append-to-free0 Append-to-free1 Graph12)
apply(clarify, simp add:abbrev Graph6 Append-to-free0 Append-to-free1 Graph12,
force dest:Graph3)+
done

lemma *interfree-Redirect-Edge-Append:*
interfree-aux (Some Redirect-Edge, {}, Some Append)
apply (unfold modules)
apply *interfree-aux*
apply(clarify, simp add:abbrev Append-to-free0)+
apply (force simp add: Append-to-free2)
apply(clarify, simp add:abbrev Append-to-free0)+
done

lemma *interfree-Append-Color-Target:*
interfree-aux (Some Append, {}, Some Color-Target)
apply (unfold modules)

```

apply interfree-aux
apply(clarify, simp add:abbrev Graph7 Graph8 Append-to-free0 Append-to-free1
Graph12 nth-list-update) +
apply(simp add:abbrev Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12
nth-list-update)
done

```

```

lemma interfree-Color-Target-Append:
  interfree-aux (Some Color-Target, {}, Some Append)
apply (unfold modules )
apply interfree-aux
apply(clarify, simp add:abbrev Append-to-free0) +
apply (force simp add: Append-to-free2)
apply(clarify, simp add:abbrev Append-to-free0) +
done

```

```

lemmas collector-mutator-interfree =
  interfree-Blacken-Roots-Redirect-Edge interfree-Blacken-Roots-Color-Target
  interfree-Propagate-Black-Redirect-Edge interfree-Propagate-Black-Color-Target
  interfree-Count-Redirect-Edge interfree-Count-Color-Target
  interfree-Append-Redirect-Edge interfree-Append-Color-Target
  interfree-Redirect-Edge-Blacken-Roots interfree-Color-Target-Blacken-Roots
  interfree-Redirect-Edge-Propagate-Black interfree-Color-Target-Propagate-Black
  interfree-Redirect-Edge-Count interfree-Color-Target-Count
  interfree-Redirect-Edge-Append interfree-Color-Target-Append

```

Interference freedom Collector-Mutator

```

lemma interfree-Collector-Mutator:
  interfree-aux (Some Collector, {}, Some Mutator)
apply(unfold Collector-def Mutator-def)
apply interfree-aux
apply(simp-all add:collector-mutator-interfree)
apply(unfold modules collector-defs Mut-init-def)
apply(tactic <TRYALL (interfree-aux-tac context)>)
— 32 subgoals left
apply(simp-all add:Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
— 20 subgoals left
apply(tactic<TRYALL (clarify-tac context)>)
apply(simp-all add:Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
apply(tactic <TRYALL (eresolve-tac context [disjE])>)
apply simp-all
apply(tactic <TRYALL(EVERY'[resolve-tac context [disjI2],
  resolve-tac context [subset-trans],
  eresolve-tac context @{thms Graph3},
  force-tac context,
  assume-tac context])>)
apply(tactic <TRYALL(EVERY'[resolve-tac context [disjI2],
  eresolve-tac context [subset-trans],

```

```

    resolve-tac context @{\thms Graph9},
    force-tac context])>))
apply(tactic < TRYALL (EVERY'[resolve-tac context [disjI1],
    eresolve-tac context @{\thms psubset-subset-trans},
    resolve-tac context @{\thms Graph9},
    force-tac context])>))
done

```

Interference freedom Mutator-Collector

```

lemma interfree-Mutator-Collector:
  interfree-aux (Some Mutator, {}, Some Collector)
apply(unfold Collector-def Mutator-def)
apply interfree-aux
apply(simp-all add:collector-mutator-interfree)
apply(unfold modules collector-defs Mut-init-def)
apply(tactic < TRYALL (interfree-aux-tac context)>))
— 64 subgoals left
apply(simp-all add:nth-list-update Invariants Append-to-free0)+
apply(tactic< TRYALL (clarify-tac context)>))
— 4 subgoals left
apply force
apply(simp add:Append-to-free2)
apply force
apply(simp add:Append-to-free2)
done

```

The Garbage Collection algorithm

In total there are 289 verification conditions.

```

lemma Gar-Coll:
  ||— { 'Proper  $\wedge$  'Mut-init  $\wedge$  'z }
  COBEGIN
    Collector
    {False}
  ||
    Mutator
    {False}
  COEND
  {False}
apply oghoare
apply(force simp add: Mutator-def Collector-def modules)
apply(rule Collector)
apply(rule Mutator)
apply(simp add:interfree-Collector-Mutator)
apply(simp add:interfree-Mutator-Collector)
apply force
done

```

end

2.3 The Multi-Mutator Case

theory *Mul-Gar-Coll* **imports** *Graph OG-Syntax* **begin**

The full theory takes aprox. 18 minutes.

record *mut* =
 Z :: *bool*
 R :: *nat*
 T :: *nat*

Declaration of variables:

record *mul-gar-coll-state* =
 M :: *nodes*
 E :: *edges*
 bc :: *nat set*
 obc :: *nat set*
 Ma :: *nodes*
 ind :: *nat*
 k :: *nat*
 q :: *nat*
 l :: *nat*
 Muts :: *mut list*

2.3.1 The Mutators

definition *Mul-mut-init* :: *mul-gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
 Mul-mut-init $\equiv$ « $\lambda n. n = \text{length } 'Muts \wedge (\forall i < n. R ('Muts!i) < \text{length } 'E$
 $\wedge T ('Muts!i) < \text{length } 'M)$ »

definition *Mul-Redirect-Edge* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *mul-gar-coll-state* *ann-com* **where**
 Mul-Redirect-Edge *j n* $\equiv$
 $\{ \{ 'Mul-mut-init\ n \wedge Z ('Muts!j) \}$
 $\langle IF\ T ('Muts!j) \in Reach\ 'E\ THEN$
 $'E := 'E[R ('Muts!j) := (fst ('E!R ('Muts!j))),\ T ('Muts!j)) \rangle FI.,$
 $'Muts := 'Muts[j := ('Muts!j)\ (Z := False)] \}$

definition *Mul-Color-Target* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *mul-gar-coll-state* *ann-com* **where**
 Mul-Color-Target *j n* $\equiv$
 $\{ \{ 'Mul-mut-init\ n \wedge \neg Z ('Muts!j) \}$
 $\langle 'M := 'M[T ('Muts!j) := Black],\ 'Muts := 'Muts[j := ('Muts!j)\ (Z := True)] \rangle \}$

definition *Mul-Mutator* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *mul-gar-coll-state* *ann-com* **where**
 Mul-Mutator *j n* $\equiv$
 $\{ \{ 'Mul-mut-init\ n \wedge Z ('Muts!j) \}$
 $WHILE\ True$
 $INV\ \{ \{ 'Mul-mut-init\ n \wedge Z ('Muts!j) \}$

DO *Mul-Redirect-Edge* $j\ n$;;
 Mul -*Color-Target* $j\ n$
 OD

lemmas *mul-mutator-defs* = *Mul-mut-init-def* *Mul-Redirect-Edge-def* *Mul-Color-Target-def*

Correctness of the proof outline of one mutator

lemma *Mul-Redirect-Edge*: $0 \leq j \wedge j < n \implies$
 $\vdash$ *Mul-Redirect-Edge* $j\ n$
 $pre(Mul-Color-Target\ j\ n)$
apply (*unfold mul-mutator-defs*)
apply *annhoare*
apply (*simp-all*)
apply *clarify*
apply (*simp add: nth-list-update*)
done

lemma *Mul-Color-Target*: $0 \leq j \wedge j < n \implies$
 $\vdash$ *Mul-Color-Target* $j\ n$
 $\{\text{' } Mul\text{-mut-init } n \wedge Z\ (\text{' } Muts!j)\}$
apply (*unfold mul-mutator-defs*)
apply *annhoare*
apply (*simp-all*)
apply *clarify*
apply (*simp add: nth-list-update*)
done

lemma *Mul-Mutator*: $0 \leq j \wedge j < n \implies$
 $\vdash$ *Mul-Mutator* $j\ n\ \{\text{False}\}$
apply (*unfold Mul-Mutator-def*)
apply *annhoare*
apply (*simp-all add: Mul-Redirect-Edge Mul-Color-Target*)
apply (*simp add: mul-mutator-defs Mul-Redirect-Edge-def*)
done

Interference freedom between mutators

lemma *Mul-interfree-Redirect-Edge-Redirect-Edge*:
 $\llbracket 0 \leq i; i < n; 0 \leq j; j < n; i \neq j \rrbracket \implies$
 $interfree\text{-aux}\ (Some\ (Mul-Redirect-Edge\ i\ n), \{\}, Some(Mul-Redirect-Edge\ j\ n))$
apply (*unfold mul-mutator-defs*)
apply *interfree-aux*
apply *safe*
apply (*simp-all add: nth-list-update*)
done

lemma *Mul-interfree-Redirect-Edge-Color-Target*:
 $\llbracket 0 \leq i; i < n; 0 \leq j; j < n; i \neq j \rrbracket \implies$
 $interfree\text{-aux}\ (Some(Mul-Redirect-Edge\ i\ n), \{\}, Some(Mul-Color-Target\ j\ n))$

```

apply (unfold mul-mutator-defs)
apply interfree-aux
apply safe
apply(simp-all add: nth-list-update)
done

```

```

lemma Mul-interfree-Color-Target-Redirect-Edge:
   $\llbracket 0 \leq i; i < n; 0 \leq j; j < n; i \neq j \rrbracket \implies$ 
  interfree-aux (Some(Mul-Color-Target i n), {}, Some(Mul-Redirect-Edge j n))
apply (unfold mul-mutator-defs)
apply interfree-aux
apply safe
apply(simp-all add: nth-list-update)
done

```

```

lemma Mul-interfree-Color-Target-Color-Target:
   $\llbracket 0 \leq i; i < n; 0 \leq j; j < n; i \neq j \rrbracket \implies$ 
  interfree-aux (Some(Mul-Color-Target i n), {}, Some(Mul-Color-Target j n))
apply (unfold mul-mutator-defs)
apply interfree-aux
apply safe
apply(simp-all add: nth-list-update)
done

```

```

lemmas mul-mutator-interfree =
  Mul-interfree-Redirect-Edge-Redirect-Edge Mul-interfree-Redirect-Edge-Color-Target
  Mul-interfree-Color-Target-Redirect-Edge Mul-interfree-Color-Target-Color-Target

```

```

lemma Mul-interfree-Mutator-Mutator:  $\llbracket i < n; j < n; i \neq j \rrbracket \implies$ 
  interfree-aux (Some (Mul-Mutator i n), {}, Some (Mul-Mutator j n))
apply(unfold Mul-Mutator-def)
apply(interfree-aux)
apply(simp-all add: mul-mutator-interfree)
apply(simp-all add: mul-mutator-defs)
apply(tactic <TRYALL (interfree-aux-tac context)>)
apply(tactic <ALLGOALS (clarify-tac context)>)
apply (simp-all add: nth-list-update)
done

```

Modular Parameterized Mutators

```

lemma Mul-Parameterized-Mutators:  $0 < n \implies$ 
   $\| - \{ \text{'Mul-mut-init } n \wedge (\forall i < n. Z (\text{'Muts! } i)) \}$ 
  COBEGIN
  SCHEME [0 ≤ j < n]
  Mul-Mutator j n
  {False}
  COEND
  {False}

```

```

apply oghoare
apply(force simp add:Mul-Mutator-def mul-mutator-defs nth-list-update)
apply(erule Mul-Mutator)
apply(simp add:Mul-interfree-Mutator-Mutator)
apply(force simp add:Mul-Mutator-def mul-mutator-defs nth-list-update)
done

```

2.3.2 The Collector

definition *Queue* :: *mul-gar-coll-state* $\Rightarrow$ *nat* **where**
Queue $\equiv$ « *length* (*filter* ($\lambda i. \neg Z\ i \wedge 'M!(T\ i) \neq Black$) '*Muts*) »

consts *M-init* :: *nodes*

definition *Proper-M-init* :: *mul-gar-coll-state* $\Rightarrow$ *bool* **where**
Proper-M-init $\equiv$ « *Blacks* *M-init*=*Roots* $\wedge$ *length* *M-init*=*length* '*M* »

definition *Mul-Proper* :: *mul-gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
Mul-Proper $\equiv$ « $\lambda n. Proper-Roots\ 'M \wedge Proper-Edges\ ('M, 'E) \wedge 'Proper-M-init$
 $\wedge n=length\ 'Muts$ »

definition *Safe* :: *mul-gar-coll-state* $\Rightarrow$ *bool* **where**
Safe $\equiv$ « *Reach* '*E* $\subseteq$ *Blacks* '*M* »

lemmas *mul-collector-defs* = *Proper-M-init-def* *Mul-Proper-def* *Safe-def*

Blackening Roots

definition *Mul-Blacken-Roots* :: *nat* $\Rightarrow$ *mul-gar-coll-state* *ann-com* **where**
Mul-Blacken-Roots *n* $\equiv$
 $\{ \{ 'Mul-Proper\ n \}$
 $\quad 'ind:=0;;$
 $\{ 'Mul-Proper\ n \wedge 'ind=0 \}$
 $\quad WHILE\ 'ind < length\ 'M$
 $\quad \quad INV\ \{ 'Mul-Proper\ n \wedge (\forall i < 'ind. i \in Roots \longrightarrow 'M!i=Black) \wedge 'ind \leq length$
 $\quad \quad 'M \}$
 $\quad \quad DO\ \{ 'Mul-Proper\ n \wedge (\forall i < 'ind. i \in Roots \longrightarrow 'M!i=Black) \wedge 'ind < length\ 'M \}$
 $\quad \quad \quad IF\ 'ind \in Roots\ THEN$
 $\quad \quad \quad \{ 'Mul-Proper\ n \wedge (\forall i < 'ind. i \in Roots \longrightarrow 'M!i=Black) \wedge 'ind < length\ 'M \wedge$
 $\quad \quad \quad 'ind \in Roots \}$
 $\quad \quad \quad 'M := 'M['ind := Black]\ FI;;$
 $\quad \quad \quad \{ 'Mul-Proper\ n \wedge (\forall i < 'ind+1. i \in Roots \longrightarrow 'M!i=Black) \wedge 'ind < length$
 $\quad \quad \quad 'M \}$
 $\quad \quad \quad 'ind := 'ind+1$
 $\quad \quad OD$

lemma *Mul-Blacken-Roots*:
 $\vdash Mul-Blacken-Roots\ n$
 $\{ 'Mul-Proper\ n \wedge Roots \subseteq Blacks\ 'M \}$
apply (*unfold* *Mul-Blacken-Roots-def*)

```

apply annhoare
apply(simp-all add:mul-collector-defs Graph-defs)
apply safe
apply(simp-all add:nth-list-update)
  apply (erule less-SucE)
  apply simp+
apply force
apply force
done

```

Propagating Black

definition *Mul-PBInv* :: *mul-gar-coll-state* $\Rightarrow$ *bool* **where**

$$\begin{aligned} \text{Mul-PBInv} \equiv & \ll \text{'Safe} \vee \text{'obc} \subseteq \text{Blacks 'M} \vee \text{'l} < \text{'Queue} \\ & \vee (\forall i < \text{'ind. } \neg \text{BtoW}(\text{'E!}i, \text{'M})) \wedge \text{'l} \leq \text{'Queue} \gg \end{aligned}$$

definition *Mul-Auxk* :: *mul-gar-coll-state* $\Rightarrow$ *bool* **where**

$$\text{Mul-Auxk} \equiv \ll \text{'l} < \text{'Queue} \vee \text{'M!}k \neq \text{Black} \vee \neg \text{BtoW}(\text{'E!} \text{'ind}, \text{'M}) \vee \text{'obc} \subseteq \text{Blacks 'M} \gg$$

definition *Mul-Propagate-Black* :: *nat* $\Rightarrow$ *mul-gar-coll-state ann-com* **where**

```

Mul-Propagate-Black n  $\equiv$ 
  { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
     $\wedge$  ( 'Safe  $\vee$  'l  $\leq$  'Queue  $\vee$  'obc  $\subseteq$  Blacks 'M ) }
    'ind := 0;;
  { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
     $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  Blacks 'M  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
     $\wedge$  ( 'Safe  $\vee$  'l  $\leq$  'Queue  $\vee$  'obc  $\subseteq$  Blacks 'M )  $\wedge$  'ind = 0 }
    WHILE 'ind < length 'E
    INV { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
       $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
       $\wedge$  'Mul-PBInv  $\wedge$  'ind  $\leq$  length 'E }
    DO { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
       $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
       $\wedge$  'Mul-PBInv  $\wedge$  'ind < length 'E }
    IF 'M!(fst ('E!'ind)) = Black THEN
      { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
         $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
         $\wedge$  'Mul-PBInv  $\wedge$  ('M!fst('E!'ind)) = Black  $\wedge$  'ind < length 'E }
        'k := snd('E!'ind);;
      { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
         $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
         $\wedge$  ( 'Safe  $\vee$  'obc  $\subseteq$  Blacks 'M  $\vee$  'l < 'Queue  $\vee$  ( $\forall i < \text{'ind. } \neg \text{BtoW}(\text{'E!}i, \text{'M}))$ 
           $\wedge$  'l  $\leq$  'Queue  $\wedge$  'Mul-Auxk )  $\wedge$  'k < length 'M  $\wedge$  'M!fst('E!'ind) = Black
           $\wedge$  'ind < length 'E }
        { 'M := 'M['k := Black], 'ind := 'ind + 1 }
      ELSE { { Mul-Prop n  $\wedge$  Roots  $\subseteq$  Blacks 'M
         $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
         $\wedge$  'Mul-PBInv  $\wedge$  'ind < length 'E }

```

$\langle IF \text{ 'M!}(\text{fst}(\text{'E!}'\text{ind})) \neq \text{Black THEN 'ind} := \text{'ind} + 1 \text{ FI} \rangle \text{ FI}$
OD

lemma *Mul-Propagate-Black:*
 $\vdash \text{Mul-Propagate-Black } n$
 $\{ \text{'Mul-Prop} n \wedge \text{Roots} \subseteq \text{Blacks 'M} \wedge \text{'obc} \subseteq \text{Blacks 'M} \wedge \text{'bc} \subseteq \text{Blacks 'M}$
 $\wedge (\text{'Safe} \vee \text{'obc} \subseteq \text{Blacks 'M} \vee \text{'l} < \text{'Queue} \wedge (\text{'l} \leq \text{'Queue} \vee \text{'obc} \subseteq \text{Blacks 'M})) \}$
apply(*unfold Mul-Propagate-Black-def*)
apply *annhoare*
apply(*simp-all add:Mul-PBInv-def mul-collector-defs Mul-Auxk-def Graph6 Graph7*
Graph8 Graph12 mul-collector-defs Queue-def)
— 8 subgoals left
apply *force*
apply *force*
apply *force*
apply(*force simp add:BtoW-def Graph-defs*)
— 4 subgoals left
apply *clarify*
apply(*simp add: mul-collector-defs Graph12 Graph6 Graph7 Graph8*)
apply(*disjE-tac*)
apply(*simp-all add:Graph12 Graph13*)
apply(*case-tac M x! k x=Black*)
apply(*simp add: Graph10*)
apply(*rule disjI2, rule disjI1, erule subset-psubset-trans, erule Graph11, force*)
apply(*case-tac M x! k x=Black*)
apply(*simp add: Graph10 BtoW-def*)
apply(*rule disjI2, clarify, erule less-SucE, force*)
apply(*case-tac M x!snd(E x! ind x)=Black*)
apply(*force*)
apply(*force*)
apply(*rule disjI2, rule disjI1, erule subset-psubset-trans, erule Graph11, force*)
— 2 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
apply (*simp-all*)
apply *clarify*
apply(*erule less-SucE*)
apply *force*
apply (*simp add:BtoW-def*)
— 1 subgoal left
apply *clarify*
apply *simp*
apply(*disjE-tac*)
apply (*simp-all*)
apply(*rule disjI1, rule Graph1*)
apply *simp-all*
done

Counting Black Nodes

definition *Mul-CountInv* :: *mul-gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**

Mul-CountInv $\equiv$ « $\lambda ind. \{i. i < ind \wedge 'Ma!i=Black\} \subseteq 'bc$ »

definition *Mul-Count* :: *nat* $\Rightarrow$ *mul-gar-coll-state ann-com* **where**

Mul-Count *n* $\equiv$

```

{ 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
   $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
   $\wedge$  length 'Ma=length 'M
   $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
   $\wedge$  'q<n+1  $\wedge$  'bc={}}
'ind:=0;;
{ 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
   $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
   $\wedge$  length 'Ma=length 'M
   $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
   $\wedge$  'q<n+1  $\wedge$  'bc={}  $\wedge$  'ind=0}
WHILE 'ind<length 'M
  INV { 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
     $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
     $\wedge$  length 'Ma=length 'M  $\wedge$  'Mul-CountInv 'ind
     $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
     $\wedge$  'q<n+1  $\wedge$  'ind $\leq$ length 'M}
  DO { 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
     $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
     $\wedge$  length 'Ma=length 'M  $\wedge$  'Mul-CountInv 'ind
     $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
     $\wedge$  'q<n+1  $\wedge$  'ind<length 'M}
  IF 'M!'ind=Black
  THEN { 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
     $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
     $\wedge$  length 'Ma=length 'M  $\wedge$  'Mul-CountInv 'ind
     $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
    'M))
     $\wedge$  'q<n+1  $\wedge$  'ind<length 'M  $\wedge$  'M!'ind=Black}
    'bc:=insert 'ind 'bc
  FI;;
{ 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
   $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
   $\wedge$  length 'Ma=length 'M  $\wedge$  'Mul-CountInv ('ind+1)
   $\wedge$  ( 'Safe  $\vee$  'obc $\subseteq$ Blacks 'Ma  $\vee$  'l<'q  $\wedge$  ( 'q $\leq$ 'Queue  $\vee$  'obc $\subseteq$ Blacks 'M) )
   $\wedge$  'q<n+1  $\wedge$  'ind<length 'M}
'ind:='ind+1
OD

```

lemma *Mul-Count*:

$\vdash$ *Mul-Count* *n*

```

{ 'Mul-Propert n  $\wedge$  Roots $\subseteq$ Blacks 'M
   $\wedge$  'obc $\subseteq$ Blacks 'Ma  $\wedge$  Blacks 'Ma $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M

```

$\wedge \text{length } 'Ma = \text{length } 'M \wedge \text{Blacks } 'Ma \subseteq 'bc$
 $\wedge ('Safe \vee 'obc \subseteq \text{Blacks } 'Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq \text{Blacks } 'M))$
 $\wedge 'q < n+1 \}$
apply (*unfold Mul-Count-def*)
apply *annhoare*
apply(*simp-all add:Mul-CountInv-def mul-collector-defs Mul-Auxk-def Graph6 Graph7 Graph8 Graph12 mul-collector-defs Queue-def*)
— 7 subgoals left
apply *force*
apply *force*
apply *force*
— 4 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
apply *simp-all*
apply(*simp add:Blacks-def*)
apply *clarify*
apply(*erule less-SucE*)
back
apply *force*
apply *force*
— 3 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
apply *simp-all*
apply *clarify*
apply(*erule less-SucE*)
back
apply *force*
apply *simp*
apply(*rotate-tac -1*)
apply (*force simp add:Blacks-def*)
— 2 subgoals left
apply *force*
— 1 subgoal left
apply *clarify*
apply(*drule-tac x = ind x in le-imp-less-or-eq*)
apply (*simp-all add:Blacks-def*)
done

Appending garbage nodes to the free list

axiomatization *Append-to-free* :: $\text{nat} \times \text{edges} \Rightarrow \text{edges}$

where

Append-to-free0: $\text{length } (\text{Append-to-free } (i, e)) = \text{length } e$ **and**

Append-to-free1: $\text{Proper-Edges } (m, e)$

$\implies \text{Proper-Edges } (m, \text{Append-to-free}(i, e))$ **and**

Append-to-free2: $i \notin \text{Reach } e$
 $\implies n \in \text{Reach } (\text{Append-to-free}(i, e)) = (n = i \vee n \in \text{Reach } e)$

definition *Mul-AppendInv* :: *mul-gar-coll-state* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
Mul-AppendInv $\equiv$ « $\lambda \text{ind. } (\forall i. \text{ind} \leq i \longrightarrow i < \text{length } 'M \longrightarrow i \in \text{Reach } 'E \longrightarrow 'M!i = \text{Black})$ »

definition *Mul-Append* :: *nat* $\Rightarrow$ *mul-gar-coll-state ann-com* **where**
Mul-Append $n \equiv$
 $\{ \text{'Mul-Prop} n \wedge \text{Roots} \subseteq \text{Blacks } 'M \wedge ' \text{Safe} \}$
 $' \text{ind} := 0;$
 $\{ \text{'Mul-Prop} n \wedge \text{Roots} \subseteq \text{Blacks } 'M \wedge ' \text{Safe} \wedge ' \text{ind} = 0 \}$
WHILE $' \text{ind} < \text{length } 'M$
 INV $\{ \text{'Mul-Prop} n \wedge ' \text{Mul-AppendInv } ' \text{ind} \wedge ' \text{ind} \leq \text{length } 'M \}$
 DO $\{ \text{'Mul-Prop} n \wedge ' \text{Mul-AppendInv } ' \text{ind} \wedge ' \text{ind} < \text{length } 'M \}$
 IF $'M! ' \text{ind} = \text{Black}$ **THEN**
 $\{ \text{'Mul-Prop} n \wedge ' \text{Mul-AppendInv } ' \text{ind} \wedge ' \text{ind} < \text{length } 'M \wedge 'M! ' \text{ind} = \text{Black} \}$
 $'M := 'M[' \text{ind} := \text{White}]$
 ELSE
 $\{ \text{'Mul-Prop} n \wedge ' \text{Mul-AppendInv } ' \text{ind} \wedge ' \text{ind} < \text{length } 'M \wedge ' \text{ind} \notin \text{Reach } 'E \}$
 $'E := \text{Append-to-free}(' \text{ind}, 'E)$
 FI;;
 $\{ \text{'Mul-Prop} n \wedge ' \text{Mul-AppendInv } (' \text{ind} + 1) \wedge ' \text{ind} < \text{length } 'M \}$
 $' \text{ind} := ' \text{ind} + 1$
 OD

lemma *Mul-Append*:
 $\vdash \text{Mul-Append } n$
 $\{ \text{'Mul-Prop} n \}$
apply(*unfold Mul-Append-def*)
apply *annhoare*
apply(*simp-all add: mul-collector-defs Mul-AppendInv-def*
 Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
apply(*force simp add:Blacks-def*)
apply(*force simp add:Blacks-def*)
apply(*force simp add:Blacks-def*)
apply(*force simp add:Graph-defs*)
apply *force*
apply(*force simp add:Append-to-free1 Append-to-free2*)
apply *force*
apply *force*
done

Collector

definition *Mul-Collector* :: *nat* $\Rightarrow$ *mul-gar-coll-state ann-com* **where**
Mul-Collector $n \equiv$
 $\{ \text{'Mul-Prop} n \}$

```

WHILE True INV { Mul-Propser n }
DO
  Mul-Blacken-Roots n ;;
  { Mul-Propser n ∧ Roots ⊆ Blacks 'M }
  'obc := { } ;;
  { Mul-Propser n ∧ Roots ⊆ Blacks 'M ∧ 'obc = { } }
  'bc := Roots ;;
  { Mul-Propser n ∧ Roots ⊆ Blacks 'M ∧ 'obc = { } ∧ 'bc = Roots }
  'l := 0 ;;
  { Mul-Propser n ∧ Roots ⊆ Blacks 'M ∧ 'obc = { } ∧ 'bc = Roots ∧ 'l = 0 }
  WHILE 'l < n + 1
    INV { Mul-Propser n ∧ Roots ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M ∧
      ('Safe ∨ ('l ≤ 'Queue ∨ 'bc ⊆ Blacks 'M) ∧ 'l < n + 1) }
    DO { Mul-Propser n ∧ Roots ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
      ∧ ('Safe ∨ 'l ≤ 'Queue ∨ 'bc ⊆ Blacks 'M) }
      'obc := 'bc ;;
      Mul-Propagate-Black n ;;
      { Mul-Propser n ∧ Roots ⊆ Blacks 'M
        ∧ 'obc ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
        ∧ ('Safe ∨ 'obc ⊆ Blacks 'M ∨ 'l < 'Queue
        ∧ ('l ≤ 'Queue ∨ 'obc ⊆ Blacks 'M)) }
      'bc := { } ;;
      { Mul-Propser n ∧ Roots ⊆ Blacks 'M
        ∧ 'obc ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
        ∧ ('Safe ∨ 'obc ⊆ Blacks 'M ∨ 'l < 'Queue
        ∧ ('l ≤ 'Queue ∨ 'obc ⊆ Blacks 'M)) ∧ 'bc = { } }
      { 'Ma := 'M,, 'q := 'Queue } ;;
      Mul-Count n ;;
      { Mul-Propser n ∧ Roots ⊆ Blacks 'M
        ∧ 'obc ⊆ Blacks 'Ma ∧ Blacks 'Ma ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
        ∧ length 'Ma = length 'M ∧ Blacks 'Ma ⊆ 'bc
        ∧ ('Safe ∨ 'obc ⊆ Blacks 'Ma ∨ 'l < 'q ∧ ('q ≤ 'Queue ∨ 'obc ⊆ Blacks 'M))
        ∧ 'q < n + 1 }
      IF 'obc = 'bc THEN
        { Mul-Propser n ∧ Roots ⊆ Blacks 'M
          ∧ 'obc ⊆ Blacks 'Ma ∧ Blacks 'Ma ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
          ∧ length 'Ma = length 'M ∧ Blacks 'Ma ⊆ 'bc
          ∧ ('Safe ∨ 'obc ⊆ Blacks 'Ma ∨ 'l < 'q ∧ ('q ≤ 'Queue ∨ 'obc ⊆ Blacks 'M))
          ∧ 'q < n + 1 ∧ 'obc = 'bc }
        'l := 'l + 1
      ELSE { Mul-Propser n ∧ Roots ⊆ Blacks 'M
        ∧ 'obc ⊆ Blacks 'Ma ∧ Blacks 'Ma ⊆ Blacks 'M ∧ 'bc ⊆ Blacks 'M
        ∧ length 'Ma = length 'M ∧ Blacks 'Ma ⊆ 'bc
        ∧ ('Safe ∨ 'obc ⊆ Blacks 'Ma ∨ 'l < 'q ∧ ('q ≤ 'Queue ∨ 'obc ⊆ Blacks 'M))
        ∧ 'q < n + 1 ∧ 'obc ≠ 'bc }
        'l := 0 FI
    OD ;;
  Mul-Append n
OD

```

lemmas *mul-modules* = *Mul-Redirect-Edge-def* *Mul-Color-Target-def*
Mul-Blacken-Roots-def *Mul-Propagate-Black-def*
Mul-Count-def *Mul-Append-def*

lemma *Mul-Collector*:
 $\vdash \text{Mul-Collector } n$
 $\{\!\{False\}\!\}$
apply(*unfold Mul-Collector-def*)
apply *annhoare*
apply(*simp-all only:pre.simps Mul-Blacken-Roots*
Mul-Propagate-Black Mul-Count Mul-Append)
apply(*simp-all add:mul-modules*)
apply(*simp-all add:mul-collector-defs Queue-def*)
apply *force*
apply *force*
apply *force*
apply (*force simp add: less-Suc-eq-le*)
apply *force*
apply (*force dest:subset-antisym*)
apply *force*
apply *force*
apply *force*
done

2.3.3 Interference Freedom

lemma *le-length-filter-update*[*rule-format*]:
 $\forall i. (\neg P (\text{list!}i) \vee P j) \wedge i < \text{length list}$
 $\longrightarrow \text{length}(\text{filter } P \text{ list}) \leq \text{length}(\text{filter } P (\text{list}[i:=j]))$
apply(*induct-tac list*)
apply(*simp*)
apply(*clarify*)
apply(*case-tac i*)
apply(*simp*)
apply(*simp*)
done

lemma *less-length-filter-update* [*rule-format*]:
 $\forall i. P j \wedge \neg(P (\text{list!}i)) \wedge i < \text{length list}$
 $\longrightarrow \text{length}(\text{filter } P \text{ list}) < \text{length}(\text{filter } P (\text{list}[i:=j]))$
apply(*induct-tac list*)
apply(*simp*)
apply(*clarify*)
apply(*case-tac i*)
apply(*simp*)
apply(*simp*)
done

lemma *Mul-interfree-Blacken-Roots-Redirect-Edge*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (Some(*Mul-Blacken-Roots* *n*), {}, Some(*Mul-Redirect-Edge* *j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply *safe*
apply(*simp-all add:Graph6 Graph9 Graph12 nth-list-update mul-mutator-defs mul-collector-defs*)
done

lemma *Mul-interfree-Redirect-Edge-Blacken-Roots*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (Some(*Mul-Redirect-Edge* *j n*), {}, Some (*Mul-Blacken-Roots* *n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply *safe*
apply(*simp-all add:mul-mutator-defs nth-list-update*)
done

lemma *Mul-interfree-Blacken-Roots-Color-Target*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (Some(*Mul-Blacken-Roots* *n*), {}, Some (*Mul-Color-Target* *j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply *safe*
apply(*simp-all add:mul-mutator-defs mul-collector-defs nth-list-update Graph7 Graph8 Graph9 Graph12*)
done

lemma *Mul-interfree-Color-Target-Blacken-Roots*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (Some(*Mul-Color-Target* *j n*), {}, Some (*Mul-Blacken-Roots* *n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply *safe*
apply(*simp-all add:mul-mutator-defs nth-list-update*)
done

lemma *Mul-interfree-Propagate-Black-Redirect-Edge*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (Some(*Mul-Propagate-Black* *n*), {}, Some (*Mul-Redirect-Edge* *j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply(*simp-all add:mul-mutator-defs mul-collector-defs Mul-PBInv-def nth-list-update Graph6*)
— 7 subgoals left
apply *clarify*
apply(*disjE-tac*)
apply(*simp-all add:Graph6*)
apply(*rule impI, rule disjI1, rule subset-trans, erule Graph3, simp, simp*)
apply(*rule conjI*)
apply(*rule impI, rule disjI2, rule disjI1, erule le-trans, force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*rule impI, rule disjI2, rule disjI1, erule le-trans, force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)

— 6 subgoals left
apply *clarify*
apply(*disjE-tac*)
 apply(*simp-all add: Graph6*)
 apply(*rule impI, rule disjI1, rule subset-trans, erule Graph3, simp, simp*)
apply(*rule conjI*)
 apply(*rule impI, rule disjI2, rule disjI1, erule le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*rule impI, rule disjI2, rule disjI1, erule le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
— 5 subgoals left
apply *clarify*
apply(*disjE-tac*)
 apply(*simp-all add: Graph6*)
 apply(*rule impI, rule disjI1, rule subset-trans, erule Graph3, simp, simp*)
apply(*rule conjI*)
 apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule less-le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule less-le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*erule conjE*)
apply(*case-tac M x!(T (Muts x!j))=Black*)
apply(*rule conjI*)
 apply(*rule impI, (rule disjI2)+, rule conjI*)
 apply *clarify*
 apply(*case-tac R (Muts x! j)=i*)
 apply (*force simp add: nth-list-update BtoW-def*)
 apply (*force simp add: nth-list-update*)
 apply(*erule le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
 apply(*rule impI, (rule disjI2)+, erule le-trans*)
 apply(*force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*rule conjI*)
 apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule le-less-trans*)
 apply(*force simp add: Queue-def less-Suc-eq-le less-length-filter-update*)
 apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule le-less-trans*)
 apply(*force simp add: Queue-def less-Suc-eq-le less-length-filter-update*)
— 4 subgoals left
apply *clarify*
apply(*disjE-tac*)
 apply(*simp-all add: Graph6*)
 apply(*rule impI, rule disjI1, rule subset-trans, erule Graph3, simp, simp*)
apply(*rule conjI*)
 apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule less-le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*rule impI, rule disjI2, rule disjI2, rule disjI1, erule less-le-trans, force simp add: Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*erule conjE*)
apply(*case-tac M x!(T (Muts x!j))=Black*)
apply(*rule conjI*)

```

apply(rule impI,(rule disjI2)+,rule conjI)
apply clarify
apply(case-tac R (Muts x! j)=i)
  apply (force simp add: nth-list-update BtoW-def)
  apply (force simp add: nth-list-update)
apply(erule le-trans,force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(rule impI,(rule disjI2)+, erule le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(rule conjI)
apply(rule impI,rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
apply(force simp add: Queue-def less-Suc-eq-le less-length-filter-update)
apply(rule impI,rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
apply(force simp add: Queue-def less-Suc-eq-le less-length-filter-update)
— 3 subgoals left
apply clarify
apply(disjE-tac)
  apply(simp-all add: Graph6)
  apply (rule impI)
  apply(rule conjI)
    apply(rule disjI1,rule subset-trans,erule Graph3,simp,simp)
    apply(case-tac R (Muts x ! j)= ind x)
    apply(simp add: nth-list-update)
    apply(simp add: nth-list-update)
    apply(case-tac R (Muts x ! j)= ind x)
    apply(simp add: nth-list-update)
    apply(simp add: nth-list-update)
apply(case-tac M x!(T (Muts x!j))=Black)
apply(rule conjI)
apply(rule impI)
apply(rule conjI)
  apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
  apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(case-tac R (Muts x ! j)= ind x)
  apply(simp add: nth-list-update)
  apply(simp add: nth-list-update)
apply(rule impI)
apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(rule conjI)
apply(rule impI)
apply(rule conjI)
  apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
  apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(case-tac R (Muts x ! j)= ind x)
  apply(simp add: nth-list-update)
  apply(simp add: nth-list-update)
apply(rule impI)
apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)

```

```

apply(erule conjE)
apply(rule conjI)
apply(case-tac M x!(T (Muts x!j))=Black)
  apply(rule impI,rule conjI,(rule disjI2)+,rule conjI)
    apply clarify
      apply(case-tac R (Muts x! j)=i)
        apply (force simp add: nth-list-update BtoW-def)
          apply (force simp add: nth-list-update)
            apply(erule le-trans,force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
              apply(case-tac R (Muts x ! j)= ind x)
                apply(simp add: nth-list-update)
              apply(simp add: nth-list-update)
            apply(rule impI,rule conjI)
              apply(rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
                apply(force simp add: Queue-def less-Suc-eq-le less-length-filter-update)
              apply(case-tac R (Muts x! j)=ind x)
                apply (force simp add: nth-list-update)
              apply (force simp add: nth-list-update)
            apply(rule impI, (rule disjI2)+, erule le-trans)
              apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
            — 2 subgoals left
          apply clarify
            apply(rule conjI)
              apply(disjE-tac)
                apply(simp-all add: Mul-Auxk-def Graph6)
                  apply (rule impI)
                    apply(rule conjI)
                      apply(rule disjI1,rule subset-trans,erule Graph3,simp,simp)
                        apply(case-tac R (Muts x ! j)= ind x)
                          apply(simp add: nth-list-update)
                        apply(simp add: nth-list-update)
                      apply(case-tac R (Muts x ! j)= ind x)
                        apply(simp add: nth-list-update)
                      apply(simp add: nth-list-update)
                    apply(case-tac M x!(T (Muts x!j))=Black)
                      apply(rule impI)
                        apply(rule conjI)
                          apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
                            apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
                          apply(case-tac R (Muts x ! j)= ind x)
                            apply(simp add: nth-list-update)
                          apply(simp add: nth-list-update)
                        apply(rule impI)
                          apply(rule conjI)
                            apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
                              apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
                            apply(case-tac R (Muts x ! j)= ind x)
                              apply(simp add: nth-list-update)
                            apply(simp add: nth-list-update)

```

```

apply(rule impI)
apply(rule conjI)
apply(erule conjE)+
apply(case-tac M x!(T (Muts x!j))=Black)
apply((rule disjI2)+,rule conjI)
apply clarify
apply(case-tac R (Muts x! j)=i)
apply (force simp add: nth-list-update BtoW-def)
apply (force simp add: nth-list-update)
apply(rule conjI)
apply(erule le-trans,force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(rule impI)
apply(case-tac R (Muts x ! j)= ind x)
apply(simp add: nth-list-update BtoW-def)
apply (simp add: nth-list-update)
apply(rule impI)
apply simp
apply(disjE-tac)
apply(rule disjI1, erule less-le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply force
apply(rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
apply(force simp add: Queue-def less-Suc-eq-le less-length-filter-update)
apply(case-tac R (Muts x ! j)= ind x)
apply(simp add: nth-list-update)
apply(simp add: nth-list-update)
apply(disjE-tac)
apply simp-all
apply(conjI-tac)
apply(rule impI)
apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(erule conjE)+
apply(rule impI,(rule disjI2)+,rule conjI)
apply(erule le-trans,force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply(rule impI)+
apply simp
apply(disjE-tac)
apply(rule disjI1, erule less-le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update)
apply force
— 1 subgoal left
apply clarify
apply(disjE-tac)
apply(simp-all add: Graph6)
apply(rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp)
apply(rule conjI)
apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule less-le-trans,force simp
add: Queue-def less-Suc-eq-le le-length-filter-update)

```

```

apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule less-le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
apply(erule conjE)
apply(case-tac M x!(T (Muts x!j))=Black)
apply(rule conjI)
  apply(rule impI,(rule disjI2)+,rule conjI)
    apply clarify
      apply(case-tac R (Muts x! j)=i)
        apply (force simp add: nth-list-update BtoW-def)
          apply (force simp add: nth-list-update)
            apply(erule le-trans,force simp add:Queue-def less-Suc-eq-le le-length-filter-update)
              apply(rule impI,(rule disjI2)+, erule le-trans)
                apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update)
                  apply(rule conjI)
                    apply(rule impI,rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
                      apply(force simp add:Queue-def less-Suc-eq-le less-length-filter-update)
                        apply(rule impI,rule disjI2,rule disjI2,rule disjI1, erule le-less-trans)
                          apply(force simp add:Queue-def less-Suc-eq-le less-length-filter-update)
                            done

```

```

lemma Mul-interfree-Redirect-Edge-Propagate-Black:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Redirect-Edge j n ),{\},Some (Mul-Propagate-Black n))
apply (unfold mul-modules)
apply interfree-aux
apply safe
apply(simp-all add:mul-mutator-defs nth-list-update)
done

```

```

lemma Mul-interfree-Propagate-Black-Color-Target:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Propagate-Black n),{\},Some (Mul-Color-Target j n ))
apply (unfold mul-modules)
apply interfree-aux
apply(simp-all add: mul-collector-defs mul-mutator-defs)
  — 7 subgoals left
apply clarify
apply (simp add:Graph7 Graph8 Graph12)
apply(disjE-tac)
  apply(simp add:Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(rule disjI2,rule disjI1, erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply((rule disjI2)+,erule subset-psubset-trans, erule Graph11, simp)
apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
  — 6 subgoals left
apply clarify
apply (simp add:Graph7 Graph8 Graph12)
apply(disjE-tac)
  apply(simp add:Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)

```

```

    apply(rule disjI2,rule disjI1, erule le-trans)
    apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
    apply((rule disjI2)+,erule subset-psubset-trans, erule Graph11, simp)
    apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
    — 5 subgoals left
    apply clarify
    apply (simp add:mul-collector-defs Mul-PBInv-def Graph7 Graph8 Graph12)
    apply(disjE-tac)
      apply(simp add:Graph7 Graph8 Graph12)
      apply(rule disjI2,rule disjI1, erule psubset-subset-trans,simp add:Graph9)
      apply(case-tac M x!(T (Muts x!j))=Black)
      apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
      apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
      apply(rule disjI2,rule disjI1,erule subset-psubset-trans, erule Graph11, simp)
      apply(erule conjE)
      apply(case-tac M x!(T (Muts x!j))=Black)
      apply((rule disjI2)+)
      apply (rule conjI)
      apply(simp add:Graph10)
      apply(erule le-trans)
      apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
      apply(rule disjI2,rule disjI1,erule subset-psubset-trans, erule Graph11, simp)
    — 4 subgoals left
    apply clarify
    apply (simp add:mul-collector-defs Mul-PBInv-def Graph7 Graph8 Graph12)
    apply(disjE-tac)
      apply(simp add:Graph7 Graph8 Graph12)
      apply(rule disjI2,rule disjI1, erule psubset-subset-trans,simp add:Graph9)
      apply(case-tac M x!(T (Muts x!j))=Black)
      apply(rule disjI2,rule disjI2,rule disjI1, erule less-le-trans)
      apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
      apply(rule disjI2,rule disjI1,erule subset-psubset-trans, erule Graph11, simp)
      apply(erule conjE)
      apply(case-tac M x!(T (Muts x!j))=Black)
      apply((rule disjI2)+)
      apply (rule conjI)
      apply(simp add:Graph10)
      apply(erule le-trans)
      apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
      apply(rule disjI2,rule disjI1,erule subset-psubset-trans, erule Graph11, simp)
    — 3 subgoals left
    apply clarify
    apply (simp add:mul-collector-defs Mul-PBInv-def Graph7 Graph8 Graph12)
    apply(case-tac M x!(T (Muts x!j))=Black)
    apply(simp add:Graph10)
    apply(disjE-tac)
      apply simp-all
      apply(rule disjI2, rule disjI2, rule disjI1,erule less-le-trans)
      apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)

```

```

  apply(erule conjE)
  apply((rule disjI2)+,erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply(rule conjI)
  apply(rule disjI2,rule disjI1, erule subset-psubset-trans,simp add:Graph11)
  apply (force simp add:nth-list-update)
  — 2 subgoals left
  apply clarify
  apply(simp add:Mul-Auxk-def Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(simp add:Graph10)
  apply(disjE-tac)
  apply simp-all
  apply(rule disjI2, rule disjI2, rule disjI1,erule less-le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply(erule conjE)+
  apply((rule disjI2)+,rule conjI, erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply((rule impI)+)
  apply simp
  apply(erule disjE)
  apply(rule disjI1, erule less-le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply force
  apply(rule conjI)
  apply(rule disjI2,rule disjI1, erule subset-psubset-trans,simp add:Graph11)
  apply (force simp add:nth-list-update)
  — 1 subgoal left
  apply clarify
  apply (simp add:mul-collector-defs Mul-PBInv-def Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(simp add:Graph10)
  apply(disjE-tac)
  apply simp-all
  apply(rule disjI2, rule disjI2, rule disjI1,erule less-le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply(erule conjE)
  apply((rule disjI2)+,erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply(rule disjI2,rule disjI1, erule subset-psubset-trans,simp add:Graph11)
done

lemma Mul-interfree-Color-Target-Propagate-Black:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Color-Target j n),{\},Some(Mul-Propagate-Black n ))
  apply (unfold mul-modules)
  apply interfree-aux
  apply safe
  apply(simp-all add:mul-mutator-defs nth-list-update)
done

```

lemma *Mul-interfree-Count-Redirect-Edge*: $\llbracket 0 \leq j; j < n \rrbracket \implies$
interfree-aux (*Some*(*Mul-Count* *n*), {}, *Some*(*Mul-Redirect-Edge* *j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
— 9 subgoals left
apply(*simp add:mul-mutator-defs mul-collector-defs Mul-CountInv-def Graph6*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6*)
 apply(*rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp*)
 apply(*simp add:Graph6*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6*)
 apply(*rule conjI*)
 apply(*rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)
 apply(*rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*simp add:Graph6*)
— 8 subgoals left
apply(*simp add:mul-mutator-defs nth-list-update*)
— 7 subgoals left
apply(*simp add:mul-mutator-defs mul-collector-defs*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6*)
 apply(*rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp*)
 apply(*simp add:Graph6*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6*)
 apply(*rule conjI*)
 apply(*rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)
 apply(*rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le le-length-filter-update*)
apply(*simp add:Graph6*)
— 6 subgoals left
apply(*simp add:mul-mutator-defs mul-collector-defs Mul-CountInv-def*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6 Queue-def*)
 apply(*rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp*)
 apply(*simp add:Graph6*)
apply *clarify*
apply *disjE-tac*
 apply(*simp add:Graph6*)

```

apply(rule conjI)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
apply(simp add:Graph6)
— 5 subgoals left
apply(simp add:mul-mutator-defs mul-collector-defs Mul-CountInv-def)
apply clarify
apply disjE-tac
  apply(simp add:Graph6)
  apply(rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp)
  apply(simp add:Graph6)
apply clarify
apply disjE-tac
  apply(simp add:Graph6)
  apply(rule conjI)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
apply(simp add:Graph6)
— 4 subgoals left
apply(simp add:mul-mutator-defs mul-collector-defs Mul-CountInv-def)
apply clarify
apply disjE-tac
  apply(simp add:Graph6)
  apply(rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp)
  apply(simp add:Graph6)
apply clarify
apply disjE-tac
  apply(simp add:Graph6)
  apply(rule conjI)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
  apply(rule impI,rule disjI2,rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def
less-Suc-eq-le le-length-filter-update)
apply(simp add:Graph6)
— 3 subgoals left
apply(simp add:mul-mutator-defs nth-list-update)
— 2 subgoals left
apply(simp add:mul-mutator-defs mul-collector-defs Mul-CountInv-def)
apply clarify
apply disjE-tac
  apply(simp add:Graph6)
  apply(rule impI,rule disjI1,rule subset-trans,erule Graph3,simp,simp)
  apply(simp add:Graph6)
apply clarify
apply disjE-tac

```

```

apply(simp add: Graph6)
apply(rule conjI)
apply(rule impI, rule disjI2, rule disjI2, rule disjI1, erule le-trans, force simp add: Queue-def
less-Suc-eq-le le-length-filter-update)
apply(rule impI, rule disjI2, rule disjI2, rule disjI1, erule le-trans, force simp add: Queue-def
less-Suc-eq-le le-length-filter-update)
apply(simp add: Graph6)
— 1 subgoal left
apply(simp add: mul-mutator-defs nth-list-update)
done

```

```

lemma Mul-interfree-Redirect-Edge-Count:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Redirect-Edge j n), {}, Some(Mul-Count n ))
apply (unfold mul-modules)
apply interfree-aux
apply safe
apply(simp-all add: mul-mutator-defs nth-list-update)
done

```

```

lemma Mul-interfree-Count-Color-Target:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Count n ), {}, Some(Mul-Color-Target j n ))
apply (unfold mul-modules)
apply interfree-aux
apply(simp-all add: mul-collector-defs mul-mutator-defs Mul-CountInv-def)
— 6 subgoals left
apply clarify
apply disjE-tac
apply (simp add: Graph7 Graph8 Graph12)
apply (simp add: Graph7 Graph8 Graph12)
apply clarify
apply disjE-tac
apply (simp add: Graph7 Graph8 Graph12)
apply(case-tac M x!(T (Muts x!j))=Black)
apply(rule disjI2, rule disjI2, rule disjI1, erule le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update Graph10)
apply((rule disjI2)+, (erule subset-psubset-trans)+, simp add: Graph11)
apply (simp add: Graph7 Graph8 Graph12)
apply((rule disjI2)+, erule psubset-subset-trans, simp add: Graph9)
— 5 subgoals left
apply clarify
apply disjE-tac
apply (simp add: Graph7 Graph8 Graph12)
apply (simp add: Graph7 Graph8 Graph12)
apply clarify
apply disjE-tac
apply (simp add: Graph7 Graph8 Graph12)
apply(case-tac M x!(T (Muts x!j))=Black)
apply(rule disjI2, rule disjI2, rule disjI1, erule le-trans)
apply(force simp add: Queue-def less-Suc-eq-le le-length-filter-update Graph10)

```

```

  apply((rule disjI2)+,(erule subset-psubset-trans)+, simp add: Graph11)
apply (simp add: Graph7 Graph8 Graph12)
apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
— 4 subgoals left
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply (simp add: Graph7 Graph8 Graph12)
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(rule disjI2,rule disjI2, rule disjI1, erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply((rule disjI2)+,(erule subset-psubset-trans)+, simp add: Graph11)
apply (simp add: Graph7 Graph8 Graph12)
apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
— 3 subgoals left
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply (simp add: Graph7 Graph8 Graph12)
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(rule disjI2,rule disjI2, rule disjI1, erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply((rule disjI2)+,(erule subset-psubset-trans)+, simp add: Graph11)
apply (simp add: Graph7 Graph8 Graph12)
apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
— 2 subgoals left
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12 nth-list-update)
  apply (simp add: Graph7 Graph8 Graph12 nth-list-update)
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply(rule conjI)
  apply(case-tac M x!(T (Muts x!j))=Black)
  apply(rule disjI2,rule disjI2, rule disjI1, erule le-trans)
  apply(force simp add:Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply((rule disjI2)+,(erule subset-psubset-trans)+, simp add: Graph11)
  apply (simp add: nth-list-update)
apply (simp add: Graph7 Graph8 Graph12)
apply(rule conjI)
  apply((rule disjI2)+,erule psubset-subset-trans, simp add: Graph9)
apply (simp add: nth-list-update)

```

— 1 subgoal left

```

apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply (simp add: Graph7 Graph8 Graph12)
apply clarify
apply disjE-tac
  apply (simp add: Graph7 Graph8 Graph12)
  apply (case-tac M x!(T (Muts x!j))=Black)
  apply (rule disjI2, rule disjI2, rule disjI1, erule le-trans)
  apply (force simp add: Queue-def less-Suc-eq-le le-length-filter-update Graph10)
  apply ((rule disjI2)+, (erule subset-psubset-trans)+, simp add: Graph11)
apply (simp add: Graph7 Graph8 Graph12)
apply ((rule disjI2)+, erule psubset-subset-trans, simp add: Graph9)
done

```

```

lemma Mul-interfree-Color-Target-Count:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Color-Target j n), {}, Some(Mul-Count n ))
apply (unfold mul-modules)
apply interfree-aux
apply safe
apply (simp-all add: mul-mutator-defs nth-list-update)
done

```

```

lemma Mul-interfree-Append-Redirect-Edge:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Append n), {}, Some(Mul-Redirect-Edge j n))
apply (unfold mul-modules)
apply interfree-aux
apply (tactic  $\langle \text{ALLGOALS (clarify-tac context)} \rangle$ )
apply (simp-all add: Graph6 Append-to-free0 Append-to-free1 mul-collector-defs mul-mutator-defs
  Mul-AppendInv-def)
apply (erule-tac x=j in alle, force dest: Graph3) +
done

```

```

lemma Mul-interfree-Redirect-Edge-Append:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Redirect-Edge j n), {}, Some(Mul-Append n))
apply (unfold mul-modules)
apply interfree-aux
apply (tactic  $\langle \text{ALLGOALS (clarify-tac context)} \rangle$ )
apply (simp-all add: mul-collector-defs Append-to-free0 Mul-AppendInv-def mul-mutator-defs
  nth-list-update)
done

```

```

lemma Mul-interfree-Append-Color-Target:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some(Mul-Append n), {}, Some(Mul-Color-Target j n))
apply (unfold mul-modules)
apply interfree-aux
apply (tactic  $\langle \text{ALLGOALS (clarify-tac context)} \rangle$ )
apply (simp-all add: mul-mutator-defs mul-collector-defs Mul-AppendInv-def Graph7)

```

```

Graph8 Append-to-free0 Append-to-free1
      Graph12 nth-list-update)
done

lemma Mul-interfree-Color-Target-Append:  $\llbracket 0 \leq j; j < n \rrbracket \implies$ 
  interfree-aux (Some (Mul-Color-Target j n), {}, Some (Mul-Append n))
apply (unfold mul-modules)
apply interfree-aux
apply (tactic <ALLGOALS (clarify-tac context)>)
apply (simp-all add: mul-mutator-defs nth-list-update)
apply (simp add: Mul-AppendInv-def Append-to-free0)
done

```

Interference freedom Collector-Mutator

```

lemmas mul-collector-mutator-interfree =
  Mul-interfree-Blacken-Roots-Redirect-Edge Mul-interfree-Blacken-Roots-Color-Target
  Mul-interfree-Propagate-Black-Redirect-Edge Mul-interfree-Propagate-Black-Color-Target
  Mul-interfree-Count-Redirect-Edge Mul-interfree-Count-Color-Target
  Mul-interfree-Append-Redirect-Edge Mul-interfree-Append-Color-Target
  Mul-interfree-Redirect-Edge-Blacken-Roots Mul-interfree-Color-Target-Blacken-Roots
  Mul-interfree-Redirect-Edge-Propagate-Black Mul-interfree-Color-Target-Propagate-Black
  Mul-interfree-Redirect-Edge-Count Mul-interfree-Color-Target-Count
  Mul-interfree-Redirect-Edge-Append Mul-interfree-Color-Target-Append

lemma Mul-interfree-Collector-Mutator:  $j < n \implies$ 
  interfree-aux (Some (Mul-Collector n), {}, Some (Mul-Mutator j n))
apply (unfold Mul-Collector-def Mul-Mutator-def)
apply interfree-aux
apply (simp-all add: mul-collector-mutator-interfree)
apply (unfold mul-modules mul-collector-defs mul-mutator-defs)
apply (tactic <TRYALL (interfree-aux-tac context)>)
— 42 subgoals left
apply (clarify, simp add: Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1
  Graph12)+
— 24 subgoals left
apply (simp-all add: Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
— 14 subgoals left
apply (tactic <TRYALL (clarify-tac context)>)
apply (simp-all add: Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
apply (tactic <TRYALL (resolve-tac context [conjI])>)
apply (tactic <TRYALL (resolve-tac context [impI])>)
apply (tactic <TRYALL (eresolve-tac context [disjE])>)
apply (tactic <TRYALL (eresolve-tac context [conjE])>)
apply (tactic <TRYALL (eresolve-tac context [disjE])>)
apply (tactic <TRYALL (eresolve-tac context [disjE])>)
— 72 subgoals left
apply (simp-all add: Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
— 35 subgoals left

```

```

apply(tactic ‹TRYALL(EVERY'[resolve-tac context [disjI1],
  resolve-tac context [subset-trans],
  eresolve-tac context @{thms Graph3},
  force-tac context,
  assume-tac context])›)
— 28 subgoals left
apply(tactic ‹TRYALL (eresolve-tac context [conjE])›)
apply(tactic ‹TRYALL (eresolve-tac context [disjE])›)
— 34 subgoals left
apply(rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le
le-length-filter-update)
apply(rule disjI2,rule disjI1,erule le-trans,force simp add:Queue-def less-Suc-eq-le
le-length-filter-update)
apply(case-tac [!] M x!(T (Muts x ! j))=Black)
apply(simp-all add:Graph10)
— 47 subgoals left
apply(tactic ‹TRYALL(EVERY'[REPEAT o resolve-tac context [disjI2],
  eresolve-tac context @{thms subset-psubset-trans},
  eresolve-tac context @{thms Graph11},
  force-tac context])›)
— 41 subgoals left
apply(tactic ‹TRYALL(EVERY'[resolve-tac context [disjI2],
  resolve-tac context [disjI1],
  eresolve-tac context @{thms le-trans},
  force-tac (context addsimps @{thms Queue-def less-Suc-eq-le le-length-filter-update})])›)
— 35 subgoals left
apply(tactic ‹TRYALL(EVERY'[resolve-tac context [disjI2],
  resolve-tac context [disjI1],
  eresolve-tac context @{thms psubset-subset-trans},
  resolve-tac context @{thms Graph9},
  force-tac context])›)
— 31 subgoals left
apply(tactic ‹TRYALL(EVERY'[resolve-tac context [disjI2],
  resolve-tac context [disjI1],
  eresolve-tac context @{thms subset-psubset-trans},
  eresolve-tac context @{thms Graph11},
  force-tac context])›)
— 29 subgoals left
apply(tactic ‹TRYALL(EVERY'[REPEAT o resolve-tac context [disjI2],
  eresolve-tac context @{thms subset-psubset-trans},
  eresolve-tac context @{thms subset-psubset-trans},
  eresolve-tac context @{thms Graph11},
  force-tac context])›)
— 25 subgoals left
apply(tactic ‹TRYALL(EVERY'[resolve-tac context [disjI2],
  resolve-tac context [disjI2],
  resolve-tac context [disjI1],
  eresolve-tac context @{thms le-trans},
  force-tac (context addsimps @{thms Queue-def less-Suc-eq-le le-length-filter-update})])›)

```

— 10 subgoals left
apply(rule *disjI2*, rule *disjI2*, rule *conjI*, erule *less-le-trans*, force simp add: *Queue-def less-Suc-eq-le le-length-filter-update*, rule *disjI1*, rule *less-imp-le*, erule *less-le-trans*, force simp add: *Queue-def less-Suc-eq-le le-length-filter-update*) +
done

Interference freedom Mutator-Collector

lemma *Mul-interfree-Mutator-Collector*: $j < n \implies$
interfree-aux (Some (Mul-Mutator j n), {}, Some (Mul-Collector n))
apply(unfold *Mul-Collector-def* *Mul-Mutator-def*)
apply *interfree-aux*
apply(simp-all add: *mul-collector-mutator-interfree*)
apply(unfold *mul-modules mul-collector-defs mul-mutator-defs*)
apply(tactic ‹*TRYALL* (*interfree-aux-tac context*)›)
— 76 subgoals left
apply (*clarsimp simp add: nth-list-update*) +
— 56 subgoals left
apply (*clarsimp simp add: Mul-AppendInv-def Append-to-free0 nth-list-update*) +
done

The Multi-Mutator Garbage Collection Algorithm

The total number of verification conditions is 328

lemma *Mul-Gar-Coll*:
 $\| - \{ \text{'Mul-} Proper\ n \wedge \text{'Mul-mut-init}\ n \wedge (\forall i < n. Z(\text{'Muts!}i)) \}$
COBEGIN
Mul-Collector n
 $\{ \text{False} \}$
 $\|$
SCHEME $[0 \leq j < n]$
Mul-Mutator j n
 $\{ \text{False} \}$
COEND
 $\{ \text{False} \}$
apply *oghore*
— Strengthening the precondition
apply(rule *Int-greatest*)
apply (*case-tac* n)
apply(force simp add: *Mul-Collector-def mul-mutator-defs mul-collector-defs nth-append*)
apply(simp add: *Mul-Mutator-def mul-collector-defs mul-mutator-defs nth-append*)
apply *force*
apply *clarify*
apply(*case-tac* i)
apply(simp add: *Mul-Collector-def mul-mutator-defs mul-collector-defs nth-append*)
apply(simp add: *Mul-Mutator-def mul-mutator-defs mul-collector-defs nth-append*
nth-map-upt)
— Collector
apply(rule *Mul-Collector*)

```

— Mutator
apply(erule Mul-Mutator)
— Interference freedom
apply(simp add:Mul-interfree-Collector-Mutator)
apply(simp add:Mul-interfree-Mutator-Collector)
apply(simp add:Mul-interfree-Mutator-Mutator)
— Weakening of the postcondition
apply(case-tac n)
  apply(simp add:Mul-Collector-def mul-mutator-defs mul-collector-defs nth-append)
apply(simp add:Mul-Mutator-def mul-mutator-defs mul-collector-defs nth-append)
done

end

```

Chapter 3

The Rely-Guarantee Method

3.1 Abstract Syntax

theory *RG-Com* **imports** *Main* **begin**

Semantics of assertions and boolean expressions (*bexp*) as sets of states.
Syntax of commands *com* and parallel commands *par-com*.

type-synonym *'a bexp* = *'a set*

datatype *'a com* =
 Basic 'a $\Rightarrow$ 'a
 | *Seq 'a com 'a com*
 | *Cond 'a bexp 'a com 'a com*
 | *While 'a bexp 'a com*
 | *Await 'a bexp 'a com*

type-synonym *'a par-com* = *'a com option list*

end

3.2 Operational Semantics

theory *RG-Tran*
imports *RG-Com*
begin

3.2.1 Semantics of Component Programs

Environment transitions

type-synonym *'a conf* = *(('a com) option) $\times$ 'a*

inductive-set

$etran :: ('a\ conf \times 'a\ conf)\ set$
 and $etran' :: 'a\ conf \Rightarrow 'a\ conf \Rightarrow bool$ ($- \dashv e \rightarrow - [81,81] 80$)

where

$P -e\rightarrow Q \equiv (P, Q) \in etran$
 $| Env: (P, s) -e\rightarrow (P, t)$

lemma *etranE*: $c -e\rightarrow c' \implies (\bigwedge P s t. c = (P, s) \implies c' = (P, t) \implies Q) \implies Q$
by (*induct c, induct c', erule etran.cases, blast*)

Component transitions

inductive-set

ctran :: ('a conf $\times$ 'a conf) set
and *ctran'* :: 'a conf $\Rightarrow$ 'a conf $\Rightarrow$ bool ($- -c\rightarrow - [81,81] 80$)
and *ctrans* :: 'a conf $\Rightarrow$ 'a conf $\Rightarrow$ bool ($- -c*\rightarrow - [81,81] 80$)

where

$P -c\rightarrow Q \equiv (P, Q) \in ctran$
 $| P -c*\rightarrow Q \equiv (P, Q) \in ctran^*$

$| Basic: (Some(Basic f), s) -c\rightarrow (None, f s)$

$| Seq1: (Some P0, s) -c\rightarrow (None, t) \implies (Some(Seq P0 P1), s) -c\rightarrow (Some P1, t)$

$| Seq2: (Some P0, s) -c\rightarrow (Some P2, t) \implies (Some(Seq P0 P1), s) -c\rightarrow (Some(Seq P2 P1), t)$

$| CondT: s \in b \implies (Some(Cond b P1 P2), s) -c\rightarrow (Some P1, s)$

$| CondF: s \notin b \implies (Some(Cond b P1 P2), s) -c\rightarrow (Some P2, s)$

$| WhileF: s \notin b \implies (Some(While b P), s) -c\rightarrow (None, s)$

$| WhileT: s \in b \implies (Some(While b P), s) -c\rightarrow (Some(Seq P (While b P)), s)$

$| Await: \llbracket s \in b; (Some P, s) -c*\rightarrow (None, t) \rrbracket \implies (Some(Await b P), s) -c\rightarrow (None, t)$

monos *rtrancl-mono*

3.2.2 Semantics of Parallel Programs

type-synonym 'a par-conf = ('a par-com) $\times$ 'a

inductive-set

par-etran :: ('a par-conf $\times$ 'a par-conf) set
and *par-etran'* :: ['a par-conf, 'a par-conf] $\Rightarrow$ bool ($- -pe\rightarrow - [81,81] 80$)

where

$P -pe\rightarrow Q \equiv (P, Q) \in par-etran$
 $| ParEnv: (Ps, s) -pe\rightarrow (Ps, t)$

inductive-set

par-ctran :: ('a par-conf $\times$ 'a par-conf) set
and *par-ctran'* :: ['a par-conf, 'a par-conf] $\Rightarrow$ bool ($- -pc\rightarrow - [81,81] 80$)

where

$P -pc\rightarrow Q \equiv (P, Q) \in \text{par-ctran}$
 $| \text{ParComp}: \llbracket i < \text{length } Ps; (Ps!i, s) -c\rightarrow (r, t) \rrbracket \Longrightarrow (Ps, s) -pc\rightarrow (Ps[i:=r], t)$

lemma $\text{par-ctranE}: c -pc\rightarrow c' \Longrightarrow$

$(\bigwedge i \text{ Ps } s \text{ r } t. c = (Ps, s) \Longrightarrow c' = (Ps[i := r], t) \Longrightarrow i < \text{length } Ps \Longrightarrow$
 $(Ps ! i, s) -c\rightarrow (r, t) \Longrightarrow P) \Longrightarrow P$

by ($\text{induct } c, \text{induct } c', \text{erule } \text{par-ctran.cases}, \text{blast}$)

3.2.3 Computations

Sequential computations

type-synonym $'a \text{ confs} = 'a \text{ conf list}$

inductive-set $\text{cptn} :: 'a \text{ confs set}$

where

$\text{CptnOne}: [(P, s)] \in \text{cptn}$
 $| \text{CptnEnv}: (P, t) \# xs \in \text{cptn} \Longrightarrow (P, s) \# (P, t) \# xs \in \text{cptn}$
 $| \text{CptnComp}: \llbracket (P, s) -c\rightarrow (Q, t); (Q, t) \# xs \in \text{cptn} \rrbracket \Longrightarrow (P, s) \# (Q, t) \# xs \in \text{cptn}$

definition $\text{cp} :: ('a \text{ com}) \text{ option} \Rightarrow 'a \Rightarrow ('a \text{ confs}) \text{ set}$ **where**

$\text{cp } P \text{ s} \equiv \{l. !!0=(P, s) \wedge l \in \text{cptn}\}$

Parallel computations

type-synonym $'a \text{ par-confs} = 'a \text{ par-conf list}$

inductive-set $\text{par-cptn} :: 'a \text{ par-confs set}$

where

$\text{ParCptnOne}: [(P, s)] \in \text{par-cptn}$
 $| \text{ParCptnEnv}: (P, t) \# xs \in \text{par-cptn} \Longrightarrow (P, s) \# (P, t) \# xs \in \text{par-cptn}$
 $| \text{ParCptnComp}: \llbracket (P, s) -pc\rightarrow (Q, t); (Q, t) \# xs \in \text{par-cptn} \rrbracket \Longrightarrow (P, s) \# (Q, t) \# xs \in \text{par-cptn}$

definition $\text{par-cp} :: 'a \text{ par-com} \Rightarrow 'a \Rightarrow ('a \text{ par-confs}) \text{ set}$ **where**

$\text{par-cp } P \text{ s} \equiv \{l. !!0=(P, s) \wedge l \in \text{par-cptn}\}$

3.2.4 Modular Definition of Computation

definition $\text{lift} :: 'a \text{ com} \Rightarrow 'a \text{ conf} \Rightarrow 'a \text{ conf}$ **where**

$\text{lift } Q \equiv \lambda(P, s). (\text{if } P = \text{None then } (\text{Some } Q, s) \text{ else } (\text{Some}(\text{Seq } (\text{the } P) \text{ } Q), s))$

inductive-set $\text{cptn-mod} :: ('a \text{ confs}) \text{ set}$

where

$\text{CptnModOne}: [(P, s)] \in \text{cptn-mod}$
 $| \text{CptnModEnv}: (P, t) \# xs \in \text{cptn-mod} \Longrightarrow (P, s) \# (P, t) \# xs \in \text{cptn-mod}$
 $| \text{CptnModNone}: \llbracket (\text{Some } P, s) -c\rightarrow (\text{None}, t); (\text{None}, t) \# xs \in \text{cptn-mod} \rrbracket \Longrightarrow$
 $(\text{Some } P, s) \# (\text{None}, t) \# xs \in \text{cptn-mod}$

$\mid \text{CptnModCondT}: \llbracket (\text{Some } P0, s) \# ys \in \text{cptn-mod}; s \in b \rrbracket \implies (\text{Some}(\text{Cond } b \ P0 \ P1), s) \# (\text{Some } P0, s) \# ys \in \text{cptn-mod}$
 $\mid \text{CptnModCondF}: \llbracket (\text{Some } P1, s) \# ys \in \text{cptn-mod}; s \notin b \rrbracket \implies (\text{Some}(\text{Cond } b \ P0 \ P1), s) \# (\text{Some } P1, s) \# ys \in \text{cptn-mod}$
 $\mid \text{CptnModSeq1}: \llbracket (\text{Some } P0, s) \# xs \in \text{cptn-mod}; zs = \text{map } (\text{lift } P1) \ xs \rrbracket \implies (\text{Some}(\text{Seq } P0 \ P1), s) \# zs \in \text{cptn-mod}$
 $\mid \text{CptnModSeq2}: \llbracket (\text{Some } P0, s) \# xs \in \text{cptn-mod}; \text{fst}(\text{last } ((\text{Some } P0, s) \# xs)) = \text{None}; (\text{Some } P1, \text{snd}(\text{last } ((\text{Some } P0, s) \# xs))) \# ys \in \text{cptn-mod}; zs = (\text{map } (\text{lift } P1) \ xs) @ ys \rrbracket \implies (\text{Some}(\text{Seq } P0 \ P1), s) \# zs \in \text{cptn-mod}$
 $\mid \text{CptnModWhile1}: \llbracket (\text{Some } P, s) \# xs \in \text{cptn-mod}; s \in b; zs = \text{map } (\text{lift } (\text{While } b \ P)) \ xs \rrbracket \implies (\text{Some}(\text{While } b \ P), s) \# (\text{Some}(\text{Seq } P \ (\text{While } b \ P)), s) \# zs \in \text{cptn-mod}$
 $\mid \text{CptnModWhile2}: \llbracket (\text{Some } P, s) \# xs \in \text{cptn-mod}; \text{fst}(\text{last } ((\text{Some } P, s) \# xs)) = \text{None}; s \in b; zs = (\text{map } (\text{lift } (\text{While } b \ P)) \ xs) @ ys; (\text{Some}(\text{While } b \ P), \text{snd}(\text{last } ((\text{Some } P, s) \# xs))) \# ys \in \text{cptn-mod} \rrbracket \implies (\text{Some}(\text{While } b \ P), s) \# (\text{Some}(\text{Seq } P \ (\text{While } b \ P)), s) \# zs \in \text{cptn-mod}$

3.2.5 Equivalence of Both Definitions.

lemma *last-length*: $((a \# xs)!(\text{length } xs)) = \text{last } (a \# xs)$
by (*induct xs*) *auto*

lemma *div-seq [rule-format]*: $\text{list} \in \text{cptn-mod} \implies$
 $(\forall s \ P \ Q \ zs. \text{list} = (\text{Some } (\text{Seq } P \ Q), s) \# zs \longrightarrow$
 $(\exists xs. (\text{Some } P, s) \# xs \in \text{cptn-mod} \wedge (zs = (\text{map } (\text{lift } Q) \ xs) \vee$
 $(\text{fst}(((\text{Some } P, s) \# xs)!\text{length } xs)) = \text{None} \wedge$
 $(\exists ys. (\text{Some } Q, \text{snd}(((\text{Some } P, s) \# xs)!\text{length } xs))) \# ys \in \text{cptn-mod}$
 $\wedge zs = (\text{map } (\text{lift } (Q)) \ xs) @ ys))))$

apply (*erule cptn-mod.induct*)

apply *simp-all*

apply *clarify*

apply (*force intro: CptnModOne*)

apply *clarify*

apply (*erule-tac x=Pa in allE*)

apply (*erule-tac x=Q in allE*)

apply *simp*

apply *clarify*

apply (*erule disjE*)

apply (*rule-tac x=(Some Pa,t)#xsa in exI*)

apply (*rule conjI*)

apply *clarify*

apply (*erule CptnModEnv*)

apply (*rule disjI1*)

apply (*simp add:lift-def*)

apply *clarify*

apply (*rule-tac x=(Some Pa,t)#xsa in exI*)

```

apply(rule conjI)
  apply(erule CptnModEnv)
apply(rule disjI2)
apply(rule conjI)
  apply(case-tac xsa,simp,simp)
apply(rule-tac x=ys in exI)
apply(rule conjI)
  apply simp
apply(simp add:lift-def)
apply clarify
apply(erule ctran.cases,simp-all)
apply clarify
apply(rule-tac x=xs in exI)
apply simp
apply clarify
apply(rule-tac x=xs in exI)
apply(simp add: last-length)
done

lemma cptn-onlyif-cptn-mod-aux [rule-format]:
   $\forall s \ Q \ t \ xs. ((Some \ a, s), Q, t) \in ctran \longrightarrow (Q, t) \# xs \in cptn-mod$ 
   $\longrightarrow (Some \ a, s) \# (Q, t) \# xs \in cptn-mod$ 
  supply [[simproc del: defined-all]]
apply(induct a)
apply simp-all
— basic
apply clarify
apply(erule ctran.cases,simp-all)
apply(rule CptnModNone,rule Basic,simp)
apply clarify
apply(erule ctran.cases,simp-all)
— Seq1
apply(rule-tac xs=[(None,ta)] in CptnModSeq2)
  apply(erule CptnModNone)
  apply(rule CptnModOne)
  apply simp
apply simp
apply(simp add:lift-def)
— Seq2
apply(erule-tac x=sa in allE)
apply(erule-tac x=Some P2 in allE)
apply(erule allE,erule impE, assumption)
apply(erule div-seq,simp)
apply clarify
apply(erule disjE)
apply clarify
apply(erule allE,erule impE, assumption)
apply(erule-tac CptnModSeq1)
apply(simp add:lift-def)

```

```

apply clarify
apply(erule allE,erule impE, assumption)
apply(erule-tac CptnModSeq2)
  apply (simp add:last-length)
  apply (simp add:last-length)
apply(simp add:lift-def)
— Cond
apply clarify
apply(erule ctran.cases,simp-all)
apply(force elim: CptnModCondT)
apply(force elim: CptnModCondF)
— While
apply clarify
apply(erule ctran.cases,simp-all)
apply(rule CptnModNone,erule WhileF,simp)
apply(drule div-seq,force)
apply clarify
apply (erule disjE)
  apply(force elim:CptnModWhile1)
apply clarify
apply(force simp add:last-length elim:CptnModWhile2)
— await
apply clarify
apply(erule ctran.cases,simp-all)
apply(rule CptnModNone,erule Await,simp+)
done

lemma cptn-onlyif-cptn-mod [rule-format]:  $c \in \text{cptn} \implies c \in \text{cptn-mod}$ 
apply(erule cptn.induct)
  apply(rule CptnModOne)
  apply(erule CptnModEnv)
apply(case-tac P)
  apply simp
  apply(erule ctran.cases,simp-all)
apply(force elim:cptn-onlyif-cptn-mod-aux)
done

lemma lift-is-cptn:  $c \in \text{cptn} \implies \text{map } (\text{lift } P) \ c \in \text{cptn}$ 
apply(erule cptn.induct)
  apply(force simp add:lift-def CptnOne)
  apply(force intro:CptnEnv simp add:lift-def)
apply(force simp add:lift-def intro:CptnComp Seq2 Seq1 elim:ctran.cases)
done

lemma cptn-append-is-cptn [rule-format]:
 $\forall b \ a. \ b \# c1 \in \text{cptn} \longrightarrow a \# c2 \in \text{cptn} \longrightarrow (b \# c1) ! \text{length } c1 = a \longrightarrow b \# c1 @ c2 \in \text{cptn}$ 
apply(induct c1)
  apply simp
apply clarify

```

```

apply(erule cptn.cases,simp-all)
apply(force intro:CptnEnv)
apply(force elim:CptnComp)
done

```

```

lemma last-lift:  $\llbracket xs \neq []; fst(xs!(length\ xs - (Suc\ 0))) = None \rrbracket$ 
 $\implies fst((map\ (lift\ P)\ xs)!(length\ (map\ (lift\ P)\ xs) - (Suc\ 0))) = (Some\ P)$ 
by (cases (xs ! (length xs - (Suc 0)))) (simp add:lift-def)

```

```

lemma last-fst [rule-format]:  $P((a \# x) ! length\ x) \longrightarrow \neg P\ a \longrightarrow P\ (x!(length\ x - (Suc\ 0)))$ 
by (induct x) simp-all

```

```

lemma last-fst-esp:
 $fst(((Some\ a,s) \# xs)!(length\ xs)) = None \implies fst(xs!(length\ xs - (Suc\ 0))) = None$ 
apply(erule last-fst)
apply simp
done

```

```

lemma last-snd:  $xs \neq [] \implies$ 
 $snd(((map\ (lift\ P)\ xs)!(length\ (map\ (lift\ P)\ xs) - (Suc\ 0))) = snd(xs!(length\ xs - (Suc\ 0)))$ 
by (cases (xs ! (length xs - (Suc 0)))) (simp-all add:lift-def)

```

```

lemma Cons-lift:  $(Some\ (Seq\ P\ Q),\ s) \# (map\ (lift\ Q)\ xs) = map\ (lift\ Q)\ ((Some\ P,\ s) \# xs)$ 
by (simp add:lift-def)

```

```

lemma Cons-lift-append:
 $(Some\ (Seq\ P\ Q),\ s) \# (map\ (lift\ Q)\ xs) @ ys = map\ (lift\ Q)\ ((Some\ P,\ s) \# xs) @ ys$ 
by (simp add:lift-def)

```

```

lemma lift-nth:  $i < length\ xs \implies map\ (lift\ Q)\ xs\ !\ i = lift\ Q\ (xs\ !\ i)$ 
by (simp add:lift-def)

```

```

lemma snd-lift:  $i < length\ xs \implies snd(lift\ Q\ (xs\ !\ i)) = snd\ (xs\ !\ i)$ 
by (cases xs!i) (simp add:lift-def)

```

```

lemma cptn-if-cptn-mod:  $c \in cptn\ mod \implies c \in cptn$ 
apply(erule cptn-mod.induct)
apply(rule CptnOne)
apply(erule CptnEnv)
apply(erule CptnComp,simp)
apply(rule CptnComp)
apply(erule CondT,simp)
apply(rule CptnComp)
apply(erule CondF,simp)
— Seq1

```

```

apply(erule cptn.cases,simp-all)
  apply(rule CptnOne)
  apply clarify
  apply(drule-tac  $P=P1$  in lift-is-cptn)
  apply(simp add:lift-def)
  apply(rule CptnEnv,simp)
apply clarify
apply(simp add:lift-def)
apply(rule conjI)
  apply clarify
  apply(rule CptnComp)
  apply(rule Seq1,simp)
  apply(drule-tac  $P=P1$  in lift-is-cptn)
  apply(simp add:lift-def)
apply clarify
apply(rule CptnComp)
  apply(rule Seq2,simp)
apply(drule-tac  $P=P1$  in lift-is-cptn)
apply(simp add:lift-def)
— Seq2
apply(rule cptn-append-is-cptn)
  apply(drule-tac  $P=P1$  in lift-is-cptn)
  apply(simp add:lift-def)
  apply simp
apply(simp split: if-split-asm)
apply(frule-tac  $P=P1$  in last-lift)
  apply(rule last-fst-esp)
  apply (simp add:last-length)
apply(simp add:Cons-lift lift-def split-def last-conv-nth)
— While1
apply(rule CptnComp)
  apply(rule WhileT,simp)
apply(drule-tac  $P=While\ b\ P$  in lift-is-cptn)
apply(simp add:lift-def)
— While2
apply(rule CptnComp)
  apply(rule WhileT,simp)
apply(rule cptn-append-is-cptn)
  apply(drule-tac  $P=While\ b\ P$  in lift-is-cptn)
  apply(simp add:lift-def)
  apply simp
apply(simp split: if-split-asm)
apply(frule-tac  $P=While\ b\ P$  in last-lift)
  apply(rule last-fst-esp,simp add:last-length)
apply(simp add:Cons-lift lift-def split-def last-conv-nth)
done

theorem cptn-iff-cptn-mod:  $(c \in cptn) = (c \in cptn-mod)$ 
apply(rule iffI)

```

apply(erule cptn-onlyif-cptn-mod)
 apply(erule cptn-if-cptn-mod)
 done

3.3 Validity of Correctness Formulas

3.3.1 Validity for Component Programs.

type-synonym 'a rgformula =
 'a com $\times$ 'a set $\times$ ('a $\times$ 'a) set $\times$ ('a $\times$ 'a) set $\times$ 'a set

definition *assum* :: ('a set $\times$ ('a $\times$ 'a) set) $\Rightarrow$ ('a confs) set **where**
assum $\equiv \lambda(pre, rely). \{c. snd(c!0) \in pre \wedge (\forall i. Suc\ i < length\ c \longrightarrow$
 $c!i -e \rightarrow c!(Suc\ i) \longrightarrow (snd(c!i), snd(c!Suc\ i)) \in rely)\}$

definition *comm* :: (('a $\times$ 'a) set $\times$ 'a set) $\Rightarrow$ ('a confs) set **where**
comm $\equiv \lambda(guar, post). \{c. (\forall i. Suc\ i < length\ c \longrightarrow$
 $c!i -c \rightarrow c!(Suc\ i) \longrightarrow (snd(c!i), snd(c!Suc\ i)) \in guar) \wedge$
 $(fst\ (last\ c) = None \longrightarrow snd\ (last\ c) \in post)\}$

definition *com-validity* :: 'a com $\Rightarrow$ 'a set $\Rightarrow$ ('a $\times$ 'a) set $\Rightarrow$ ('a $\times$ 'a) set $\Rightarrow$ 'a set $\Rightarrow$ bool

$(\models - sat [-, -, -, -] [60,0,0,0,0] 45)$ **where**
 $\models P sat [pre, rely, guar, post] \equiv$
 $\forall s. cp\ (Some\ P)\ s \cap assum(pre, rely) \subseteq comm(guar, post)$

3.3.2 Validity for Parallel Programs.

definition *All-None* :: (('a com) option) list $\Rightarrow$ bool **where**
All-None *xs* $\equiv \forall c \in set\ xs. c = None$

definition *par-assum* :: ('a set $\times$ ('a $\times$ 'a) set) $\Rightarrow$ ('a par-confs) set **where**
par-assum $\equiv \lambda(pre, rely). \{c. snd(c!0) \in pre \wedge (\forall i. Suc\ i < length\ c \longrightarrow$
 $c!i -pe \rightarrow c!Suc\ i \longrightarrow (snd(c!i), snd(c!Suc\ i)) \in rely)\}$

definition *par-comm* :: (('a $\times$ 'a) set $\times$ 'a set) $\Rightarrow$ ('a par-confs) set **where**
par-comm $\equiv \lambda(guar, post). \{c. (\forall i. Suc\ i < length\ c \longrightarrow$
 $c!i -pc \rightarrow c!Suc\ i \longrightarrow (snd(c!i), snd(c!Suc\ i)) \in guar) \wedge$
 $(All-None\ (fst\ (last\ c)) \longrightarrow snd\ (last\ c) \in post)\}$

definition *par-com-validity* :: 'a par-com $\Rightarrow$ 'a set $\Rightarrow$ ('a $\times$ 'a) set $\Rightarrow$ ('a $\times$ 'a) set $\Rightarrow$ bool

$\Rightarrow$ 'a set $\Rightarrow$ bool $(\models - SAT [-, -, -, -] [60,0,0,0,0] 45)$ **where**
 $\models Ps SAT [pre, rely, guar, post] \equiv$
 $\forall s. par-cp\ Ps \cap par-assum(pre, rely) \subseteq par-comm(guar, post)$

3.3.3 Compositionality of the Semantics

Definition of the conjoin operator

definition *same-length* :: 'a par-confs $\Rightarrow$ ('a confs) list $\Rightarrow$ bool **where**
same-length c clist $\equiv (\forall i < \text{length clist}. \text{length}(clist!i) = \text{length } c)$

definition *same-state* :: 'a par-confs $\Rightarrow$ ('a confs) list $\Rightarrow$ bool **where**
same-state c clist $\equiv (\forall i < \text{length clist}. \forall j < \text{length } c. \text{snd}(c!j) = \text{snd}((clist!i)!j))$

definition *same-program* :: 'a par-confs $\Rightarrow$ ('a confs) list $\Rightarrow$ bool **where**
same-program c clist $\equiv (\forall j < \text{length } c. \text{fst}(c!j) = \text{map } (\lambda x. \text{fst}(\text{nth } x \ j)) \text{ clist})$

definition *compat-label* :: 'a par-confs $\Rightarrow$ ('a confs) list $\Rightarrow$ bool **where**
compat-label c clist $\equiv (\forall j. \text{Suc } j < \text{length } c \longrightarrow$
 $(c!j - pc \rightarrow c!\text{Suc } j \wedge (\exists i < \text{length clist}. (clist!i)!j - c \rightarrow (clist!i)! \text{Suc } j \wedge$
 $(\forall l < \text{length clist}. l \neq i \longrightarrow (clist!l)!j - e \rightarrow (clist!l)! \text{Suc } j))) \vee$
 $(c!j - pe \rightarrow c!\text{Suc } j \wedge (\forall i < \text{length clist}. (clist!i)!j - e \rightarrow (clist!i)! \text{Suc } j)))$

definition *conjoin* :: 'a par-confs $\Rightarrow$ ('a confs) list $\Rightarrow$ bool $(- \propto - [65, 65] \ 64)$
where
 $c \propto \text{clist} \equiv (\text{same-length } c \text{ clist}) \wedge (\text{same-state } c \text{ clist}) \wedge (\text{same-program } c \text{ clist})$
 $\wedge (\text{compat-label } c \text{ clist})$

Some previous lemmas

lemma *list-eq-if* [rule-format]:
 $\forall ys. xs = ys \longrightarrow (\text{length } xs = \text{length } ys) \longrightarrow (\forall i < \text{length } xs. xs!i = ys!i)$
by (induct xs) auto

lemma *list-eq*: $(\text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. xs!i = ys!i)) = (xs = ys)$
apply (rule iffI)
apply clarify
apply (erule nth-equalityI)
apply simp+
done

lemma *nth-tl*: $\llbracket ys!0 = a; ys \neq [] \rrbracket \Longrightarrow ys = (a \# (\text{tl } ys))$
by (cases ys) simp-all

lemma *nth-tl-if* [rule-format]: $ys \neq [] \longrightarrow ys!0 = a \longrightarrow P \text{ } ys \longrightarrow P \ (a \# (\text{tl } ys))$
by (induct ys) simp-all

lemma *nth-tl-onlyif* [rule-format]: $ys \neq [] \longrightarrow ys!0 = a \longrightarrow P \ (a \# (\text{tl } ys)) \longrightarrow P \text{ } ys$
by (induct ys) simp-all

lemma *seq-not-eq1*: $\text{Seq } c1 \ c2 \neq c1$
by (induct c1) auto

lemma *seq-not-eq2*: $\text{Seq } c1 \ c2 \neq c2$

```

by (induct c2) auto

lemma if-not-eq1: Cond b c1 c2 ≠ c1
  by (induct c1) auto

lemma if-not-eq2: Cond b c1 c2 ≠ c2
  by (induct c2) auto

lemmas seq-and-if-not-eq [simp] = seq-not-eq1 seq-not-eq2
seq-not-eq1 [THEN not-sym] seq-not-eq2 [THEN not-sym]
if-not-eq1 if-not-eq2 if-not-eq1 [THEN not-sym] if-not-eq2 [THEN not-sym]

lemma prog-not-eq-in-ctran-aux:
  assumes c: (P,s) -c→ (Q,t)
  shows P≠Q using c
  by (induct x1 ≡ (P,s) x2 ≡ (Q,t) arbitrary: P s Q t) auto

lemma prog-not-eq-in-ctran [simp]: ¬ (P,s) -c→ (P,t)
  apply clarify
  apply (drule prog-not-eq-in-ctran-aux)
  apply simp
  done

lemma prog-not-eq-in-par-ctran-aux [rule-format]: (P,s) -pc→ (Q,t) ⇒ (P≠Q)
  apply (erule par-ctran.induct)
  apply (drule prog-not-eq-in-ctran-aux)
  apply clarify
  apply (drule list-eq-if)
  apply simp-all
  apply force
  done

lemma prog-not-eq-in-par-ctran [simp]: ¬ (P,s) -pc→ (P,t)
  apply clarify
  apply (drule prog-not-eq-in-par-ctran-aux)
  apply simp
  done

lemma tl-in-cptn: [ a#xs ∈ cptn; xs≠[] ] ⇒ xs∈cptn
  by (force elim: cptn.cases)

lemma tl-zero[rule-format]:
  P (ys!Suc j) → Suc j < length ys → ys≠[] → P (tl(ys)!j)
  by (induct ys) simp-all

```

3.3.4 The Semantics is Compositional

```

lemma aux-if [rule-format]:
  ∀ xs s clist. (length clist = length xs ∧ (∀ i < length xs. (xs!i,s)#clist!i ∈ cptn)

```

```

 $\wedge ((xs, s) \# ys \propto \text{map } (\lambda i. (fst\ i, s) \# snd\ i) (\text{zip } xs\ clist))$ 
 $\longrightarrow (xs, s) \# ys \in \text{par-cptn}$ 
apply(induct ys)
apply(clarify)
apply(rule ParCptnOne)
apply(clarify)
apply(simp add:conjoin-def compat-label-def)
apply clarify
apply(erule-tac x=0 and P= $\lambda j. H\ j \longrightarrow (P\ j \vee Q\ j)$  for H P Q in all-dupE,
simp)
apply(erule disjE)
— first step is a Component step
apply clarify
apply simp
apply(subgoal-tac a=(xs[i:=(fst(clist!i!0))]))
apply(subgoal-tac b=snd(clist!i!0),simp)
prefer 2
apply(simp add:same-state-def)
apply(erule-tac x=i in allE, erule impE, assumption,
erule-tac x=1 and P= $\lambda j. (H\ j) \longrightarrow (snd\ (d\ j)) = (snd\ (e\ j))$  for H d e in
allE, simp)
prefer 2
apply(simp add:same-program-def)
apply(erule-tac x=1 and P= $\lambda j. H\ j \longrightarrow (fst\ (s\ j)) = (t\ j)$  for H s t in allE, simp)
apply(rule nth-equalityI, simp)
apply clarify
apply(case-tac i=ia,simp,simp)
apply(erule-tac x=ia and P= $\lambda j. H\ j \longrightarrow I\ j \longrightarrow J\ j$  for H I J in allE)
apply(drule-tac t=i in not-sym, simp)
apply(erule etranE, simp)
apply(rule ParCptnComp)
apply(erule ParComp, simp)
— applying the induction hypothesis
apply(erule-tac x=xs[i := fst (clist ! i ! 0)] in allE)
apply(erule-tac x=snd (clist ! i ! 0) in allE)
apply(erule mp)
apply(rule-tac x=map tl clist in exI, simp)
apply(rule conjI, clarify)
apply(case-tac i=ia,simp)
apply(rule nth-tl-if)
apply(force simp add:same-length-def length-Suc-conv)
apply simp
apply(erule allE, erule impE, assumption, erule tl-in-cptn)
apply(force simp add:same-length-def length-Suc-conv)
apply(rule nth-tl-if)
apply(force simp add:same-length-def length-Suc-conv)
apply(simp add:same-state-def)
apply(erule-tac x=ia in allE, erule impE, assumption,
erule-tac x=1 and P= $\lambda j. H\ j \longrightarrow (snd\ (d\ j)) = (snd\ (e\ j))$  for H d e in allE)

```

```

    apply(erule-tac x=ia and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE)
    apply(drule-tac t=i in not-sym,simp)
    apply(erule etranE,simp)
    apply(erule allE,erule impE,assumption,erule tl-in-cptn)
    apply(force simp add:same-length-def length-Suc-conv)
    apply(simp add:same-length-def same-state-def)
    apply(rule conjI)
    apply clarify
    apply(case-tac j,simp,simp)
    apply(erule-tac x=ia in allE, erule impE, assumption,
           erule-tac x=Suc(Suc nat) and P= $\lambda j. H j \longrightarrow (snd (d j))=(snd (e j))$  for
H d e in allE,simp)
    apply(force simp add:same-length-def length-Suc-conv)
    apply(rule conjI)
    apply(simp add:same-program-def)
    apply clarify
    apply(case-tac j,simp)
    apply(rule nth-equalityI,simp)
    apply clarify
    apply(case-tac i=ia,simp,simp)
    apply(erule-tac x=Suc(Suc nat) and P= $\lambda j. H j \longrightarrow (fst (s j))=(t j)$  for  $H s t$ 
in allE,simp)
    apply(rule nth-equalityI,simp,simp)
    apply(force simp add:length-Suc-conv)
    apply(rule allI,erule impI)
    apply(erule-tac x=Suc j and P= $\lambda j. H j \longrightarrow (I j \vee J j)$  for  $H I J$  in allE,simp)
    apply(erule disjE)
    apply clarify
    apply(rule-tac x=ia in exI,simp)
    apply(case-tac i=ia,simp)
    apply(rule conjI)
    apply(force simp add: length-Suc-conv)
    apply clarify
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE,erule
impE,assumption)
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE,erule
impE,assumption)
    apply simp
    apply(case-tac j,simp)
    apply(rule tl-zero)
    apply(erule-tac x=l in allE, erule impE, assumption,
           erule-tac x=1 and P= $\lambda j. (H j) \longrightarrow (snd (d j))=(snd (e j))$  for  $H d e$ 
in allE,simp)
    apply(force elim:etranE intro:Env)
    apply force
    apply force
    apply simp
    apply(rule tl-zero)
    apply(erule tl-zero)

```

```

    apply force
    apply force
    apply force
    apply force
    apply(rule conjI,simp)
    apply(rule nth-tl-if)
    apply force
    apply(erule-tac x=ia in allE, erule impE, assumption,
      erule-tac x=1 and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in
allE)
    apply(erule-tac x=ia and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE)
    apply(drule-tac t=i in not-sym,simp)
    apply(erule etranE,simp)
    apply(erule tl-zero)
    apply force
    apply force
    apply clarify
    apply(case-tac i=l,simp)
    apply(rule nth-tl-if)
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in
allE,force)
    apply simp
    apply(erule-tac P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE,erule impE,assumption,erule
impE,assumption)
    apply(erule tl-zero,force)
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in allE,force)
    apply(rule nth-tl-if)
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in
allE,force)
    apply(erule-tac x=l in allE, erule impE, assumption,
      erule-tac x=1 and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in
allE)
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE,erule
impE, assumption,simp)
    apply(erule etranE,simp)
    apply(rule tl-zero)
    apply force
    apply force
    apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in allE,force)
    apply(rule disjI2)
    apply(case-tac j,simp)
    apply clarify
    apply(rule tl-zero)
    apply(erule-tac x=ia and P= $\lambda j. H j \longrightarrow I j \in \text{etran}$  for  $H I$  in allE,erule impE,
assumption)
    apply(case-tac i=ia,simp,simp)
    apply(erule-tac x=ia in allE, erule impE, assumption,
      erule-tac x=1 and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in allE)
    apply(erule-tac x=ia and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for  $H I J$  in allE,erule

```

```

impE, assumption, simp)
  apply(force elim:etranE intro:Env)
  apply force
  apply(erule-tac x=ia and P= $\lambda j$ .  $H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in
allE, force)
  apply simp
  apply clarify
  apply(rule tl-zero)
  apply(rule tl-zero, force)
  apply force
  apply(erule-tac x=ia and P= $\lambda j$ .  $H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in
allE, force)
  apply force
  apply(erule-tac x=ia and P= $\lambda j$ .  $H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in allE, force)
— first step is an environmental step
  apply clarify
  apply(erule par-etran.cases)
  apply simp
  apply(rule ParCptnEnv)
  apply(erule-tac x=Ps in allE)
  apply(erule-tac x=t in allE)
  apply(erule mp)
  apply(rule-tac x=map tl clist in exI, simp)
  apply(rule conjI)
  apply clarify
  apply(erule-tac x=i and P= $\lambda j$ .  $H j \longrightarrow I j \in \text{cptn}$  for  $H I$  in allE, simp)
  apply(erule cptn.cases)
    apply(simp add:same-length-def)
    apply(erule-tac x=i and P= $\lambda j$ .  $H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in allE, force)
    apply(simp add:same-state-def)
    apply(erule-tac x=i in allE, erule impE, assumption,
      erule-tac x=1 and P= $\lambda j$ .  $H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in allE, simp)
    apply(erule-tac x=i and P= $\lambda j$ .  $H j \longrightarrow J j \in \text{etran}$  for  $H J$  in allE, simp)
    apply(erule etranE, simp)
  apply(simp add:same-state-def same-length-def)
  apply(rule conjI, clarify)
  apply(case-tac j, simp, simp)
  apply(erule-tac x=i in allE, erule impE, assumption,
    erule-tac x=Suc(Suc nat) and P= $\lambda j$ .  $H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H$ 
d e in allE, simp)
  apply(rule tl-zero)
  apply(simp)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $H j \longrightarrow (\text{length } (s j) = t)$  for  $H s t$  in allE, force)
  apply(rule conjI)
  apply(simp add:same-program-def)
  apply clarify
  apply(case-tac j, simp)
  apply(rule nth-equalityI, simp)

```

```

apply clarify
apply simp
apply(erule-tac  $x = \text{Suc}(\text{Suc } \text{nat})$  and  $P = \lambda j. H\ j \longrightarrow (\text{fst } (s\ j)) = (t\ j)$  for  $H\ s\ t$ 
in  $\text{allE}, \text{simp}$ )
apply(rule nth-equalityI, simp, simp)
apply(force simp add:length-Suc-conv)
apply(rule allI, rule impI)
apply(erule-tac  $x = \text{Suc } j$  and  $P = \lambda j. H\ j \longrightarrow (I\ j \vee J\ j)$  for  $H\ I\ J$  in  $\text{allE}, \text{simp}$ )
apply(erule disjE)
apply clarify
apply(rule-tac  $x = i$  in  $\text{exI}, \text{simp}$ )
apply(rule conjI)
apply(erule-tac  $x = i$  and  $P = \lambda i. H\ i \longrightarrow J\ i \in \text{etran}$  for  $H\ J$  in  $\text{allE}$ , erule impE,
assumption)
apply(erule etranE, simp)
apply(erule-tac  $x = i$  in  $\text{allE}$ , erule impE, assumption,
erule-tac  $x = 1$  and  $P = \lambda j. H\ j \longrightarrow (\text{snd } (d\ j)) = (\text{snd } (e\ j))$  for  $H\ d\ e$  in
 $\text{allE}, \text{simp}$ )
apply(rule nth-tl-if)
apply(erule-tac  $x = i$  and  $P = \lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in  $\text{allE}, \text{force}$ )
apply simp
apply(erule tl-zero, force)
apply(erule-tac  $x = i$  and  $P = \lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in  $\text{allE}, \text{force}$ )
apply clarify
apply(erule-tac  $x = l$  and  $P = \lambda i. H\ i \longrightarrow J\ i \in \text{etran}$  for  $H\ J$  in  $\text{allE}$ , erule impE,
assumption)
apply(erule etranE, simp)
apply(erule-tac  $x = l$  in  $\text{allE}$ , erule impE, assumption,
erule-tac  $x = 1$  and  $P = \lambda j. H\ j \longrightarrow (\text{snd } (d\ j)) = (\text{snd } (e\ j))$  for  $H\ d\ e$  in
 $\text{allE}, \text{simp}$ )
apply(rule nth-tl-if)
apply(erule-tac  $x = l$  and  $P = \lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in  $\text{allE}, \text{force}$ )
apply simp
apply(rule tl-zero, force)
apply force
apply(erule-tac  $x = l$  and  $P = \lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in  $\text{allE}, \text{force}$ )
apply(rule disjI2)
apply simp
apply clarify
apply(case-tac  $j$ , simp)
apply(rule tl-zero)
apply(erule-tac  $x = i$  and  $P = \lambda i. H\ i \longrightarrow J\ i \in \text{etran}$  for  $H\ J$  in  $\text{allE}$ , erule
impE, assumption)
apply(erule-tac  $x = i$  and  $P = \lambda i. H\ i \longrightarrow J\ i \in \text{etran}$  for  $H\ J$  in  $\text{allE}$ , erule
impE, assumption)
apply(force elim:etranE intro:Env)
apply force
apply(erule-tac  $x = i$  and  $P = \lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in  $\text{allE}, \text{force}$ )
apply simp

```

```

apply(rule tl-zero)
  apply(rule tl-zero,force)
  apply force
  apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE,force)
  apply force
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE,force)
done

lemma aux-onlyif [rule-format]:  $\forall xs s. (xs, s) \# ys \in \text{par-cptn} \longrightarrow$ 
  ( $\exists \text{clist}. (\text{length } \text{clist} = \text{length } xs) \wedge$ 
   $(xs, s) \# ys \propto \text{map } (\lambda i. (\text{fst } i, s) \# (\text{snd } i)) (\text{zip } xs \text{ clist}) \wedge$ 
   $(\forall i < \text{length } xs. (xs!i, s) \# (\text{clist}!i) \in \text{cptn}))$ 
  supply [[simproc del: defined-all]]
apply(induct ys)
apply(clarify)
apply(rule-tac x=map ( $\lambda i. []$ ) [0.. $\text{length } xs$ ] in exI)
apply(simp add: conjoin-def same-length-def same-state-def same-program-def compat-label-def)
apply(rule conjI)
  apply(rule nth-equalityI,simp,simp)
  apply(force intro: cptn.intros)
apply(clarify)
apply(erule par-cptn.cases,simp)
  apply simp
  apply(erule-tac x=xs in allE)
  apply(erule-tac x=t in allE,simp)
  apply clarify
  apply(rule-tac x=(map ( $\lambda j. (P!j, t) \# (\text{clist}!j)$ ) [0.. $\text{length } P$ ]) in exI,simp)
  apply(rule conjI)
  prefer 2
  apply clarify
  apply(rule CptnEnv,simp)
apply(simp add:conjoin-def same-length-def same-state-def)
apply (rule conjI)
  apply clarify
  apply(case-tac j,simp,simp)
apply(rule conjI)
  apply(simp add:same-program-def)
  apply clarify
  apply(case-tac j,simp)
  apply(rule nth-equalityI,simp,simp)
  apply simp
  apply(rule nth-equalityI,simp,simp)
apply(simp add:compat-label-def)
apply clarify
apply(case-tac j,simp)
apply(simp add:ParEnv)
apply clarify
apply(simp add:Env)

```

```

apply simp
apply(erule-tac  $x=\text{nat}$  in allE,erule impE, assumption)
apply(erule disjE,simp)
  apply clarify
  apply(rule-tac  $x=i$  in exI,simp)
apply force
apply(erule par-ctran.cases,simp)
apply(erule-tac  $x=Ps[i:=r]$  in allE)
apply(erule-tac  $x=ta$  in allE,simp)
apply clarify
apply(rule-tac  $x=(\text{map } (\lambda j. (Ps!j, ta)\#(clist!j)) [0..\text{length } Ps]) [i:=((r, ta)\#(clist!i))]$ 
in exI,simp)
apply(rule conjI)
prefer 2
apply clarify
apply(case-tac  $i=ia$ ,simp)
  apply(erule CptnComp)
  apply(erule-tac  $x=ia$  and  $P=\lambda j. H\ j \longrightarrow (I\ j \in \text{cptn})$  for  $H\ I$  in allE,simp)
apply simp
apply(erule-tac  $x=ia$  in allE)
apply(rule CptnEnv,simp)
apply(simp add:conjoin-def)
apply (rule conjI)
  apply(simp add:same-length-def)
  apply clarify
  apply(case-tac  $i=ia$ ,simp,simp)
apply(rule conjI)
  apply(simp add:same-state-def)
  apply clarify
  apply(case-tac  $j$ , simp, simp (no-asm-simp))
  apply(case-tac  $i=ia$ ,simp,simp)
apply(rule conjI)
  apply(simp add:same-program-def)
  apply clarify
  apply(case-tac  $j$ ,simp)
  apply(rule nth-equalityI,simp,simp)
apply simp
apply(rule nth-equalityI,simp,simp)
apply(erule-tac  $x=\text{nat}$  and  $P=\lambda j. H\ j \longrightarrow (\text{fst } (a\ j))=((b\ j))$  for  $H\ a\ b$  in allE)
apply(case-tac  $\text{nat}$ )
  apply clarify
  apply(case-tac  $i=ia$ ,simp,simp)
  apply clarify
  apply(case-tac  $i=ia$ ,simp,simp)
apply(simp add:compat-label-def)
apply clarify
apply(case-tac  $j$ )
  apply(rule conjI,simp)
  apply(erule ParComp,assumption)

```

```

  apply clarify
  apply(rule-tac x=i in exI,simp)
  apply clarify
  apply(rule Env)
  apply simp
  apply(erule-tac x=nat and P=λj. H j → (P j ∨ Q j) for H P Q in allE,simp)
  apply(erule disjE)
  apply clarify
  apply(rule-tac x=ia in exI,simp)
  apply(rule conjI)
  apply(case-tac i=ia,simp,simp)
  apply clarify
  apply(case-tac i=l,simp)
  apply(case-tac l=ia,simp,simp)
  apply(erule-tac x=l in allE,erule impE,assumption,erule impE, assumption,simp)
  apply simp
  apply(erule-tac x=l in allE,erule impE,assumption,erule impE, assumption,simp)
  apply clarify
  apply(erule-tac x=ia and P=λj. H j → (P j)∈etran for H P in allE, erule
  impE, assumption)
  apply(case-tac i=ia,simp,simp)
  done

lemma one-iff-aux: xs≠[] ⇒ (∀ ys. ((xs, s)#ys ∈ par-cptn) =
  (∃ clist. length clist= length xs ∧
    ((xs, s)#ys ∝ map (λi. (fst i,s)#(snd i)) (zip xs clist)) ∧
    (∀ i<length xs. (xs!i,s)#(clist!i) ∈ cptn))) =
  (par-cp (xs) s = {c. ∃ clist. (length clist)=(length xs) ∧
    (∀ i<length clist. (clist!i) ∈ cp(xs!i) s) ∧ c ∝ clist})
  apply (rule iffI)
  apply(rule subset-antisym)
  apply(rule subsetI)
  apply(clarify)
  apply(simp add:par-cp-def cp-def)
  apply(case-tac x)
  apply(force elim:par-cptn.cases)
  apply simp
  apply(rename-tac a list)
  apply(erule-tac x=list in allE)
  apply clarify
  apply simp
  apply(rule-tac x=map (λi. (fst i, s) # snd i) (zip xs clist) in exI,simp)
  apply(rule subsetI)
  apply(clarify)
  apply(case-tac x)
  apply(erule-tac x=0 in allE)
  apply(simp add:cp-def conjoin-def same-length-def same-program-def same-state-def
  compat-label-def)
  apply clarify

```

```

apply(erule cptn.cases,force,force,force)
apply(simp add:par-cp-def conjoin-def same-length-def same-program-def same-state-def
compat-label-def)
apply clarify
apply(erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in all-dupE)
apply(subgoal-tac  $a = xs$ )
apply(subgoal-tac  $b = s, \text{simp}$ )
prefer 3
apply(erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\text{fst } (s\ j))=((t\ j))$  for  $H\ s\ t$  in allE)
apply (simp add:cp-def)
apply(rule nth-equalityI,simp,simp)
prefer 2
apply(erule-tac  $x=0$  in allE)
apply (simp add:cp-def)
apply(erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\forall i. T\ i \longrightarrow (\text{snd } (d\ j\ i))=(\text{snd } (e\ j\ i)))$  for  $H\ T\ d\ e$  in allE,simp)
apply(erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\text{snd } (d\ j))=(\text{snd } (e\ j))$  for  $H\ d\ e$  in allE,simp)
apply(erule-tac  $x=\text{list}$  in allE)
apply(rule-tac  $x=\text{map } tl\ clist$  in exI,simp)
apply(rule conjI)
apply clarify
apply(case-tac  $j, \text{simp}$ )
apply(erule-tac  $x=i$  in allE, erule impE, assumption,
    erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\text{snd } (d\ j))=(\text{snd } (e\ j))$  for  $H\ d\ e$  in allE,simp)
apply(erule-tac  $x=i$  in allE, erule impE, assumption,
    erule-tac  $x=\text{Suc } nat$  and  $P=\lambda j. H\ j \longrightarrow (\text{snd } (d\ j))=(\text{snd } (e\ j))$  for  $H\ d\ e$  in allE)
apply(erule-tac  $x=i$  and  $P=\lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
apply(case-tac clist!i,simp,simp)
apply(rule conjI)
apply clarify
apply(rule nth-equalityI,simp,simp)
apply(case-tac  $j$ )
apply clarify
apply(erule-tac  $x=i$  in allE)
apply(simp add:cp-def)
apply clarify
apply simp
apply(erule-tac  $x=i$  and  $P=\lambda j. H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
apply(case-tac clist!i,simp,simp)
apply(thin-tac  $H = (\exists i. J\ i)$  for  $H\ J$ )
apply(rule conjI)
apply clarify
apply(erule-tac  $x=j$  in allE,erule impE, assumption,erule disjE)
apply clarify
apply(rule-tac  $x=i$  in exI,simp)
apply(case-tac  $j, \text{simp}$ )

```

```

apply(rule conjI)
apply(erule-tac x=i in allE)
apply(simp add:cp-def)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!i,simp,simp)
apply clarify
apply(erule-tac x=l in allE)
apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for H I J in allE)
apply clarify
apply(simp add:cp-def)
apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!l,simp,simp)
apply simp
apply(rule conjI)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!i,simp,simp)
apply clarify
apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow I j \longrightarrow J j$  for H I J in allE)
apply(erule-tac x=l and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!l,simp,simp)
apply clarify
apply(erule-tac x=i in allE)
apply(simp add:cp-def)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!i,simp)
apply(rule nth-tl-if,simp,simp)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (P j) \in \text{etran}$  for H P in allE, erule
impE, assumption,simp)
apply(simp add:cp-def)
apply clarify
apply(rule nth-tl-if)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{length } (s j) = t)$  for H s t in allE)
apply(case-tac clist!i,simp,simp)
apply force
apply force
apply clarify
apply(rule iffI)
apply(simp add:par-cp-def)
apply(erule-tac c=(xs, s) # ys in equalityCE)
apply simp
apply clarify
apply(rule-tac x=map tl clist in exI)
apply simp
apply (rule conjI)
apply(simp add:conjoin-def cp-def)
apply(rule conjI)
apply clarify
apply(unfold same-length-def)
apply clarify

```

```

    apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in
allE,simp)
  apply(rule conjI)
  apply(simp add:same-state-def)
  apply clarify
  apply(erule-tac x=i in allE, erule impE, assumption,
    erule-tac x=j and P= $\lambda j$ .  $H\ j \longrightarrow (\text{snd } (d\ j)) = (\text{snd } (e\ j))$  for  $H\ d\ e$  in allE)
  apply(case-tac j,simp)
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
  apply(case-tac clist!i,simp,simp)
  apply(rule conjI)
  apply(simp add:same-program-def)
  apply clarify
  apply(rule nth-equalityI,simp,simp)
  apply(case-tac j,simp)
  apply clarify
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
  apply(case-tac clist!i,simp,simp)
  apply clarify
  apply(simp add:compat-label-def)
  apply(rule allI,rule impI)
  apply(erule-tac x=j in allE,erule impE, assumption)
  apply(erule disjE)
  apply clarify
  apply(rule-tac x=i in exI,simp)
  apply(rule conjI)
  apply(erule-tac x=i in allE)
  apply(case-tac j,simp)
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
  apply(case-tac clist!i,simp,simp)
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
  apply(case-tac clist!i,simp,simp)
  apply clarify
  apply(erule-tac x=l and P= $\lambda j$ .  $H\ j \longrightarrow I\ j \longrightarrow J\ j$  for  $H\ I\ J$  in allE)
  apply(erule-tac x=l and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE)
  apply(case-tac clist!l,simp,simp)
  apply(erule-tac x=l in allE,simp)
  apply(rule disjI2)
  apply clarify
  apply(rule tl-zero)
  apply(case-tac j,simp,simp)
  apply(rule tl-zero,force)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in
allE,force)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $H\ j \longrightarrow (\text{length } (s\ j) = t)$  for  $H\ s\ t$  in allE,force)
  apply clarify
  apply(erule-tac x=i in allE)

```

```

apply(simp add:cp-def)
apply(rule nth-tl-if)
  apply(simp add:conjoin-def)
  apply clarify
  apply(simp add:same-length-def)
  apply(erule-tac x=i in allE,simp)
  apply simp
  apply simp
  apply simp
apply clarify
apply(erule-tac c=(xs, s) # ys in equalityCE)
  apply(simp add:par-cp-def)
apply simp
apply(erule-tac x=map (λi. (fst i, s) # snd i) (zip xs clist) in allE)
apply simp
apply clarify
apply(simp add:cp-def)
done

theorem one: xs≠[] ⇒
  par-cp xs s = {c. ∃ clist. (length clist)=(length xs) ∧
    (∀ i<length clist. (clist!i) ∈ cp(xs!i) s) ∧ c ∝ clist}
apply(frule one-iff-aux)
apply(drule sym)
apply(erule iffD2)
apply clarify
apply(rule iffI)
  apply(erule aux-onlyif)
apply clarify
apply(force intro:aux-if)
done

end

```

3.4 The Proof System

theory *RG-Hoare* **imports** *RG-Tran* **begin**

3.4.1 Proof System for Component Programs

declare *Un-subset-iff* [*simp del*] *sup.bounded-iff* [*simp del*]

definition *stable* :: 'a set ⇒ ('a × 'a) set ⇒ bool **where**
stable ≡ λf g. (∀ x y. x ∈ f ⟶ (x, y) ∈ g ⟶ y ∈ f)

inductive

rghoare :: ['a com, 'a set, ('a × 'a) set, ('a × 'a) set, 'a set] ⇒ bool
 (⊢ - sat [-, -, -, -] [60,0,0,0,0] 45)

where

Basic: $\llbracket pre \subseteq \{s. f \ s \in post\}; \{(s,t). s \in pre \wedge (t=f \ s \vee t=s)\} \subseteq guar; \text{stable } pre \text{ rely}; \text{stable } post \text{ rely} \rrbracket$
 $\implies \vdash \text{Basic } f \text{ sat } [pre, \text{rely}, guar, post]$

Seq: $\llbracket \vdash P \text{ sat } [pre, \text{rely}, guar, mid]; \vdash Q \text{ sat } [mid, \text{rely}, guar, post] \rrbracket$
 $\implies \vdash \text{Seq } P \ Q \text{ sat } [pre, \text{rely}, guar, post]$

Cond: $\llbracket \text{stable } pre \text{ rely}; \vdash P1 \text{ sat } [pre \cap b, \text{rely}, guar, post]; \vdash P2 \text{ sat } [pre \cap \neg b, \text{rely}, guar, post]; \forall s. (s,s) \in guar \rrbracket$
 $\implies \vdash \text{Cond } b \ P1 \ P2 \text{ sat } [pre, \text{rely}, guar, post]$

While: $\llbracket \text{stable } pre \text{ rely}; (pre \cap \neg b) \subseteq post; \text{stable } post \text{ rely}; \vdash P \text{ sat } [pre \cap b, \text{rely}, guar, pre]; \forall s. (s,s) \in guar \rrbracket$
 $\implies \vdash \text{While } b \ P \text{ sat } [pre, \text{rely}, guar, post]$

Await: $\llbracket \text{stable } pre \text{ rely}; \text{stable } post \text{ rely}; \forall V. \vdash P \text{ sat } [pre \cap b \cap \{V\}, \{(s, t). s = t\}, UNIV, \{s. (V, s) \in guar\} \cap post] \rrbracket$
 $\implies \vdash \text{Await } b \ P \text{ sat } [pre, \text{rely}, guar, post]$

Conseq: $\llbracket pre \subseteq pre'; \text{rely} \subseteq \text{rely}'; guar' \subseteq guar; post' \subseteq post; \vdash P \text{ sat } [pre', \text{rely}', guar', post'] \rrbracket$
 $\implies \vdash P \text{ sat } [pre, \text{rely}, guar, post]$

definition $Pre :: 'a \text{ rgformula} \Rightarrow 'a \text{ set}$ **where**
 $Pre \ x \equiv fst(snd \ x)$

definition $Post :: 'a \text{ rgformula} \Rightarrow 'a \text{ set}$ **where**
 $Post \ x \equiv snd(snd(snd \ x))$

definition $Rely :: 'a \text{ rgformula} \Rightarrow ('a \times 'a) \text{ set}$ **where**
 $Rely \ x \equiv fst(snd(snd \ x))$

definition $Guar :: 'a \text{ rgformula} \Rightarrow ('a \times 'a) \text{ set}$ **where**
 $Guar \ x \equiv fst(snd(snd(snd \ x)))$

definition $Com :: 'a \text{ rgformula} \Rightarrow 'a \text{ com}$ **where**
 $Com \ x \equiv fst \ x$

3.4.2 Proof System for Parallel Programs

type-synonym $'a \text{ par-rgformula} =$
 $('a \text{ rgformula}) \text{ list} \times 'a \text{ set} \times ('a \times 'a) \text{ set} \times ('a \times 'a) \text{ set} \times 'a \text{ set}$

inductive

$par\text{-}rg\text{hoare} :: ('a \text{ rgformula}) \text{ list} \Rightarrow 'a \text{ set} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow bool$
 $(\vdash - SAT [-, -, -, -] [60, 0, 0, 0, 0] 45)$

where

Parallel:

$$\begin{aligned} & \llbracket \forall i < \text{length } xs. \text{ rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. \text{ Guar}(xs!j)) \subseteq \text{ Rely}(xs!i); \\ & (\bigcup j \in \{j. j < \text{length } xs\}. \text{ Guar}(xs!j)) \subseteq \text{ guar}; \\ & \text{ pre} \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. \text{ Pre}(xs!i)); \\ & (\bigcap i \in \{i. i < \text{length } xs\}. \text{ Post}(xs!i)) \subseteq \text{ post}; \\ & \forall i < \text{length } xs. \vdash \text{ Com}(xs!i) \text{ sat } [\text{ Pre}(xs!i), \text{ Rely}(xs!i), \text{ Guar}(xs!i), \text{ Post}(xs!i)] \rrbracket \\ & \implies \vdash xs \text{ SAT } [\text{ pre}, \text{ rely}, \text{ guar}, \text{ post}] \end{aligned}$$

3.5 Soundness

Some previous lemmas

lemma *tl-of-assum-in-assum*:

$$(P, s) \# (P, t) \# xs \in \text{assum}(\text{pre}, \text{rely}) \implies \text{stable pre rely}$$

$$\implies (P, t) \# xs \in \text{assum}(\text{pre}, \text{rely})$$
apply(*simp add:assum-def*)
apply *clarify*
apply(*rule conjI*)
apply(*erule-tac x=0 in allE*)
apply(*simp (no-asm-use) only:stable-def*)
apply(*erule allE,erule allE,erule impE,assumption,erule mp*)
apply(*simp add:Env*)
apply *clarify*
apply(*erule-tac x=Suc i in allE*)
apply *simp*
done

lemma *etran-in-comm*:

$$(P, t) \# xs \in \text{comm}(\text{guar}, \text{post}) \implies (P, s) \# (P, t) \# xs \in \text{comm}(\text{guar}, \text{post})$$
apply(*simp add:comm-def*)
apply *clarify*
apply(*case-tac i,simp+*)
done

lemma *ctran-in-comm*:

$$\llbracket (s, s) \in \text{guar}; (Q, s) \# xs \in \text{comm}(\text{guar}, \text{post}) \rrbracket$$

$$\implies (P, s) \# (Q, s) \# xs \in \text{comm}(\text{guar}, \text{post})$$
apply(*simp add:comm-def*)
apply *clarify*
apply(*case-tac i,simp+*)
done

lemma *takecptn-is-cptn* [*rule-format, elim!*]:

$$\forall j. c \in \text{cptn} \longrightarrow \text{take } (Suc j) c \in \text{cptn}$$
apply(*induct c*)
apply(*force elim: cptn.cases*)
apply *clarify*
apply(*case-tac j*)
apply *simp*

```

  apply(rule CptnOne)
  apply simp
  apply(force intro:cptn.intros elim:cptn.cases)
  done

```

```

lemma dropcptn-is-cptn [rule-format,elim!]:
   $\forall j < \text{length } c. c \in \text{cptn} \longrightarrow \text{drop } j \ c \in \text{cptn}$ 
  apply(induct c)
  apply(force elim: cptn.cases)
  apply clarify
  apply(case-tac j,simp+)
  apply(erule cptn.cases)
  apply simp
  apply force
  apply force
  done

```

```

lemma takepar-cptn-is-par-cptn [rule-format,elim]:
   $\forall j. c \in \text{par-cptn} \longrightarrow \text{take } (\text{Suc } j) \ c \in \text{par-cptn}$ 
  apply(induct c)
  apply(force elim: cptn.cases)
  apply clarify
  apply(case-tac j,simp)
  apply(rule ParCptnOne)
  apply(force intro:par-cptn.intros elim:par-cptn.cases)
  done

```

```

lemma droppar-cptn-is-par-cptn [rule-format]:
   $\forall j < \text{length } c. c \in \text{par-cptn} \longrightarrow \text{drop } j \ c \in \text{par-cptn}$ 
  apply(induct c)
  apply(force elim: par-cptn.cases)
  apply clarify
  apply(case-tac j,simp+)
  apply(erule par-cptn.cases)
  apply simp
  apply force
  apply force
  done

```

```

lemma tl-of-cptn-is-cptn:  $\llbracket x \# xs \in \text{cptn}; xs \neq [] \rrbracket \Longrightarrow xs \in \text{cptn}$ 
  apply(subgoal-tac 1 < length (x # xs))
  apply(drule dropcptn-is-cptn,simp+)
  done

```

```

lemma not-ctran-None [rule-format]:
   $\forall s. (\text{None}, s) \# xs \in \text{cptn} \longrightarrow (\forall i < \text{length } xs. ((\text{None}, s) \# xs)!i -e\rightarrow xs!i)$ 
  apply(induct xs,simp+)
  apply clarify
  apply(erule cptn.cases,simp)

```

```

apply simp
apply(case-tac i, simp)
  apply(rule Env)
apply simp
apply(force elim:ctran.cases)
done

```

```

lemma cptn-not-empty [simp]: []  $\notin$  cptn
apply(force elim:cptn.cases)
done

```

```

lemma etran-or-ctran [rule-format]:
   $\forall m\ i. x \in \text{cptn} \longrightarrow m \leq \text{length } x$ 
   $\longrightarrow (\forall i. \text{Suc } i < m \longrightarrow \neg x!i -c \rightarrow x!\text{Suc } i) \longrightarrow \text{Suc } i < m$ 
   $\longrightarrow x!i -e \rightarrow x!\text{Suc } i$ 
  supply [[simproc del: defined-all]]
apply(induct x, simp)
apply clarify
apply(erule cptn.cases, simp)
apply(case-tac i, simp)
  apply(rule Env)
apply simp
apply(erule-tac x=m - 1 in allE)
apply(case-tac m, simp, simp)
apply(subgoal-tac ( $\forall i. \text{Suc } i < \text{ nata} \longrightarrow (((P, t) \# xs) ! i, xs ! i) \notin \text{ctran}$ ))
  apply force
apply clarify
  apply(erule-tac x=Suc ia in allE, simp)
apply(erule-tac x=0 and P= $\lambda j. H\ j \longrightarrow (J\ j) \notin \text{ctran}$  for  $H\ J$  in allE, simp)
done

```

```

lemma etran-or-ctran2 [rule-format]:
   $\forall i. \text{Suc } i < \text{length } x \longrightarrow x \in \text{cptn} \longrightarrow (x!i -c \rightarrow x!\text{Suc } i \longrightarrow \neg x!i -e \rightarrow x!\text{Suc } i)$ 
   $\vee (x!i -e \rightarrow x!\text{Suc } i \longrightarrow \neg x!i -c \rightarrow x!\text{Suc } i)$ 
apply(induct x)
  apply simp
apply clarify
apply(erule cptn.cases, simp)
  apply(case-tac i, simp+)
apply(case-tac i, simp)
  apply(force elim:etran.cases)
apply simp
done

```

```

lemma etran-or-ctran2-disjI1:
  [ $x \in \text{cptn}; \text{Suc } i < \text{length } x; x!i -c \rightarrow x!\text{Suc } i$ ]  $\Longrightarrow \neg x!i -e \rightarrow x!\text{Suc } i$ 
by(drule etran-or-ctran2, simp-all)

```

```

lemma etran-or-ctran2-disjI2:

```

$\llbracket x \in \text{cptn}; \text{Suc } i < \text{length } x; x!i -e \rightarrow x!\text{Suc } i \rrbracket \implies \neg x!i -c \rightarrow x!\text{Suc } i$
by(*drule etran-or-ctran2,simp-all*)

lemma *not-ctran-None2* [*rule-format*]:

$\llbracket (\text{None}, s) \# xs \in \text{cptn}; i < \text{length } xs \rrbracket \implies \neg ((\text{None}, s) \# xs) ! i -c \rightarrow xs ! i$
apply(*frule not-ctran-None,simp*)
apply(*case-tac i,simp*)
apply(*force elim:etranE*)
apply *simp*
apply(*rule etran-or-ctran2-disjI2,simp-all*)
apply(*force intro:tl-of-cptn-is-cptn*)
done

lemma *Ex-first-occurrence* [*rule-format*]: $P (n::\text{nat}) \longrightarrow (\exists m. P m \wedge (\forall i < m. \neg P i))$
apply(*rule nat-less-induct*)
apply *clarify*
apply(*case-tac $\forall m. m < n \longrightarrow \neg P m$*)
apply *auto*
done

lemma *stability* [*rule-format*]:

$\forall j k. x \in \text{cptn} \longrightarrow \text{stable } p \text{ rely} \longrightarrow j \leq k \longrightarrow k < \text{length } x \longrightarrow \text{snd}(x!j) \in p \longrightarrow$
 $(\forall i. (\text{Suc } i) < \text{length } x \longrightarrow$
 $(x!i -e \rightarrow x!(\text{Suc } i)) \longrightarrow (\text{snd}(x!i), \text{snd}(x!(\text{Suc } i))) \in \text{rely}) \longrightarrow$
 $(\forall i. j \leq i \wedge i < k \longrightarrow x!i -e \rightarrow x!\text{Suc } i) \longrightarrow \text{snd}(x!k) \in p \wedge \text{fst}(x!j) = \text{fst}(x!k)$
supply [*simproc del: defined-all*]
apply(*induct x*)
apply *clarify*
apply(*force elim:cptn.cases*)
apply *clarify*
apply(*erule cptn.cases,simp*)
apply *simp*
apply(*case-tac k,simp,simp*)
apply(*case-tac j,simp*)
apply(*erule-tac x=0 in allE*)
apply(*erule-tac x=nat and P= $\lambda j. (0 \leq j) \longrightarrow (J j)$ for J in allE,simp*)
apply(*subgoal-tac t ∈ p*)
apply(*subgoal-tac $(\forall i. i < \text{length } xs \longrightarrow ((P, t) \# xs) ! i -e \rightarrow xs ! i \longrightarrow (\text{snd}((P, t) \# xs) ! i), \text{snd}(xs ! i) \in \text{rely}))$*)
apply *clarify*
apply(*erule-tac x=Suc i and P= $\lambda j. (H j) \longrightarrow (J j) \in \text{etran}$ for H J in allE,simp*)
apply *clarify*
apply(*erule-tac x=Suc i and P= $\lambda j. (H j) \longrightarrow (J j) \longrightarrow (T j) \in \text{rely}$ for H J T in allE,simp*)
apply(*erule-tac x=0 and P= $\lambda j. (H j) \longrightarrow (J j) \in \text{etran} \longrightarrow T j$ for H J T in allE,simp*)
apply(*simp(no-asm-use) only:stable-def*)
apply(*erule-tac x=s in allE*)

```

apply(erule-tac  $x=t$  in  $allE$ )
apply simp
apply(erule mp)
apply(erule mp)
apply(rule Env)
apply simp
apply(erule-tac  $x=nata$  in  $allE$ )
apply(erule-tac  $x=nat$  and  $P=\lambda j. (s \leq j) \longrightarrow (J\ j)$  for  $s\ J$  in  $allE, simp$ )
apply(subgoal-tac  $(\forall i. i < length\ xs \longrightarrow ((P, t) \# xs) ! i -e\rightarrow xs ! i \longrightarrow (snd$ 
 $((P, t) \# xs) ! i), snd\ (xs ! i)) \in rely$ )
apply clarify
apply(erule-tac  $x=Suc\ i$  and  $P=\lambda j. (H\ j) \longrightarrow (J\ j) \in etran$  for  $H\ J$  in  $allE, simp$ )
apply clarify
apply(erule-tac  $x=Suc\ i$  and  $P=\lambda j. (H\ j) \longrightarrow (J\ j) \longrightarrow (T\ j) \in rely$  for  $H\ J\ T$ 
in  $allE, simp$ )
apply(case-tac  $k, simp, simp$ )
apply(case-tac  $j$ )
apply(erule-tac  $x=0$  and  $P=\lambda j. (H\ j) \longrightarrow (J\ j) \in etran$  for  $H\ J$  in  $allE, simp$ )
apply(erule etran.cases, simp)
apply(erule-tac  $x=nata$  in  $allE$ )
apply(erule-tac  $x=nat$  and  $P=\lambda j. (s \leq j) \longrightarrow (J\ j)$  for  $s\ J$  in  $allE, simp$ )
apply(subgoal-tac  $(\forall i. i < length\ xs \longrightarrow ((Q, t) \# xs) ! i -e\rightarrow xs ! i \longrightarrow (snd$ 
 $((Q, t) \# xs) ! i), snd\ (xs ! i)) \in rely$ )
apply clarify
apply(erule-tac  $x=Suc\ i$  and  $P=\lambda j. (H\ j) \longrightarrow (J\ j) \in etran$  for  $H\ J$  in  $allE, simp$ )
apply clarify
apply(erule-tac  $x=Suc\ i$  and  $P=\lambda j. (H\ j) \longrightarrow (J\ j) \longrightarrow (T\ j) \in rely$  for  $H\ J\ T$  in
 $allE, simp$ )
done

```

3.5.1 Soundness of the System for Component Programs

Soundness of the Basic rule

lemma *unique-ctran-Basic* [rule-format]:

$$\forall s\ i. x \in cptn \longrightarrow x ! 0 = (Some\ (Basic\ f), s) \longrightarrow$$

$$Suc\ i < length\ x \longrightarrow x!i -c\rightarrow x!Suc\ i \longrightarrow$$

$$(\forall j. Suc\ j < length\ x \longrightarrow i \neq j \longrightarrow x!j -e\rightarrow x!Suc\ j)$$

```

apply(induct  $x, simp$ )
apply simp
apply clarify
apply(erule  $cptn.cases, simp$ )
apply(case-tac  $i, simp+$ )
apply clarify
apply(case-tac  $j, simp$ )
apply(rule Env)
apply simp
apply clarify
apply simp
apply(case-tac  $i$ )

```

```

apply(case-tac j,simp,simp)
apply(erule ctran.cases,simp-all)
apply(force elim: not-ctran-None)
apply(ind-cases ((Some (Basic f), sa), Q, t) ∈ ctran for sa Q t)
apply simp
apply(drule-tac i=nat in not-ctran-None,simp)
apply(erule etranE,simp)
done

```

```

lemma exists-ctran-Basic-None [rule-format]:
   $\forall s\ i. x \in \text{cptn} \longrightarrow x!0 = (\text{Some } (\text{Basic } f), s)$ 
   $\longrightarrow i < \text{length } x \longrightarrow \text{fst}(x!i) = \text{None} \longrightarrow (\exists j < i. x!j -c \rightarrow x!\text{Suc } j)$ 
apply(induct x,simp)
apply simp
apply clarify
apply(erule cptn.cases,simp)
apply(case-tac i,simp,simp)
apply(erule-tac x=nat in allE,simp)
apply clarify
apply(rule-tac x=Suc j in exI,simp,simp)
apply clarify
apply(case-tac i,simp,simp)
apply(rule-tac x=0 in exI,simp)
done

```

```

lemma Basic-sound:
   $\llbracket \text{pre} \subseteq \{s. f\ s \in \text{post}\}; \{(s, t). s \in \text{pre} \wedge t = f\ s\} \subseteq \text{guar};$ 
   $\text{stable pre rely}; \text{stable post rely} \rrbracket$ 
   $\implies \models \text{Basic } f \text{ sat } [\text{pre}, \text{rely}, \text{guar}, \text{post}]$ 
supply [simproc del: defined-all]
apply(unfold com-validity-def)
apply clarify
apply(simp add:comm-def)
apply(rule conjI)
apply clarify
apply(simp add:cp-def assum-def)
apply clarify
apply(frule-tac j=0 and k=i and p=pre in stability)
  apply simp-all
  apply(erule-tac x=ia in allE,simp)
  apply(erule-tac i=i and f=f in unique-ctran-Basic,simp-all)
apply(erule subsetD,simp)
apply(case-tac x!i)
apply clarify
apply(drule-tac s=Some (Basic f) in sym,simp)
apply(thin-tac  $\forall j. H\ j$  for H)
apply(force elim:ctran.cases)
apply clarify
apply(simp add:cp-def)

```

```

apply clarify
apply(frule-tac  $i = \text{length } x - 1$  and  $f = f$  in exists-ctran-Basic-None, simp+)
  apply(case-tac  $x$ , simp+)
  apply(rule last-fst-esp, simp add:last-length)
  apply (case-tac  $x$ , simp+)
apply(simp add:assum-def)
apply clarify
apply(frule-tac  $j = 0$  and  $k = j$  and  $p = \text{pre}$  in stability)
  apply simp-all
  apply(erule-tac  $x = i$  in allE, simp)
  apply(erule-tac  $i = j$  and  $f = f$  in unique-ctran-Basic, simp-all)
apply(case-tac  $x!j$ )
apply clarify
apply simp
apply(drule-tac  $s = \text{Some } (\text{Basic } f)$  in sym, simp)
apply(case-tac  $x! \text{Suc } j$ , simp)
apply(rule ctran.cases, simp)
apply(simp-all)
apply(drule-tac  $c = sa$  in subsetD, simp)
apply clarify
apply(frule-tac  $j = \text{Suc } j$  and  $k = \text{length } x - 1$  and  $p = \text{post}$  in stability, simp-all)
  apply(case-tac  $x$ , simp+)
  apply(erule-tac  $x = i$  in allE)
apply(erule-tac  $i = j$  and  $f = f$  in unique-ctran-Basic, simp-all)
  apply arith+
apply(case-tac  $x$ )
apply(simp add:last-length) +
done

```

Soundness of the Await rule

```

lemma unique-ctran-Await [rule-format]:
   $\forall s \ i. \ x \in \text{cptn} \longrightarrow x!0 = (\text{Some } (\text{Await } b \ c), \ s) \longrightarrow$ 
   $\text{Suc } i < \text{length } x \longrightarrow x!i - c \rightarrow x! \text{Suc } i \longrightarrow$ 
   $(\forall j. \ \text{Suc } j < \text{length } x \longrightarrow i \neq j \longrightarrow x!j - e \rightarrow x! \text{Suc } j)$ 
apply(induct  $x$ , simp+)
apply clarify
apply(erule cptn.cases, simp)
  apply(case-tac  $i$ , simp+)
  apply clarify
  apply(case-tac  $j$ , simp)
  apply(rule Env)
  apply simp
apply clarify
apply simp
apply(case-tac  $i$ )
  apply(case-tac  $j$ , simp, simp)
  apply(erule ctran.cases, simp-all)
  apply(force elim: not-ctran-None)

```

apply(*ind-cases* ((*Some* (*Await* *b c*), *sa*), *Q*, *t*) $\in$ *ctran* **for** *sa Q t,simp*)
apply(*drule-tac* *i=nat in not-ctran-None,simp*)
apply(*erule etranE,simp*)
done

lemma *exists-ctran-Await-None* [*rule-format*]:
 $\forall s i. x \in \text{cptn} \longrightarrow x!0 = (\text{Some } (\text{Await } b c), s)$
 $\longrightarrow i < \text{length } x \longrightarrow \text{fst}(x!i) = \text{None} \longrightarrow (\exists j < i. x!j -c \rightarrow x! \text{Suc } j)$
apply(*induct x,simp+*)
apply *clarify*
apply(*erule cptn.cases,simp*)
apply(*case-tac i,simp+*)
apply(*erule-tac x=nat in allE,simp*)
apply *clarify*
apply(*rule-tac x=Suc j in exI,simp,simp*)
apply *clarify*
apply(*case-tac i,simp,simp*)
apply(*rule-tac x=0 in exI,simp*)
done

lemma *Star-imp-cptn*:
 $(P, s) -c* \rightarrow (R, t) \implies \exists l \in \text{cp } P s. (\text{last } l) = (R, t)$
 $\wedge (\forall i. \text{Suc } i < \text{length } l \longrightarrow !i -c \rightarrow ! \text{Suc } i)$
apply (*erule converse-rtrancl-induct2*)
apply(*rule-tac x=[(R,t)] in bexI*)
apply *simp*
apply(*simp add:cp-def*)
apply(*rule CptnOne*)
apply *clarify*
apply(*rule-tac x=(a, b)#l in bexI*)
apply (*rule conjI*)
apply(*case-tac l,simp add:cp-def*)
apply(*simp add:last-length*)
apply *clarify*
apply(*case-tac i,simp*)
apply(*simp add:cp-def*)
apply *force*
apply(*simp add:cp-def*)
apply(*case-tac l*)
apply(*force elim:cptn.cases*)
apply *simp*
apply(*erule CptnComp*)
apply *clarify*
done

lemma *Await-sound*:
 $\llbracket \text{stable pre rely}; \text{stable post rely};$
 $\forall V. \vdash P \text{ sat } [\text{pre} \cap b \cap \{s. s = V\}, \{(s, t). s = t\},$
 $\text{UNIV}, \{s. (V, s) \in \text{guar}\} \cap \text{post}] \wedge$

$\models P \text{ sat } [pre \cap b \cap \{s. s = V\}, \{(s, t). s = t\},$
 $\quad UNIV, \{s. (V, s) \in guar\} \cap post] \parallel$
 $\implies \models \text{Await } b \text{ } P \text{ sat } [pre, rely, guar, post]$
apply(*unfold com-validity-def*)
apply *clarify*
apply(*simp add:comm-def*)
apply(*rule conjI*)
apply *clarify*
apply(*simp add:cp-def assum-def*)
apply *clarify*
apply(*frule-tac j=0 and k=i and p=pre in stability,simp-all*)
apply(*erule-tac x=ia in allE,simp*)
apply(*subgoal-tac x ∈ cp (Some(Await b P)) s*)
apply(*erule-tac i=i in unique-ctran-Await,force,simp-all*)
apply(*simp add:cp-def*)
— here starts the different part.
apply(*erule ctran.cases,simp-all*)
apply(*drule Star-imp-cptn*)
apply *clarify*
apply(*erule-tac x=sa in allE*)
apply *clarify*
apply(*erule-tac x=sa in allE*)
apply(*drule-tac c=l in subsetD*)
apply (*simp add:cp-def*)
apply *clarify*
apply(*erule-tac x=ia and P=λi. H i ⟶ (J i, I i) ∈ ctran for H J I in allE,simp*)
apply(*erule etranE,simp*)
apply *simp*
apply *clarify*
apply(*simp add:cp-def*)
apply *clarify*
apply(*frule-tac i=length x - 1 in exists-ctran-Await-None,force*)
apply (*case-tac x,simp+*)
apply(*rule last-fst-esp,simp add:last-length*)
apply(*case-tac x, simp+*)
apply *clarify*
apply(*simp add:assum-def*)
apply *clarify*
apply(*frule-tac j=0 and k=j and p=pre in stability,simp-all*)
apply(*erule-tac x=i in allE,simp*)
apply(*erule-tac i=j in unique-ctran-Await,force,simp-all*)
apply(*case-tac x!j*)
apply *clarify*
apply *simp*
apply(*drule-tac s=Some (Await b P) in sym,simp*)
apply(*case-tac x!Suc j,simp*)
apply(*rule ctran.cases,simp*)
apply(*simp-all*)
apply(*drule Star-imp-cptn*)

```

apply clarify
apply(erule-tac  $x=sa$  in allE)
apply clarify
apply(erule-tac  $x=sa$  in allE)
apply(drule-tac  $c=l$  in subsetD)
  apply (simp add:cp-def)
  apply clarify
  apply(erule-tac  $x=i$  and  $P=\lambda i. H\ i \longrightarrow (J\ i, I\ i) \in ctran$  for  $H\ J\ I$  in allE, simp)
  apply(erule etranE, simp)
apply simp
apply clarify
apply(frule-tac  $j=Suc\ j$  and  $k=length\ x - 1$  and  $p=post$  in stability, simp-all)
  apply(case-tac  $x, simp+$ )
  apply(erule-tac  $x=i$  in allE)
apply(erule-tac  $i=j$  in unique-ctran-Await, force, simp-all)
  apply arith+
apply(case-tac  $x$ )
apply(simp add:last-length)+
done

```

Soundness of the Conditional rule

lemma *Cond-sound*:

```

   $\llbracket stable\ pre\ rely; \models P1\ sat\ [pre \cap b, rely, guar, post];$ 
   $\models P2\ sat\ [pre \cap -\ b, rely, guar, post]; \forall s. (s,s) \in guar \rrbracket$ 
   $\implies \models (Cond\ b\ P1\ P2)\ sat\ [pre, rely, guar, post]$ 
apply(unfold com-validity-def)
apply clarify
apply(simp add:cp-def comm-def)
apply(case-tac  $\exists i. Suc\ i < length\ x \wedge x!i - c \rightarrow x!Suc\ i$ )
  prefer 2
  apply simp
  apply clarify
  apply(frule-tac  $j=0$  and  $k=length\ x - 1$  and  $p=pre$  in stability, simp+)
    apply(case-tac  $x, simp+$ )
    apply(simp add:assum-def)
    apply(simp add:assum-def)
    apply(erule-tac  $m=length\ x$  in etran-or-ctran, simp+)
    apply(case-tac  $x, (simp\ add:last-length)+$ )
  apply(erule exE)
apply(drule-tac  $n=i$  and  $P=\lambda i. H\ i \wedge (J\ i, I\ i) \in ctran$  for  $H\ J\ I$  in Ex-first-occurrence)
apply clarify
apply (simp add:assum-def)
apply(frule-tac  $j=0$  and  $k=m$  and  $p=pre$  in stability, simp+)
  apply(erule-tac  $m=Suc\ m$  in etran-or-ctran, simp+)
apply(erule ctran.cases, simp-all)
apply(erule-tac  $x=sa$  in allE)
apply(drule-tac  $c=drop\ (Suc\ m)\ x$  in subsetD)
  apply simp

```

```

  apply clarify
  apply simp
  apply clarify
  apply(case-tac i≤m)
  apply(drule le-imp-less-or-eq)
  apply(erule disjE)
  apply(erule-tac x=i in allE, erule impE, assumption)
  apply simp+
  apply(erule-tac x=i - (Suc m) and P=λj. H j → J j → (I j)∈guar for H J
  I in allE)
  apply(subgoal-tac (Suc m)+(i - Suc m) ≤ length x)
  apply(subgoal-tac (Suc m)+Suc (i - Suc m) ≤ length x)
  apply(rotate-tac -2)
  apply simp
  apply arith
  apply arith
  apply(case-tac length (drop (Suc m) x),simp)
  apply(erule-tac x=sa in allE)
  back
  apply(drule-tac c=drop (Suc m) x in subsetD,simp)
  apply clarify
  apply simp
  apply clarify
  apply(case-tac i≤m)
  apply(drule le-imp-less-or-eq)
  apply(erule disjE)
  apply(erule-tac x=i in allE, erule impE, assumption)
  apply simp
  apply simp
  apply(erule-tac x=i - (Suc m) and P=λj. H j → J j → (I j)∈guar for H J I
  in allE)
  apply(subgoal-tac (Suc m)+(i - Suc m) ≤ length x)
  apply(subgoal-tac (Suc m)+Suc (i - Suc m) ≤ length x)
  apply(rotate-tac -2)
  apply simp
  apply arith
  apply arith
  done

```

Soundness of the Sequential rule

inductive-cases *Seq-cases* [elim!]: $(\text{Some } (\text{Seq } P \ Q), s) -c \rightarrow t$

lemma *last-lift-not-None*: $\text{fst } ((\text{lift } Q) ((x \# xs)!(\text{length } xs))) \neq \text{None}$
apply(subgoal-tac $\text{length } xs < \text{length } (x \# xs)$)
apply(drule-tac $Q = Q$ in lift-nth)
apply(erule ssubst)
apply (simp add:lift-def)
apply(case-tac $(x \# xs) ! \text{length } xs$,simp)

apply *simp*
done

lemma *Seq-sound1* [rule-format]:
 $x \in \text{cptn-mod} \implies \forall s P. x \neq (\text{Some } (\text{Seq } P \ Q), s) \longrightarrow$
 $(\forall i < \text{length } x. \text{fst}(x!i) \neq \text{Some } Q) \longrightarrow$
 $(\exists xs \in \text{cp } (\text{Some } P) \ s. x = \text{map } (\text{lift } Q) \ xs)$
supply [[*simproc* del: *defined-all*]]
apply(*erule* *cptn-mod.induct*)
apply(*unfold* *cp-def*)
apply *safe*
apply *simp-all*
 apply(*simp* *add:lift-def*)
 apply(*rule-tac* $x = [(\text{Some } Pa, sa)]$ **in** *exI, simp* *add:CptnOne*)
 apply(*subgoal-tac* $(\forall i < \text{Suc } (\text{length } xs). \text{fst } (((\text{Some } (\text{Seq } Pa \ Q), t) \# xs) ! i)$
 $\neq \text{Some } Q))$
 apply *clarify*
 apply(*rule-tac* $x = (\text{Some } Pa, sa) \# (\text{Some } Pa, t) \# zs$ **in** *exI, simp*)
 apply(*rule* *conjI,erule CptnEnv*)
 apply(*simp* (*no-asm-use*) *add:lift-def*)
 apply *clarify*
 apply(*erule-tac* $x = \text{Suc } i$ **in** *allE, simp*)
 apply(*ind-cases* $((\text{Some } (\text{Seq } Pa \ Q), sa), \text{None}, t) \in \text{ctran}$ **for** *Pa sa t*)
 apply(*rule-tac* $x = (\text{Some } P, sa) \# xs$ **in** *exI, simp* *add:cptn-iff-cptn-mod lift-def*)
 apply(*erule-tac* $x = \text{length } xs$ **in** *allE, simp*)
 apply(*simp* *only:Cons-lift-append*)
 apply(*subgoal-tac* $\text{length } xs < \text{length } ((\text{Some } P, sa) \# xs)$)
 apply(*simp* *only :nth-append length-map last-length nth-map*)
 apply(*case-tac* $\text{last}((\text{Some } P, sa) \# xs)$)
 apply(*simp* *add:lift-def*)
apply *simp*
done

lemma *Seq-sound2* [rule-format]:
 $x \in \text{cptn} \implies \forall s P i. x \neq (\text{Some } (\text{Seq } P \ Q), s) \longrightarrow i < \text{length } x$
 $\longrightarrow \text{fst}(x!i) = \text{Some } Q \longrightarrow$
 $(\forall j < i. \text{fst}(x!j) \neq (\text{Some } Q)) \longrightarrow$
 $(\exists xs \ ys. xs \in \text{cp } (\text{Some } P) \ s \wedge \text{length } xs = \text{Suc } i$
 $\wedge ys \in \text{cp } (\text{Some } Q) \ (\text{snd}(xs ! i)) \wedge x = (\text{map } (\text{lift } Q) \ xs) @ \text{tl } ys)$
supply [[*simproc* del: *defined-all*]]
apply(*erule* *cptn.induct*)
apply(*unfold* *cp-def*)
apply *safe*
apply *simp-all*
 apply(*case-tac* *i, simp+*)
 apply(*erule* *allE,erule impE,assumption,simp*)
 apply *clarify*
 apply(*subgoal-tac* $(\forall j < \text{nat}. \text{fst } (((\text{Some } (\text{Seq } Pa \ Q), t) \# xs) ! j) \neq \text{Some}$
 $Q), \text{clarify})$

```

  prefer 2
  apply force
  apply(case-tac xsa,simp,simp)
  apply(rename-tac list)
  apply(rule-tac x=(Some Pa, sa) # (Some Pa, t) # list in exI,simp)
  apply(rule conjI,erule CptnEnv)
  apply(simp (no-asm-use) add:lift-def)
  apply(rule-tac x=ys in exI,simp)
  apply(ind-cases ((Some (Seq Pa Q), sa), t) ∈ ctran for Pa sa t)
  apply simp
  apply(rule-tac x=(Some Pa, sa)#[(None, ta)] in exI,simp)
  apply(rule conjI)
  apply(drule-tac xs=[] in CptnComp,force simp add:CptnOne,simp)
  apply(case-tac i, simp+)
  apply(case-tac nat,simp+)
  apply(rule-tac x=(Some Q,ta)#xs in exI,simp add:lift-def)
  apply(case-tac nat,simp+)
  apply(force)
  apply(case-tac i, simp+)
  apply(case-tac nat,simp+)
  apply(erule-tac x=Suc nata in allE,simp)
  apply clarify
  apply(subgoal-tac (∀ j<Suc nata. fst (((Some (Seq P2 Q), ta) # xs) ! j) ≠ Some
Q),clarify)
  prefer 2
  apply clarify
  apply force
  apply(rule-tac x=(Some Pa, sa)#(Some P2, ta)#(tl xsa) in exI,simp)
  apply(rule conjI,erule CptnComp)
  apply(rule nth-tl-if,force,simp+)
  apply(rule-tac x=ys in exI,simp)
  apply(rule conjI)
  apply(rule nth-tl-if,force,simp+)
  apply(rule tl-zero,simp+)
  apply force
  apply(rule conjI,simp add:lift-def)
  apply(subgoal-tac lift Q (Some P2, ta) =(Some (Seq P2 Q), ta))
  apply(simp add:Cons-lift del:list.map)
  apply(rule nth-tl-if)
  apply force
  apply simp+
  apply(simp add:lift-def)
done

```

```

lemma last-lift-not-None2: fst ((lift Q) (last (x#xs))) ≠ None
apply(simp only:last-length [THEN sym])
apply(subgoal-tac length xs<length (x # xs))
apply(drule-tac Q=Q in lift-nth)

```

```

apply(erule ssubst)
apply (simp add:lift-def)
apply(case-tac (x # xs) ! length xs,simp)
apply simp
done

lemma Seq-sound:
  [| $\models P \text{ sat } [pre, rely, guar, mid]; \models Q \text{ sat } [mid, rely, guar, post]$ |]
   $\implies \models Seq P Q \text{ sat } [pre, rely, guar, post]$ 
apply(unfold com-validity-def)
apply clarify
apply(case-tac  $\exists i < \text{length } x. \text{fst}(x!i) = \text{Some } Q$ )
prefer 2
apply (simp add:cp-def cptn-iff-cptn-mod)
apply clarify
apply(frute-tac Seq-sound1,force)
  apply force
  apply clarify
  apply(erule-tac  $x=s$  in allE,simp)
  apply(drute-tac  $c=xs$  in subsetD,simp add:cp-def cptn-iff-cptn-mod)
  apply(simp add:assum-def)
  apply clarify
  apply(erule-tac  $P=\lambda j. H j \longrightarrow J j \longrightarrow I j$  for  $H J I$  in allE,erule impE,
assumption)
  apply(simp add:snd-lift)
  apply(erule mp)
  apply(force elim:etranE intro:Env simp add:lift-def)
  apply(simp add:comm-def)
  apply(rule conjI)
  apply clarify
  apply(erule-tac  $P=\lambda j. H j \longrightarrow J j \longrightarrow I j$  for  $H J I$  in allE,erule impE,
assumption)
  apply(simp add:snd-lift)
  apply(erule mp)
  apply(case-tac (xs!i))
  apply(case-tac (xs! Suc i))
  apply(case-tac fst(xs!i))
  apply(erule-tac  $x=i$  in allE, simp add:lift-def)
  apply(case-tac fst(xs!Suc i))
  apply(force simp add:lift-def)
  apply(force simp add:lift-def)
  apply clarify
  apply(case-tac xs,simp add:cp-def)
  apply clarify
  apply (simp del:list.map)
  apply (rename-tac list)
  apply(subgoal-tac (map (lift Q) ((a, b) # list)) $\neq []$ )
  apply(drute last-conv-nth)
  apply (simp del:list.map)

```

```

    apply(simp only:last-lift-not-None)
  apply simp
  —  $\exists i < \text{length } x. \text{fst } (x ! i) = \text{Some } Q$ 
  apply(erule exE)
  apply(drule-tac n=i and  $P = \lambda i. i < \text{length } x \wedge \text{fst } (x ! i) = \text{Some } Q$  in Ex-first-occurrence)
  apply clarify
  apply (simp add:cp-def)
  apply clarify
  apply(frule-tac i=m in Seq-sound2,force)
  apply simp+
  apply clarify
  apply(simp add:comm-def)
  apply(erule-tac x=s in allE)
  apply(drule-tac c=xs in subsetD,simp)
  apply(case-tac xs=[],simp)
  apply(simp add:cp-def assum-def nth-append)
  apply clarify
  apply(erule-tac x=i in allE)
  back
  apply(simp add:snd-lift)
  apply(erule mp)
  apply(force elim:etranE intro:Env simp add:lift-def)
  apply simp
  apply clarify
  apply(erule-tac x=snd(xs!m) in allE)
  apply(drule-tac c=ys in subsetD,simp add:cp-def assum-def)
  apply(case-tac xs≠[],)
  apply(drule last-conv-nth,simp)
  apply(rule conjI)
  apply(erule mp)
  apply(case-tac xs!m)
  apply(case-tac fst(xs!m),simp)
  apply(simp add:lift-def nth-append)
  apply clarify
  apply(erule-tac x=m+i in allE)
  back
  back
  apply(case-tac ys,(simp add:nth-append)+)
  apply (case-tac i, (simp add:snd-lift)+)
  apply(erule mp)
  apply(case-tac xs!m)
  apply(force elim:etran.cases intro:Env simp add:lift-def)
  apply simp
  apply simp
  apply clarify
  apply(rule conjI,clarify)
  apply(case-tac i<m,simp add:nth-append)
  apply(simp add:snd-lift)
  apply(erule allE, erule impE, assumption, erule mp)

```

```

apply(case-tac (xs ! i))
apply(case-tac (xs ! Suc i))
apply(case-tac fst(xs ! i),force simp add:lift-def)
apply(case-tac fst(xs ! Suc i))
  apply (force simp add:lift-def)
apply (force simp add:lift-def)
apply(erule-tac x=i-m in allE)
back
back
apply(subgoal-tac Suc (i - m) < length ys,simp)
prefer 2
apply arith
apply(simp add:nth-append snd-lift)
apply(rule conjI,clarify)
apply(subgoal-tac i=m)
prefer 2
apply arith
apply clarify
apply(simp add:cp-def)
apply(rule tl-zero)
  apply(erule mp)
  apply(case-tac lift Q (xs!m),simp add:snd-lift)
  apply(case-tac xs!m,case-tac fst(xs!m),simp add:lift-def snd-lift)
  apply(case-tac ys,simp+)
  apply(simp add:lift-def)
apply simp
apply force
apply clarify
apply(rule tl-zero)
apply(rule tl-zero)
  apply (subgoal-tac i-m=Suc(i-Suc m))
  apply simp
  apply(erule mp)
  apply(case-tac ys,simp+)
apply force
apply arith
apply force
apply clarify
apply(case-tac (map (lift Q) xs @ tl ys)≠[])
apply(drule last-conv-nth)
apply(simp add: snd-lift nth-append)
apply(rule conjI,clarify)
  apply(case-tac ys,simp+)
apply clarify
apply(case-tac ys,simp+)
done

```

Soundness of the While rule

lemma *last-append*[*rule-format*]:

$\forall xs. ys \neq [] \longrightarrow ((xs @ ys)!(length (xs @ ys) - (Suc 0))) = (ys!(length ys - (Suc 0)))$
apply (*induct* *ys*)
apply *simp*
apply *clarify*
apply (*simp add: nth-append*)
done

lemma *assum-after-body*:

$\llbracket \models P \text{ sat } [pre \cap b, \text{ rely}, \text{ guar}, pre];$
 $(Some\ P, s) \# xs \in \text{cptn-mod}; \text{fst}(\text{last}((Some\ P, s) \# xs)) = \text{None}; s \in b;$
 $(Some\ (While\ b\ P), s) \# (Some\ (Seq\ P\ (While\ b\ P)), s) \#$
 $\text{map}(\text{lift}\ (While\ b\ P))\ xs\ @\ ys \in \text{assum}\ (pre, \text{ rely}) \rrbracket$
 $\implies (Some\ (While\ b\ P), \text{snd}(\text{last}((Some\ P, s) \# xs))) \# ys \in \text{assum}\ (pre, \text{ rely})$
apply (*simp add: assum-def com-validity-def cp-def cptn-iff-cptn-mod*)
apply *clarify*
apply (*erule-tac x=s in allE*)
apply (*drule-tac c=(Some\ P, s) \# xs in subsetD, simp*)
apply *clarify*
apply (*erule-tac x=Suc\ i in allE*)
apply *simp*
apply (*simp add: Cons-lift-append nth-append snd-lift del: list.map*)
apply (*erule mp*)
apply (*erule etranE, simp*)
apply (*case-tac fst(((Some\ P, s) \# xs) ! i)*)
apply (*force intro: Env simp add: lift-def*)
apply (*force intro: Env simp add: lift-def*)
apply (*rule conjI*)
apply *clarify*
apply (*simp add: comm-def last-length*)
apply *clarify*
apply (*rule conjI*)
apply (*simp add: comm-def*)
apply *clarify*
apply (*erule-tac x=Suc(length\ xs + i) in allE, simp*)
apply (*case-tac i, simp add: nth-append Cons-lift-append snd-lift last-conv-nth lift-def split-def*)
apply (*simp add: Cons-lift-append nth-append snd-lift*)
done

lemma *While-sound-aux* [*rule-format*]:

$\llbracket pre \cap -\ b \subseteq post; \models P \text{ sat } [pre \cap b, \text{ rely}, \text{ guar}, pre]; \forall s. (s, s) \in \text{ guar};$
 $\text{stable}\ pre\ \text{ rely};\ \text{stable}\ post\ \text{ rely};\ x \in \text{cptn-mod} \rrbracket$
 $\implies \forall s\ xs. x = (Some\ (While\ b\ P), s) \# xs \longrightarrow x \in \text{assum}(pre, \text{ rely}) \longrightarrow x \in \text{comm}$
 $(\text{ guar}, \text{ post})$
supply [*simproc del: defined-all*]
apply (*erule cptn-mod.induct*)
apply *safe*

```

apply (simp-all del:last.simps)
— 5 subgoals left
apply(simp add:comm-def)
— 4 subgoals left
apply(rule etran-in-comm)
apply(erule mp)
apply(erule tl-of-assum-in-assum, simp)
— While-None
apply(ind-cases ((Some (While b P), s), None, t) in ctran for s t)
apply(simp add:comm-def)
apply(simp add:cptn-iff-cptn-mod [THEN sym])
apply(rule conjI, clarify)
  apply(force simp add:assum-def)
apply clarify
apply(rule conjI, clarify)
  apply(case-tac i, simp, simp)
  apply(force simp add:not-ctran-None2)
apply(subgoal-tac  $\forall i. \text{Suc } i < \text{length } ((\text{None}, t) \# xs) \longrightarrow (((\text{None}, t) \# xs) ! i, ((\text{None}, t) \# xs) ! \text{Suc } i) \in \text{etran}$ )
prefer 2
apply clarify
apply(rule-tac m=length ((None, s)  $\#$  xs) in etran-or-ctran, simp+)
apply(erule not-ctran-None2, simp)
apply simp+
apply(frule-tac j=0 and k=length ((None, s)  $\#$  xs)  $- 1$  and p=post in stability, simp+)
  apply(force simp add:assum-def subsetD)
  apply(simp add:assum-def)
  apply clarify
  apply(erule-tac x=i in allE, simp)
  apply(erule-tac x=Suc i in allE, simp)
  apply simp
apply clarify
apply (simp add:last-length)
— WhileOne
apply(rule ctran-in-comm, simp)
apply(simp add:Cons-lift del:list.map)
apply(simp add:comm-def del:list.map)
apply(rule conjI)
apply clarify
apply(case-tac fst((Some P, sa)  $\#$  xs) ! i)
apply(case-tac ((Some P, sa)  $\#$  xs) ! i)
apply (simp add:lift-def)
apply(ind-cases (Some (While b P), ba)  $-c \rightarrow t$  for ba t)
  apply simp
  apply simp
apply(simp add:snd-lift del:list.map)
apply(simp only:com-validity-def cp-def cptn-iff-cptn-mod)
apply(erule-tac x=sa in allE)

```

```

apply(drule-tac c=(Some P, sa) # xs in subsetD)
apply (simp add:assum-def del:list.map)
apply clarify
apply(erule-tac x=Suc ia in allE,simp add:snd-lift del:list.map)
apply(erule mp)
apply(case-tac fst((Some P, sa) # xs) ! ia)
  apply(erule etranE,simp add:lift-def)
  apply(rule Env)
apply(erule etranE,simp add:lift-def)
apply(rule Env)
apply (simp add:comm-def del:list.map)
apply clarify
apply(erule allE,erule impE,assumption)
apply(erule mp)
apply(case-tac ((Some P, sa) # xs) ! i)
apply(case-tac xs! i)
apply(simp add:lift-def)
apply(case-tac fst(xs! i))
  apply force
apply force
— last=None
apply clarify
apply(subgoal-tac (map (lift (While b P)) ((Some P, sa) # xs))≠[])
  apply(drule last-conv-nth)
  apply (simp del:list.map)
  apply(simp only:last-lift-not-None)
apply simp
— WhileMore
apply(rule ctran-in-comm,simp del:last.simps)
— metiendo la hipotesis antes de dividir la conclusion.
apply(subgoal-tac (Some (While b P), snd (last ((Some P, sa) # xs))) # ys ∈
assum (pre, rely))
  apply (simp del:last.simps)
prefer 2
apply(erule assum-after-body)
  apply (simp del:last.simps) +
— lo de antes.
apply(simp add:comm-def del:list.map last.simps)
apply(rule conjI)
apply clarify
apply(simp only:Cons-lift-append)
apply(case-tac i < length xs)
apply(simp add:nth-append del:list.map last.simps)
apply(case-tac fst((Some P, sa) # xs) ! i)
  apply(case-tac ((Some P, sa) # xs) ! i)
  apply (simp add:lift-def del:last.simps)
apply(ind-cases (Some (While b P), ba) — c → t for ba t)
  apply simp
apply simp

```

```

apply(simp add:snd-lift del:list.map last.simps)
apply(thin-tac  $\forall i. i < \text{length } ys \longrightarrow P\ i$  for  $P$ )
apply(simp only:com-validity-def cp-def cptn-iff-cptn-mod)
apply(erule-tac  $x=sa$  in  $allE$ )
apply(drule-tac  $c=(Some\ P, sa) \# xs$  in  $subsetD$ )
apply (simp add:assum-def del:list.map last.simps)
apply clarify
apply(erule-tac  $x=Suc\ ia$  in  $allE$ ,simp add:nth-append snd-lift del:list.map
last.simps, erule mp)
apply(case-tac fst(((Some  $P$ ,  $sa$ )  $\# xs$ ) !  $ia$ ))
apply(erule etranE,simp add:lift-def)
apply(rule Env)
apply(erule etranE,simp add:lift-def)
apply(rule Env)
apply (simp add:comm-def del:list.map)
apply clarify
apply(erule allE,erule impE,assumption)
apply(erule mp)
apply(case-tac ((Some  $P$ ,  $sa$ )  $\# xs$ ) !  $i$ )
apply(case-tac  $xs!i$ )
apply(simp add:lift-def)
apply(case-tac fst( $xs!i$ ))
apply force
apply force
—  $i \geq \text{length } xs$ 
apply(subgoal-tac  $i - \text{length } xs < \text{length } ys$ )
prefer 2
apply arith
apply(erule-tac  $x=i - \text{length } xs$  in  $allE$ ,clarify)
apply(case-tac  $i = \text{length } xs$ )
apply (simp add:nth-append snd-lift del:list.map last.simps)
apply(simp add:last-length del:last.simps)
apply(erule mp)
apply(case-tac last((Some  $P$ ,  $sa$ )  $\# xs$ ))
apply(simp add:lift-def del:last.simps)
—  $i > \text{length } xs$ 
apply(case-tac  $i - \text{length } xs$ )
apply arith
apply(simp add:nth-append del:list.map last.simps)
apply(rotate-tac  $-3$ )
apply(subgoal-tac  $i - Suc\ (\text{length } xs) = \text{nat}$ )
prefer 2
apply arith
apply simp
— last=None
apply clarify
apply(case-tac  $ys$ )
apply(simp add:Cons-lift del:list.map last.simps)
apply(subgoal-tac (map (lift (While  $b\ P$ )) ((Some  $P$ ,  $sa$ )  $\# xs$ )) $\neq []$ )

```

```

apply(drule last-conv-nth)
apply (simp del:list.map)
apply(simp only:last-lift-not-None)
apply simp
apply(subgoal-tac ((Some (Seq P (While b P))), sa) # map (lift (While b P)) xs
@ ys)≠[])
apply(drule last-conv-nth)
apply (simp del:list.map last.simps)
apply(simp add:nth-append del:last.simps)
apply(rename-tac a list)
apply(subgoal-tac ((Some (While b P), snd (last ((Some P, sa) # xs))) # a #
list)≠[])
apply(drule last-conv-nth)
apply (simp del:list.map last.simps)
apply simp
apply simp
done

```

lemma *While-sound*:

```

[[stable pre rely; pre ∩ − b ⊆ post; stable post rely;
  ⊢ P sat [pre ∩ b, rely, guar, pre]; ∀ s. (s,s)∈guar]
⇒ ⊢ While b P sat [pre, rely, guar, post]
apply(unfold com-validity-def)
apply clarify
apply(erule-tac xs=tl x in While-sound-aux)
apply(simp add:com-validity-def)
apply force
apply simp-all
apply(simp add:cptn-iff-cptn-mod cp-def)
apply(simp add:cp-def)
apply clarify
apply(rule nth-equalityI)
apply simp-all
apply(case-tac x,simp+)
apply(case-tac i,simp+)
apply(case-tac x,simp+)
done

```

Soundness of the Rule of Consequence

lemma *Conseq-sound*:

```

[[pre ⊆ pre'; rely ⊆ rely'; guar' ⊆ guar; post' ⊆ post;
  ⊢ P sat [pre', rely', guar', post']]
⇒ ⊢ P sat [pre, rely, guar, post]
apply(simp add:com-validity-def assum-def comm-def)
apply clarify
apply(erule-tac x=s in allE)
apply(drule-tac c=x in subsetD)
apply force

```

apply *force*
done

Soundness of the system for sequential component programs

theorem *rgsound*:

$\vdash P \text{ sat } [pre, rely, guar, post] \implies \models P \text{ sat } [pre, rely, guar, post]$
apply(*erule* *rghoare.induct*)
apply(*force* *elim:Basic-sound*)
apply(*force* *elim:Seq-sound*)
apply(*force* *elim:Cond-sound*)
apply(*force* *elim:While-sound*)
apply(*force* *elim:Await-sound*)
apply(*erule* *Conseq-sound,simp+*)
done

3.5.2 Soundness of the System for Parallel Programs

definition *ParallelCom* :: ('a *rgformula*) *list* $\Rightarrow$ 'a *par-com* **where**
ParallelCom *Ps* $\equiv$ *map* (*Some* $\circ$ *fst*) *Ps*

lemma *two*:

$$\begin{aligned} & \llbracket \forall i < \text{length } xs. \text{ rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. \text{ Guar } (xs ! j)) \\ & \quad \subseteq \text{ Rely } (xs ! i); \\ & \text{pre} \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. \text{ Pre } (xs ! i)); \\ & \forall i < \text{length } xs. \\ & \quad \models \text{ Com } (xs ! i) \text{ sat } [\text{Pre } (xs ! i), \text{ Rely } (xs ! i), \text{ Guar } (xs ! i), \text{ Post } (xs ! i)]; \\ & \quad \text{length } xs = \text{length } \text{clist}; x \in \text{par-cp } (\text{ParallelCom } xs) \text{ } s; x \in \text{par-assum}(\text{pre}, \text{rely}); \\ & \quad \forall i < \text{length } \text{clist}. \text{clist} ! i \in \text{cp } (\text{Some}(\text{Com}(xs ! i))) \text{ } s; x \propto \text{clist} \rrbracket \\ & \implies \forall j \text{ } i. i < \text{length } \text{clist} \wedge \text{Suc } j < \text{length } x \longrightarrow (\text{clist} ! i ! j) -c \longrightarrow (\text{clist} ! i ! \text{Suc } j) \\ & \longrightarrow (\text{snd}(\text{clist} ! i ! j), \text{snd}(\text{clist} ! i ! \text{Suc } j)) \in \text{Guar}(xs ! i) \end{aligned}$$

apply(*unfold* *par-cp-def*)

apply (*rule* *ccontr*)

— By contradiction:

apply *simp*

apply(*erule* *exE*)

— the first c-tran that does not satisfy the guarantee-condition is from σ -*i* at step *m*.

apply(*drule-tac* *n=j and P= $\lambda j. \exists i. H \text{ } i \text{ } j$ for H in Ex-first-occurrence*)

apply(*erule* *exE*)

apply *clarify*

— σ -*i* $\in A(\text{pre}, \text{rely-1})$

apply(*subgoal-tac* *take (Suc (Suc m)) (clist ! i) $\in$ assum(Pre(xs ! i), Rely(xs ! i))*)

— but this contradicts $\models \sigma$ -*i* *sat* [*pre-i, rely-i, guar-i, post-i*]

apply(*erule-tac* *x=i and P= $\lambda i. H \text{ } i \longrightarrow \models (J \text{ } i) \text{ sat } [I \text{ } i, K \text{ } i, M \text{ } i, N \text{ } i]$ for H J I K M N in allE,erule impE,assumption*)

apply(*simp* *add:com-validity-def*)

apply(*erule-tac* *x=s in allE*)

apply(*simp* *add:cp-def comm-def*)

apply(*drule-tac* *c=take (Suc (Suc m)) (clist ! i) in subsetD*)

```

apply simp
apply (blast intro: takecptn-is-cptn)
apply simp
apply clarify
apply(erule-tac  $x=m$  and  $P=\lambda j. I\ j \wedge J\ j \longrightarrow H\ j$  for  $I\ J\ H$  in  $allE$ )
apply (simp add:conjoin-def same-length-def)
apply(simp add:assum-def)
apply(rule conjI)
apply(erule-tac  $x=i$  and  $P=\lambda j. H\ j \longrightarrow I\ j \in cp\ (K\ j)\ (J\ j)$  for  $H\ I\ K\ J$  in  $allE$ )
apply(simp add:cp-def par-assum-def)
apply(erule-tac  $c=s$  in  $subsetD, simp$ )
apply simp
apply clarify
apply(erule-tac  $x=i$  and  $P=\lambda j. H\ j \longrightarrow M \cup \bigcup ((T\ j) \text{ ' } (S\ j)) \subseteq (L\ j)$  for  $H\ M\ S\ T\ L$  in  $allE$ )
apply simp
apply(erule subsetD)
apply simp
apply(simp add:conjoin-def compat-label-def)
apply clarify
apply(erule-tac  $x=ia$  and  $P=\lambda j. H\ j \longrightarrow (P\ j) \vee Q\ j$  for  $H\ P\ Q$  in  $allE, simp$ )
— each etran in  $\sigma-1[0..m]$  corresponds to
apply(erule disjE)
— a c-tran in some  $\sigma-\{ib\}$ 
apply clarify
apply(case-tac  $i=ib, simp$ )
apply(erule etranE, simp)
apply(erule-tac  $x=ib$  and  $P=\lambda i. H\ i \longrightarrow (I\ i) \vee (J\ i)$  for  $H\ I\ J$  in  $allE$ )
apply (erule etranE)
apply(case-tac  $ia=m, simp$ )
apply simp
apply(erule-tac  $x=ia$  and  $P=\lambda j. H\ j \longrightarrow (\forall i. P\ i\ j)$  for  $H\ P$  in  $allE$ )
apply(subgoal-tac  $ia < m, simp$ )
prefer 2
apply arith
apply(erule-tac  $x=ib$  and  $P=\lambda j. (I\ j, H\ j) \in ctran \longrightarrow P\ i\ j$  for  $I\ H\ P$  in  $allE, simp$ )
apply(simp add:same-state-def)
apply(erule-tac  $x=i$  and  $P=\lambda j. (T\ j) \longrightarrow (\forall i. (H\ j\ i) \longrightarrow (snd\ (d\ j\ i))=(snd\ (e\ j\ i)))$  for  $T\ H\ d\ e$  in  $all-dupE$ )
apply(erule-tac  $x=ib$  and  $P=\lambda j. (T\ j) \longrightarrow (\forall i. (H\ j\ i) \longrightarrow (snd\ (d\ j\ i))=(snd\ (e\ j\ i)))$  for  $T\ H\ d\ e$  in  $allE, simp$ )
— or an e-tran in  $\sigma$ , therefore it satisfies  $rely \vee guar-\{ib\}$ 
apply (force simp add:par-assum-def same-state-def)
done

```

lemma *three* [*rule-format*]:

$\llbracket xs \neq []; \forall i < \text{length } xs. rely \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. Guar\ (xs\ !\ j))$

$\subseteq \text{Rely } (xs ! i);$
 $pre \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. Pre (xs ! i));$
 $\forall i < \text{length } xs.$
 $\models \text{Com } (xs ! i) \text{ sat } [Pre (xs ! i), \text{Rely } (xs ! i), \text{Guar } (xs ! i), \text{Post } (xs ! i)];$
 $\text{length } xs = \text{length } clist; x \in \text{par-cp } (\text{ParallelCom } xs) s; x \in \text{par-assum}(pre, \text{rely});$
 $\forall i < \text{length } clist. clist!i \in cp (\text{Some}(\text{Com}(xs!i))) s; x \propto clist \parallel$
 $\implies \forall j i. i < \text{length } clist \wedge \text{Suc } j < \text{length } x \longrightarrow (clist!i!j) -e \longrightarrow (clist!i! \text{Suc } j)$
 $\longrightarrow (\text{snd}(clist!i!j), \text{snd}(clist!i! \text{Suc } j)) \in \text{rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}.$
 $\text{Guar } (xs ! j))$
apply(*drule two*)
apply *simp-all*
apply *clarify*
apply(*simp add: conjoin-def compat-label-def*)
apply *clarify*
apply(*erule-tac x=j and P= $\lambda j. H j \longrightarrow (J j \wedge (\exists i. P i j)) \vee I j$ for $H J P I$ in $allE, simp$*)
apply(*erule disjE*)
prefer 2
apply(*force simp add: same-state-def par-assum-def*)
apply *clarify*
apply(*case-tac i=ia, simp*)
apply(*erule etranE, simp*)
apply(*erule-tac x=ia and P= $\lambda i. H i \longrightarrow (I i) \vee (J i)$ for $H I J$ in $allE, simp$*)
apply(*erule-tac x=j and P= $\lambda j. \forall i. S j i \longrightarrow (I j i, H j i) \in \text{ctran} \longrightarrow P i j$ for $S I H P$ in $allE$*)
apply(*erule-tac x=ia and P= $\lambda j. S j \longrightarrow (I j, H j) \in \text{ctran} \longrightarrow P j$ for $S I H P$ in $allE$*)
apply(*simp add: same-state-def*)
apply(*erule-tac x=i and P= $\lambda j. T j \longrightarrow (\forall i. H j i \longrightarrow (\text{snd } (d j i)) = (\text{snd } (e j i)))$ for $T H d e$ in $all\text{-}dupE$*)
apply(*erule-tac x=ia and P= $\lambda j. T j \longrightarrow (\forall i. H j i \longrightarrow (\text{snd } (d j i)) = (\text{snd } (e j i)))$ for $T H d e$ in $allE, simp$*)
done

lemma four:

$\llbracket xs \neq [] \rrbracket; \forall i < \text{length } xs. \text{rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. \text{Guar } (xs ! j))$
 $\subseteq \text{Rely } (xs ! i);$
 $(\bigcup j \in \{j. j < \text{length } xs\}. \text{Guar } (xs ! j)) \subseteq \text{guar};$
 $pre \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. Pre (xs ! i));$
 $\forall i < \text{length } xs.$
 $\models \text{Com } (xs ! i) \text{ sat } [Pre (xs ! i), \text{Rely } (xs ! i), \text{Guar } (xs ! i), \text{Post } (xs ! i)];$
 $x \in \text{par-cp } (\text{ParallelCom } xs) s; x \in \text{par-assum } (pre, \text{rely}); \text{Suc } i < \text{length } x;$
 $x ! i -pc \rightarrow x ! \text{Suc } i \parallel$
 $\implies (\text{snd } (x ! i), \text{snd } (x ! \text{Suc } i)) \in \text{guar}$
apply(*simp add: ParallelCom-def*)
apply(*subgoal-tac (map (Some $\circ$ fst) xs) $\neq []$*)
prefer 2
apply *simp*
apply(*frule rev-subsetD*)

```

apply(erule one [THEN equalityD1])
apply(erule subsetD)
apply simp
apply clarify
apply(drule-tac pre=pre and rely=rely and x=x and s=s and xs=xs and clist=clist
in two)
apply(assumption+)
  apply(erule sym)
  apply(simp add:ParallelCom-def)
  apply assumption
  apply(simp add:Com-def)
  apply assumption
apply(simp add:conjoin-def same-program-def)
apply clarify
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow \text{fst}(I j) = (J j)$  for  $H I J$  in all-dupE)
apply(erule-tac x=Suc i and P= $\lambda j. H j \longrightarrow \text{fst}(I j) = (J j)$  for  $H I J$  in allE)
apply(erule par-ctranE,simp)
apply(erule-tac x=i and P= $\lambda j. \forall i. S j i \longrightarrow (I j i, H j i) \in \text{ctran} \longrightarrow P i j$  for
 $S I H P$  in allE)
apply(erule-tac x=ia and P= $\lambda j. S j \longrightarrow (I j, H j) \in \text{ctran} \longrightarrow P j$  for  $S I H P$ 
in allE)
apply(erule-tac x=ia in exI)
apply(simp add:same-state-def)
apply(erule-tac x=ia and P= $\lambda j. T j \longrightarrow (\forall i. H j i \longrightarrow (\text{snd } (d j i)) = (\text{snd } (e j i)))$  for  $T H d e$  in all-dupE,simp)
apply(erule-tac x=ia and P= $\lambda j. T j \longrightarrow (\forall i. H j i \longrightarrow (\text{snd } (d j i)) = (\text{snd } (e j i)))$  for  $T H d e$  in allE,simp)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in
all-dupE)
apply(erule-tac x=i and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in
all-dupE,simp)
apply(erule-tac x=Suc i and P= $\lambda j. H j \longrightarrow (\text{snd } (d j)) = (\text{snd } (e j))$  for  $H d e$  in
allE,simp)
apply(erule mp)
apply(subgoal-tac r=fst(clist ! ia ! Suc i),simp)
apply(drule-tac i=ia in list-eq-if)
back
apply simp-all
done

```

```

lemma parcptn-not-empty [simp]:  $\square \notin \text{par-cptn}$ 
apply(force elim:par-cptn.cases)
done

```

lemma five:

$$\begin{aligned}
& \llbracket xs \neq [] \rrbracket; \forall i < \text{length } xs. \text{ rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. \text{ Guar } (xs ! j)) \\
& \subseteq \text{ Rely } (xs ! i); \\
& \text{pre} \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. \text{ Pre } (xs ! i)); \\
& (\bigcap i \in \{i. i < \text{length } xs\}. \text{ Post } (xs ! i)) \subseteq \text{post};
\end{aligned}$$

$\forall i < \text{length } xs.$
 $\models \text{Com } (xs ! i) \text{ sat } [\text{Pre } (xs ! i), \text{Rely } (xs ! i), \text{Guar } (xs ! i), \text{Post } (xs ! i)];$
 $x \in \text{par-cp } (\text{ParallelCom } xs) \text{ s}; x \in \text{par-assum } (pre, rely);$
 $\text{All-None } (fst \text{ (last } x)) \models \text{snd } (last \text{ } x) \in post$
apply(simp add: ParallelCom-def)
apply(subgoal-tac (map (Some $\circ$ fst) xs) $\neq$ [])
prefer 2
apply simp
apply(frule rev-subsetD)
apply(erule one [THEN equalityD1])
apply(erule subsetD)
apply simp
apply clarify
apply(subgoal-tac $\forall i < \text{length } clist. clist!i \in \text{assum}(\text{Pre}(xs!i), \text{Rely}(xs!i))$)
apply(erule-tac $x=xa$ and $P=\lambda i. H \text{ } i \longrightarrow \models (J \text{ } i) \text{ sat } [I \text{ } i, K \text{ } i, M \text{ } i, N \text{ } i]$ for $H \text{ } J$
 $I \text{ } K \text{ } M \text{ } N$ in $allE, \text{erule impE}, \text{assumption}$)
apply(simp add: com-validity-def)
apply(erule-tac $x=s$ in $allE$)
apply(erule-tac $x=xa$ and $P=\lambda j. H \text{ } j \longrightarrow (I \text{ } j) \in \text{cp } (J \text{ } j) \text{ s}$ for $H \text{ } I \text{ } J$ in $allE, \text{simp}$)
apply(drule-tac $c=clist!xa$ in subsetD)
apply (force simp add: Com-def)
apply(simp add: comm-def conjoin-def same-program-def del: last.simps)
apply clarify
apply(erule-tac $x=\text{length } x - 1$ and $P=\lambda j. H \text{ } j \longrightarrow \text{fst}(I \text{ } j)=(J \text{ } j)$ for $H \text{ } I \text{ } J$ in
 $allE$)
apply (simp add: All-None-def same-length-def)
apply(erule-tac $x=xa$ and $P=\lambda j. H \text{ } j \longrightarrow \text{length}(J \text{ } j)=(K \text{ } j)$ for $H \text{ } J \text{ } K$ in $allE$)
apply(subgoal-tac $\text{length } x - 1 < \text{length } x, \text{simp}$)
apply(case-tac $x \neq []$)
apply(simp add: last-conv-nth)
apply(erule-tac $x=clist!xa$ in $allE$)
apply(simp add: same-state-def)
apply(subgoal-tac $clist!xa \neq []$)
apply(simp add: last-conv-nth)
apply(case-tac x)
apply (force simp add: par-cp-def)
apply (force simp add: par-cp-def)
apply force
apply (force simp add: par-cp-def)
apply(case-tac x)
apply (force simp add: par-cp-def)
apply (force simp add: par-cp-def)
apply clarify
apply(simp add: assum-def)
apply(rule conjI)
apply(simp add: conjoin-def same-state-def par-cp-def)
apply clarify
apply(erule-tac $x=i$ and $P=\lambda j. T \text{ } j \longrightarrow (\forall i. H \text{ } j \text{ } i \longrightarrow (\text{snd } (d \text{ } j \text{ } i))=(\text{snd } (e \text{ } j \text{ } i)))$ for $T \text{ } H \text{ } d \text{ } e$ in $allE, \text{simp}$)

```

apply(erule-tac  $x=0$  and  $P=\lambda j. H\ j \longrightarrow (\text{snd } (d\ j))=(\text{snd } (e\ j))$  for  $H\ d\ e$  in
allE)
apply(case-tac  $x, \text{simp}+$ )
apply (simp add:par-assum-def)
apply clarify
apply(erule-tac  $c=\text{snd } (\text{clist } !\ i\ !\ 0)$  in subsetD)
apply assumption
apply simp
apply clarify
apply(erule-tac  $x=i$  in all-dupE)
apply(rule subsetD, erule mp, assumption)
apply(erule-tac  $\text{pre}=\text{pre}$  and  $\text{rely}=\text{rely}$  and  $x=x$  and  $s=s$  in three)
apply(erule-tac  $x=ib$  in allE, erule mp)
apply simp-all
apply(simp add:ParallelCom-def)
apply(force simp add:Com-def)
apply(simp add:conjoin-def same-length-def)
done

```

```

lemma ParallelEmpty [rule-format]:
   $\forall i\ s. x \in \text{par-cp } (\text{ParallelCom } [])\ s \longrightarrow$ 
   $\text{Suc } i < \text{length } x \longrightarrow (x\ !\ i, x\ !\ \text{Suc } i) \notin \text{par-ctran}$ 
apply(induct-tac  $x$ )
apply(simp add:par-cp-def ParallelCom-def)
apply clarify
apply(case-tac  $\text{list}, \text{simp}, \text{simp}$ )
apply(case-tac  $i$ )
apply(simp add:par-cp-def ParallelCom-def)
apply(erule par-ctranE, simp)
apply(simp add:par-cp-def ParallelCom-def)
apply clarify
apply(erule par-cptn.cases, simp)
apply simp
apply(erule par-ctranE)
back
apply simp
done

```

```

theorem par-rgsound:
   $\vdash c\ \text{SAT } [\text{pre}, \text{rely}, \text{guar}, \text{post}] \implies$ 
   $\models (\text{ParallelCom } c)\ \text{SAT } [\text{pre}, \text{rely}, \text{guar}, \text{post}]$ 
apply(erule par-rghoare.induct)
apply(case-tac  $xs, \text{simp}$ )
apply(simp add:par-com-validity-def par-comm-def)
apply clarify
apply(case-tac  $\text{post}=\text{UNIV}, \text{simp}$ )
apply clarify
apply(erule ParallelEmpty)
apply assumption

```

```

    apply simp
    apply clarify
    apply simp
    apply(subgoal-tac xs≠[])
    prefer 2
    apply simp
    apply(rename-tac a list)
    apply(thin-tac xs = a # list)
    apply(simp add:par-com-validity-def par-comm-def)
    apply clarify
    apply(rule conjI)
    apply clarify
    apply(erule-tac pre=pre and rely=rely and guar=guar and x=x and s=s and
xs=xs in four)
      apply(assumption+)
      apply clarify
      apply (erule allE, erule impE, assumption,erule rgsound)
      apply(assumption+)
    apply clarify
    apply(erule-tac pre=pre and rely=rely and post=post and x=x and s=s and
xs=xs in five)
      apply(assumption+)
      apply clarify
      apply (erule allE, erule impE, assumption,erule rgsound)
      apply(assumption+)
    done
end

```

3.6 Concrete Syntax

```

theory RG-Syntax
imports RG-Hoare Quote-Antiquote
begin

```

```

abbreviation Skip :: 'a com (SKIP)
  where SKIP ≡ Basic id

```

```

notation Seq ((-;;/ -) [60,61] 60)

```

```

syntax

```

```

  -Assign    :: idt ⇒ 'b ⇒ 'a com                (( ' - :=/ -) [70, 65] 61)
  -Cond      :: 'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a com  ((0IF -/ THEN -/ ELSE
-/FI) [0, 0, 0] 61)
  -Cond2     :: 'a bexp ⇒ 'a com ⇒ 'a com            ((0IF - THEN - FI) [0,0] 56)
  -While     :: 'a bexp ⇒ 'a com ⇒ 'a com            ((0WHILE - /DO - /OD) [0,
0] 61)
  -Await     :: 'a bexp ⇒ 'a com ⇒ 'a com            ((0AWAIT - /THEN - /END)
[0,0] 61)

```

$-Atom \quad :: 'a \text{ com} \Rightarrow 'a \text{ com} \quad ((\langle - \rangle) \ 61)$
 $-Wait \quad :: 'a \text{ bexp} \Rightarrow 'a \text{ com} \quad ((0WAIT - END) \ 61)$

translations

$'x := a \rightarrow CONST \text{ Basic } \langle '(-update-name \ x \ (\lambda -. \ a)) \rangle$
 $IF \ b \ THEN \ c1 \ ELSE \ c2 \ FI \rightarrow CONST \text{ Cond } \langle \{b\} \ c1 \ c2 \rangle$
 $IF \ b \ THEN \ c \ FI \rightleftharpoons IF \ b \ THEN \ c \ ELSE \ SKIP \ FI$
 $WHILE \ b \ DO \ c \ OD \rightarrow CONST \text{ While } \langle \{b\} \ c \rangle$
 $AWAIT \ b \ THEN \ c \ END \rightleftharpoons CONST \text{ Await } \langle \{b\} \ c \rangle$
 $\langle c \rangle \rightleftharpoons AWAIT \ CONST \ True \ THEN \ c \ END$
 $WAIT \ b \ END \rightleftharpoons AWAIT \ b \ THEN \ SKIP \ END$

nonterminal prgs

syntax

$-PAR \quad :: prgs \Rightarrow 'a \quad (COBEGIN // - // COEND \ 60)$
 $-prg \quad :: 'a \Rightarrow prgs \quad (- \ 57)$
 $-prgs \quad :: ['a, prgs] \Rightarrow prgs \quad (- // || // - \ [60, 57] \ 57)$

translations

$-prg \ a \rightarrow [a]$
 $-prgs \ a \ ps \rightarrow a \ \# \ ps$
 $-PAR \ ps \rightarrow ps$

syntax

$-prg-scheme \ :: ['a, 'a, 'a, 'a] \Rightarrow prgs \quad (SCHEME \ [- \leq - < -] - \ [0, 0, 0, 60] \ 57)$

translations

$-prg-scheme \ j \ i \ k \ c \rightleftharpoons (CONST \ map \ (\lambda i. \ c) \ [j..<k])$

Translations for variables before and after a transition:

syntax

$-before \ :: id \Rightarrow 'a \ (^o-)$
 $-after \ :: id \Rightarrow 'a \ (^a-)$

translations

$^o x \rightleftharpoons x \ ' \ CONST \ fst$
 $^a x \rightleftharpoons x \ ' \ CONST \ snd$

print-translation <

let
 $fun \ quote-tr' \ f \ (t \ :: \ ts) =$
 $\quad Term.list-comb \ (f \ \$ \ Syntax-Trans.quote-tr' \ \textbf{syntax-const} \ \langle -antiquote \rangle \ t,$
 $ts)$
 $\quad | \ quote-tr' \ - \ = \ raise \ Match;$
 $val \ assert-tr' = quote-tr' \ (Syntax.const \ \textbf{syntax-const} \ \langle -Assert \rangle);$
 $fun \ bexp-tr' \ name \ ((Const \ (\textbf{const-syntax} \ \langle Collect \rangle, \ -) \ \$ \ t) \ :: \ ts) =$

```

      quote-tr' (Syntax.const name) (t :: ts)
    | bexp-tr' - - = raise Match;

  fun assign-tr' (Abs (x, -, f $ k $ Bound 0) :: ts) =
    quote-tr' (Syntax.const syntax-const <-Assign> $ Syntax-Trans.update-name-tr'
f)
      (Abs (x, dummyT, Syntax-Trans.const-abs-tr' k) :: ts)
    | assign-tr' - - = raise Match;
  in
    [(const-syntax <Collect>, K assert-tr'),
     (const-syntax <Basic>, K assign-tr'),
     (const-syntax <Cond>, K (bexp-tr' syntax-const <-Cond>)),
     (const-syntax <While>, K (bexp-tr' syntax-const <-While>))]
  end
>

end

```

3.7 Examples

```

theory RG-Examples
imports RG-Syntax
begin

```

```

lemmas definitions [simp] = stable-def Pre-def Rely-def Guar-def Post-def Com-def

```

3.7.1 Set Elements of an Array to Zero

```

lemma le-less-trans2:  $\llbracket (j::nat) < k; i \leq j \rrbracket \implies i < k$ 
by simp

```

```

lemma add-le-less-mono:  $\llbracket (a::nat) < c; b \leq d \rrbracket \implies a + b < c + d$ 
by simp

```

```

record Example1 =
  A :: nat list

```

```

lemma Example1:
  ⊢ COBEGIN
    SCHEME [0 ≤ i < n]
    ( 'A := 'A [i := 0],
      { n < length 'A },
      { length °A = length °A ∧ °A ! i = °A ! i },
      { length °A = length °A ∧ (∀ j < n. i ≠ j ⟶ °A ! j = °A ! j) },
      { 'A ! i = 0 } )
  COEND
  SAT [{ n < length 'A }, { °A = °A }, { True }, { ∀ i < n. 'A ! i = 0 }]
apply(rule Parallel)
apply(auto intro!: Basic)

```

done

lemma *Example1-parameterized:*

$k < t \implies$

⊢ COBEGIN

SCHEME $[k*n \leq i < (Suc\ k)*n]$ ($\text{' } A := \text{' } A[i := 0]$,

$\{\{t*n < length\ \text{' } A\},$

$\{\{t*n < length\ ^o A \wedge length\ ^o A = length\ ^a A \wedge ^o A!i = ^a A!i\},$

$\{\{t*n < length\ ^o A \wedge length\ ^o A = length\ ^a A \wedge (\forall j < length\ ^o A . i \neq j \longrightarrow ^o A!j = ^a A!j)\},$

$\{\{\text{' } A!i = 0\}\})$

COEND

SAT $[\{\{t*n < length\ \text{' } A\},$

$\{\{t*n < length\ ^o A \wedge length\ ^o A = length\ ^a A \wedge (\forall i < n. ^o A!(k*n+i) = ^a A!(k*n+i))\},$

$\{\{t*n < length\ ^o A \wedge length\ ^o A = length\ ^a A \wedge$

$(\forall i < length\ ^o A . (i < k*n \longrightarrow ^o A!i = ^a A!i) \wedge ((Suc\ k)*n \leq i \longrightarrow ^o A!i = ^a A!i))\},$

$\{\{\forall i < n. \text{' } A!(k*n+i) = 0\}\}]$

apply(rule Parallel)

 apply auto

 apply(erule-tac $x = k*n + i$ in allE)

 apply(subgoal-tac $k*n + i < length\ (A\ b)$)

 apply force

 apply(erule le-less-trans2)

 apply(case-tac t,simp+)

 apply (simp add:add commute)

 apply(simp add: add-le-mono)

apply(rule Basic)

 apply simp

 apply clarify

 apply (subgoal-tac $k*n + i < length\ (A\ x)$)

 apply simp

 apply(erule le-less-trans2)

 apply(case-tac t,simp+)

 apply (simp add:add commute)

 apply(rule add-le-mono, auto)

done

3.7.2 Increment a Variable in Parallel

Two components

record *Example2* =

$x :: nat$

$c-0 :: nat$

$c-1 :: nat$

lemma *Example2:*

⊢ COBEGIN

($\langle \text{' } x := \text{' } x + 1;; \text{' } c-0 := \text{' } c-0 + 1 \rangle,$

$\{\{\text{' } x = \text{' } c-0 + \text{' } c-1 \wedge \text{' } c-0 = 0\}\},$

$$\begin{aligned}
& \{ \{^{\circ}c-0 = {}^a c-0 \wedge \\
& \quad ({}^{\circ}x = {}^{\circ}c-0 + {}^{\circ}c-1 \\
& \quad \longrightarrow {}^a x = {}^a c-0 + {}^a c-1) \} \}, \\
& \{ \{^{\circ}c-1 = {}^a c-1 \wedge \\
& \quad ({}^{\circ}x = {}^{\circ}c-0 + {}^{\circ}c-1 \\
& \quad \longrightarrow {}^a x = {}^a c-0 + {}^a c-1) \} \}, \\
& \{ \{x = {}^{\circ}c-0 + {}^{\circ}c-1 \wedge {}^{\circ}c-0 = 1 \} \} \\
& \parallel \\
& \quad (\langle {}^{\circ}x := {}^{\circ}x + 1 ;; {}^{\circ}c-1 := {}^{\circ}c-1 + 1 \rangle, \\
& \quad \{ \{x = {}^{\circ}c-0 + {}^{\circ}c-1 \wedge {}^{\circ}c-1 = 0 \} \}, \\
& \quad \{ \{^{\circ}c-1 = {}^a c-1 \wedge \\
& \quad \quad ({}^{\circ}x = {}^{\circ}c-0 + {}^{\circ}c-1 \\
& \quad \quad \longrightarrow {}^a x = {}^a c-0 + {}^a c-1) \} \}, \\
& \quad \{ \{^{\circ}c-0 = {}^a c-0 \wedge \\
& \quad \quad ({}^{\circ}x = {}^{\circ}c-0 + {}^{\circ}c-1 \\
& \quad \quad \longrightarrow {}^a x = {}^a c-0 + {}^a c-1) \} \}, \\
& \quad \{ \{x = {}^{\circ}c-0 + {}^{\circ}c-1 \wedge {}^{\circ}c-1 = 1 \} \}) \\
& \text{COEND} \\
& \text{SAT } [\{ \{x = 0 \wedge {}^{\circ}c-0 = 0 \wedge {}^{\circ}c-1 = 0 \} \}, \\
& \quad \{ \{^{\circ}x = {}^a x \wedge {}^{\circ}c-0 = {}^a c-0 \wedge {}^{\circ}c-1 = {}^a c-1 \} \}, \\
& \quad \{ \text{True} \}, \\
& \quad \{ \{x = 2 \} \}] \\
& \text{apply}(\text{rule Parallel}) \\
& \quad \text{apply simp-all} \\
& \quad \text{apply clarify} \\
& \quad \text{apply(case-tac i)} \\
& \quad \text{apply simp} \\
& \quad \text{apply(rule conjI)} \\
& \quad \text{apply clarify} \\
& \quad \text{apply simp} \\
& \quad \text{apply clarify} \\
& \quad \text{apply simp} \\
& \quad \text{apply simp} \\
& \quad \text{apply(rule conjI)} \\
& \quad \text{apply clarify} \\
& \quad \text{apply simp} \\
& \quad \text{apply clarify} \\
& \quad \text{apply simp} \\
& \quad \text{apply(subgoal-tac xa=0)} \\
& \quad \text{apply simp} \\
& \quad \text{apply arith} \\
& \quad \text{apply clarify} \\
& \quad \text{apply(case-tac xaa, simp, simp)} \\
& \quad \text{apply clarify} \\
& \quad \text{apply simp} \\
& \quad \text{apply(erule-tac x=0 in all-dupE)} \\
& \quad \text{apply(erule-tac x=1 in allE,simp)} \\
& \quad \text{apply clarify} \\
& \quad \text{apply(case-tac i,simp)}
\end{aligned}$$

```

apply(rule Await)
  apply simp-all
apply(clarify)
apply(rule Seq)
  prefer 2
  apply(rule Basic)
  apply simp-all
  apply(rule subset-refl)
apply(rule Basic)
apply simp-all
apply clarify
apply simp
apply(rule Await)
  apply simp-all
apply(clarify)
apply(rule Seq)
  prefer 2
  apply(rule Basic)
  apply simp-all
  apply(rule subset-refl)
apply(auto intro!: Basic)
done

```

Parameterized

```

lemma Example2-lemma2-aux:  $j < n \implies$ 
   $(\sum_{i=0..<n. (b \ i::nat)) =$ 
   $(\sum_{i=0..<j. b \ i} + b \ j + (\sum_{i=0..<n-(Suc \ j). b \ (Suc \ j + i)})$ 
apply(induct n)
  apply simp-all
apply(simp add:less-Suc-eq)
  apply(auto)
apply(subgoal-tac  $n - j = Suc(n - Suc \ j)$ )
  apply simp
apply arith
done

```

```

lemma Example2-lemma2-aux2:
   $j \leq s \implies (\sum_{i::nat=0..<j. (b \ (s:=t)) \ i} = (\sum_{i=0..<j. b \ i})$ 
  by (induct j) simp-all

```

```

lemma Example2-lemma2:
   $\llbracket j < n; b \ j = 0 \rrbracket \implies Suc \ (\sum_{i::nat=0..<n. b \ i) = (\sum_{i=0..<n. (b \ (j := Suc \ 0)) \ i)$ 
apply(frule-tac  $b = (b \ (j := (Suc \ 0)))$ ) in Example2-lemma2-aux)
apply(erule-tac  $t = sum \ (b \ (j := (Suc \ 0))) \ \{0..<n\}$ ) in ssubst)
apply(frule-tac  $b = b$ ) in Example2-lemma2-aux)
apply(erule-tac  $t = sum \ b \ \{0..<n\}$ ) in ssubst)
apply(subgoal-tac  $Suc \ (sum \ b \ \{0..<j\} + b \ j + (\sum_{i=0..<n - Suc \ j. b \ (Suc \ j + i))) = (sum \ b \ \{0..<j\} + Suc \ (b \ j) + (\sum_{i=0..<n - Suc \ j. b \ (Suc \ j + i)))$ )

```

```

apply(rotate-tac -1)
apply(erule ssubst)
apply(subgoal-tac j≤j)
  apply(drule-tac b=b and t=(Suc 0) in Example2-lemma2-aux2)
apply(rotate-tac -1)
apply(erule ssubst)
apply simp-all
done

lemma Example2-lemma2-Suc0:  $\llbracket j < n; b \ j = 0 \rrbracket \implies$ 
   $Suc \ (\sum i::nat=0..< n. b \ i) = (\sum i=0..< n. (b \ (j:=Suc \ 0)) \ i)$ 
by(simp add:Example2-lemma2)

record Example2-parameterized =
  C :: nat  $\Rightarrow$  nat
  y :: nat

lemma Example2-parameterized:  $0 < n \implies$ 
   $\vdash COBEGIN \ SCHEME \ [0 \leq i < n]$ 
    ( $\langle \ 'y := 'y + 1;; \ 'C := 'C \ (i:=1) \ \rangle,$ 
 $\{\ 'y = (\sum i=0..< n. \ 'C \ i) \wedge \ 'C \ i = 0 \},$ 
 $\{\ ^oC \ i = ^aC \ i \wedge$ 
 $(^oy = (\sum i=0..< n. \ ^oC \ i) \longrightarrow ^ay = (\sum i=0..< n. \ ^aC \ i)) \},$ 
 $\{\ (\forall j < n. \ i \neq j \longrightarrow ^oC \ j = ^aC \ j) \wedge$ 
 $(^oy = (\sum i=0..< n. \ ^oC \ i) \longrightarrow ^ay = (\sum i=0..< n. \ ^aC \ i)) \},$ 
 $\{\ 'y = (\sum i=0..< n. \ 'C \ i) \wedge \ 'C \ i = 1 \}$ )
  COEND
  SAT  $[\{\ 'y = 0 \wedge (\sum i=0..< n. \ 'C \ i) = 0 \}, \{\ ^oC = ^aC \wedge ^oy = ^ay \}, \{ True \}, \{\ 'y = n \}]$ 
apply(rule Parallel)
apply force
apply force
apply(force)
apply clarify
apply simp
apply simp
apply clarify
apply simp
apply(rule Await)
apply simp-all
apply clarify
apply(rule Seq)
prefer 2
apply(rule Basic)
apply(rule subset-refl)
apply simp+
apply(rule Basic)
apply simp
apply clarify
apply simp

```

```

apply(simp add:Example2-lemma2-Suc0 cong:if-cong)
apply simp-all
done

```

3.7.3 Find Least Element

A previous lemma:

```

lemma mod-aux :  $\llbracket i < (n::nat); a \bmod n = i; j < a + n; j \bmod n = i; a < j \rrbracket \implies$ 
  False
apply(subgoal-tac a=a div n*n + a mod n )
  prefer 2 apply (simp (no-asm-use))
apply(subgoal-tac j=j div n*n + j mod n)
  prefer 2 apply (simp (no-asm-use))
apply simp
apply(subgoal-tac a div n*n < j div n*n)
prefer 2 apply arith
apply(subgoal-tac j div n*n < (a div n + 1)*n)
prefer 2 apply simp
apply (simp only:mult-less-cancel2)
apply arith
done

```

```

record Example3 =
  X :: nat  $\Rightarrow$  nat
  Y :: nat  $\Rightarrow$  nat

```

```

lemma Example3:  $m \bmod n = 0 \implies$ 
   $\vdash$  COBEGIN
    SCHEME  $[0 \leq i < n]$ 
    ( WHILE  $(\forall j < n. 'X\ i < 'Y\ j)$  DO
      IF  $P(B!( 'X\ i))$  THEN  $'Y := 'Y\ (i := 'X\ i)$ 
      ELSE  $'X := 'X\ (i := ('X\ i) + n)$  FI
    OD,
     $\llbracket ('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < m \longrightarrow P(B!( 'Y\ i))) \wedge 'Y\ i \leq m+i \rrbracket$ ,
     $\llbracket (\forall j < n. i \neq j \longrightarrow {}^aY\ j \leq {}^oY\ j) \wedge {}^oX\ i = {}^aX\ i \wedge$ 
       ${}^oY\ i = {}^aY\ i \rrbracket$ ,
     $\llbracket (\forall j < n. i \neq j \longrightarrow {}^oX\ j = {}^aX\ j \wedge {}^oY\ j = {}^aY\ j) \wedge$ 
       ${}^aY\ i \leq {}^oY\ i \rrbracket$ ,
     $\llbracket ('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < m \longrightarrow P(B!( 'Y\ i))) \wedge 'Y\ i \leq m+i \wedge (\exists j < n. 'Y\ j \leq 'X\ i) \rrbracket$ 
    COEND
    SAT  $[\llbracket \forall i < n. 'X\ i = i \wedge 'Y\ i = m+i \rrbracket, \llbracket {}^oX = {}^aX \wedge {}^oY = {}^aY \rrbracket, \llbracket True \rrbracket,$ 
       $\llbracket \forall i < n. ('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge$ 
       $('Y\ i < m \longrightarrow P(B!( 'Y\ i))) \wedge 'Y\ i \leq m+i \wedge (\exists j < n. 'Y\ j \leq 'X\ i) \rrbracket]$ 
apply(rule Parallel)
  — 5 subgoals left
apply force+
apply clarify

```

```

apply simp
apply(rule While)
  apply force
  apply force
  apply force
  apply (erule dvdE)
  apply(rule-tac pre'= $\llbracket \text{' } X \text{ } i \text{ } \text{mod } n = i \wedge (\forall j. j < \text{' } X \text{ } i \longrightarrow j \text{ } \text{mod } n = i \longrightarrow \neg P(B!j)) \wedge (\text{' } Y \text{ } i < n * k \longrightarrow P(B!(\text{' } Y \text{ } i))) \wedge \text{' } X \text{ } i < \text{' } Y \text{ } i \rrbracket$  in Conseq)
    apply force
    apply(rule subset-refl)+
apply(rule Cond)
  apply force
  apply(rule Basic)
  apply force
  apply fastforce
  apply force
  apply force
apply(rule Basic)
  apply simp
  apply clarify
  apply simp
  apply (case-tac  $X \text{ } x \text{ } (j \text{ } \text{mod } n) \leq j$ )
  apply (drule le-imp-less-or-eq)
  apply (erule disjE)
  apply (drule-tac  $j=j$  and  $n=n$  and  $i=j \text{ } \text{mod } n$  and  $a=X \text{ } x \text{ } (j \text{ } \text{mod } n)$  in mod-aux)
  apply auto
done

```

Same but with a list as auxiliary variable:

```

record Example3-list =
   $X :: \text{nat list}$ 
   $Y :: \text{nat list}$ 

```

lemma *Example3-list*: $m \text{ } \text{mod } n = 0 \implies \vdash (\text{COBEGIN SCHEME } [0 \leq i < n]$
 $(\text{WHILE } (\forall j < n. \text{' } X!i < \text{' } Y!j) \text{ DO}$
 $\text{IF } P(B!(\text{' } X!i)) \text{ THEN } \text{' } Y := \text{' } Y[i := \text{' } X!i] \text{ ELSE } \text{' } X := \text{' } X[i := (\text{' } X!i) + n] \text{ FI}$
 OD,
 $\llbracket n < \text{length } \text{' } X \wedge n < \text{length } \text{' } Y \wedge (\text{' } X!i) \text{ } \text{mod } n = i \wedge (\forall j < \text{' } X!i. j \text{ } \text{mod } n = i \longrightarrow \neg P(B!j)) \wedge (\text{' } Y!i < m \longrightarrow P(B!(\text{' } Y!i)) \wedge \text{' } Y!i \leq m + i) \rrbracket,$
 $\llbracket (\forall j < n. i \neq j \longrightarrow \text{' } Y!j \leq \text{' } Y!i) \wedge \text{' } X!i = \text{' } X!i \wedge$
 $\text{' } Y!i = \text{' } Y!i \wedge \text{length } \text{' } X = \text{length } \text{' } X \wedge \text{length } \text{' } Y = \text{length } \text{' } Y \rrbracket,$
 $\llbracket (\forall j < n. i \neq j \longrightarrow \text{' } X!j = \text{' } X!i \wedge \text{' } Y!j = \text{' } Y!i) \wedge$
 $\text{' } Y!i \leq \text{' } Y!i \wedge \text{length } \text{' } X = \text{length } \text{' } X \wedge \text{length } \text{' } Y = \text{length } \text{' } Y \rrbracket,$
 $\llbracket (\text{' } X!i) \text{ } \text{mod } n = i \wedge (\forall j < \text{' } X!i. j \text{ } \text{mod } n = i \longrightarrow \neg P(B!j)) \wedge (\text{' } Y!i < m \longrightarrow P(B!(\text{' } Y!i))$
 $\wedge \text{' } Y!i \leq m + i) \wedge (\exists j < n. \text{' } Y!j \leq \text{' } X!i) \rrbracket \text{ COEND}$
 $\text{SAT } [\llbracket n < \text{length } \text{' } X \wedge n < \text{length } \text{' } Y \wedge (\forall i < n. \text{' } X!i = i \wedge \text{' } Y!i = m + i) \rrbracket,$
 $\llbracket \text{' } X = \text{' } X \wedge \text{' } Y = \text{' } Y \rrbracket,$
 $\llbracket \text{True} \rrbracket,$

```

    { $\forall i < n. ('X!i) \text{ mod } n = i \wedge (\forall j < 'X!i. j \text{ mod } n = i \longrightarrow \neg P(B!j)) \wedge$ 
       $('Y!i < m \longrightarrow P(B!('Y!i)) \wedge 'Y!i \leq m+i) \wedge (\exists j < n. 'Y!j \leq 'X!i)}$ }
  apply (rule Parallel)
apply (auto cong del: image-cong-simp)
apply force
apply (rule While)
  apply force
  apply force
  apply force
  apply (erule dvdE)
  apply (rule-tac pre'= $\{n < \text{length } 'X \wedge n < \text{length } 'Y \wedge 'X!i \text{ mod } n = i \wedge (\forall j. j$ 
     $< 'X!i \longrightarrow j \text{ mod } n = i \longrightarrow \neg P(B!j)) \wedge ('Y!i < n * k \longrightarrow P(B!('Y!$ 
     $i))) \wedge 'X!i < 'Y!i\}$  in Conseq)
    apply force
    apply (rule subset-refl)+
  apply (rule Cond)
    apply force
    apply (rule Basic)
      apply force
      apply force
      apply force
      apply force
    apply (rule Basic)
      apply simp
      apply clarify
      apply simp
      apply (rule allI)
      apply (rule impI)+
      apply (case-tac  $X\ x!\ i \leq j$ )
      apply (drule le-imp-less-or-eq)
      apply (erule disjE)
      apply (drule-tac  $j=j$  and  $n=n$  and  $i=i$  and  $a=X\ x!\ i$  in mod-aux)
      apply auto
done

end
theory Hoare-Parallel
imports OG-Examples Gar-Coll Mul-Gar-Coll RG-Examples
begin

end

```

Bibliography

- [1] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.
- [2] Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999.
- [3] Leonor Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618, pages 348–362, 2003.
- [4] Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS 2000)*, volume 1893 of *LNCS*, pages 619–628. Springer-Verlag, 2000.