

Hoare Logic for Parallel Programs

Leonor Prensa Nieto

June 21, 2010

Abstract

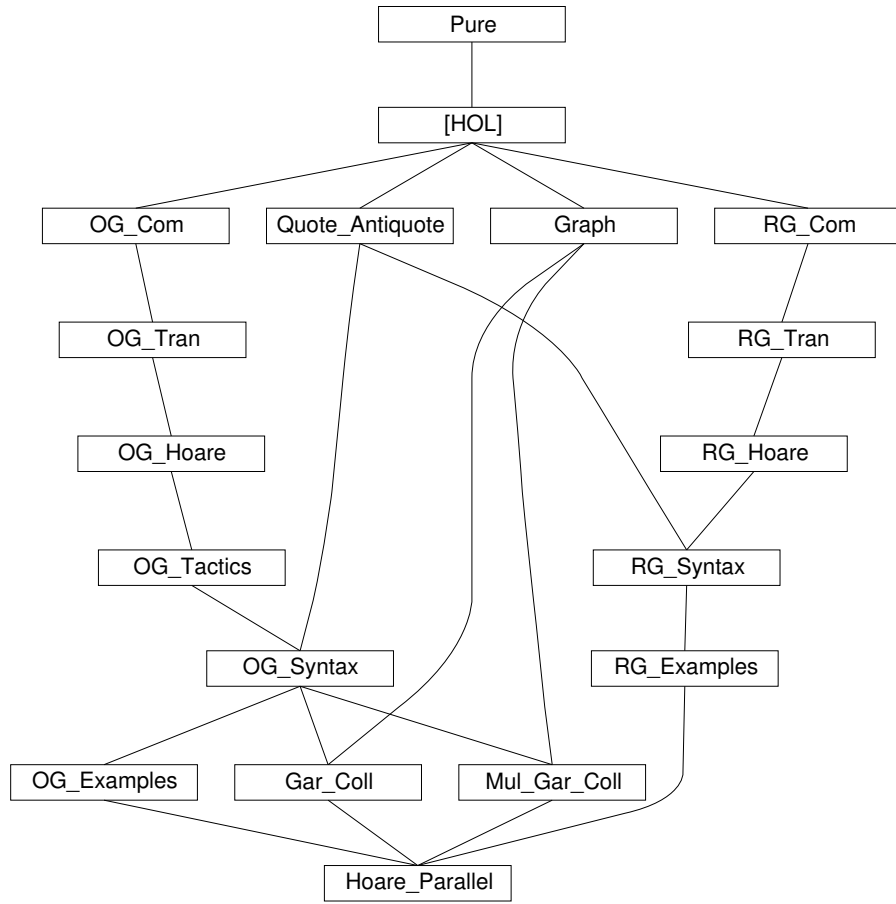
In the following theories a formalization of the Owicki-Gries and the rely-guarantee methods is presented. These methods are widely used for correctness proofs of parallel imperative programs with shared variables. We define syntax, semantics and proof rules in Isabelle/HOL. The proof rules also provide 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	7
1.3	Validity of Correctness Formulas	11
1.4	The Proof System	11
1.5	Soundness	13
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	110
3.4	The Proof System	122
3.4.1	Proof System for Component Programs	122
3.4.2	Proof System for Parallel Programs	123
3.5	Soundness	123
3.5.1	Soundness of the System for Component Programs	128
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	153
3.7.2	Increment a Variable in Parallel	154
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:

types

'a bexp = *'a set*
'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:

consts

pre :: *'a ann-com* \Rightarrow *'a assn*

primrec

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:

```

consts atom-com :: 'a com  $\Rightarrow$  bool
primrec
  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**

```

types
  'a ann-com-op = ('a ann-com) option
  '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
    (-  $-1 \rightarrow$  -[81,81] 100)
  and transition' :: ('a com  $\times$  'a)  $\Rightarrow$  ('a com  $\times$  'a)  $\Rightarrow$  bool
    (-  $-P1 \rightarrow$  -[81,81] 100)
  and transitions :: ('a com  $\times$  'a)  $\Rightarrow$  ('a com  $\times$  'a)  $\Rightarrow$  bool
    (-  $-P* \rightarrow$  -[81,81] 100)
where
  con-0  $-1 \rightarrow$  con-1  $\equiv (con-0, con-1) \in \text{ann-transition}$ 
  | con-0  $-P1 \rightarrow$  con-1  $\equiv (con-0, con-1) \in \text{transition}$ 
  | con-0  $-P* \rightarrow$  con-1  $\equiv (con-0, con-1) \in \text{transition}^*$ 

  | AnnBasic: (Some (AnnBasic r f), s)  $-1 \rightarrow$  (None, f s)

  | AnnSeq1: (Some c0, s)  $-1 \rightarrow$  (None, t)  $\Longrightarrow$ 

```

$(\text{Some } (\text{AnnSeq } c0 \ c1), s) -1 \rightarrow (\text{Some } c1, t)$
 $| \text{AnnSeq2}: (\text{Some } c0, s) -1 \rightarrow (\text{Some } c2, t) \implies$
 $(\text{Some } (\text{AnnSeq } c0 \ c1), s) -1 \rightarrow (\text{Some } (\text{AnnSeq } c2 \ c1), t)$

$| \text{AnnCond1T}: s \in b \implies (\text{Some } (\text{AnnCond1 } r \ b \ c1 \ c2), s) -1 \rightarrow (\text{Some } c1, s)$
 $| \text{AnnCond1F}: s \notin b \implies (\text{Some } (\text{AnnCond1 } r \ b \ c1 \ c2), s) -1 \rightarrow (\text{Some } c2, s)$

$| \text{AnnCond2T}: s \in b \implies (\text{Some } (\text{AnnCond2 } r \ b \ c), s) -1 \rightarrow (\text{Some } c, s)$
 $| \text{AnnCond2F}: s \notin b \implies (\text{Some } (\text{AnnCond2 } r \ b \ c), s) -1 \rightarrow (\text{None}, s)$

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

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

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

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

$| \text{Seq1}: \text{All-None } Ts \implies (\text{Seq } (\text{Parallel } Ts) \ c, s) -P1 \rightarrow (c, s)$
 $| \text{Seq2}: (c0, s) -P1 \rightarrow (c2, t) \implies (\text{Seq } c0 \ c1, s) -P1 \rightarrow (\text{Seq } c2 \ c1, t)$

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

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

monos *rtrancl-mono*

The corresponding abbreviations are:

abbreviation

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

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

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

1.2.2 Definition of Semantics

definition $ann\text{-}sem :: 'a\ ann\text{-}com \Rightarrow 'a \Rightarrow 'a\ set$ **where**
 $ann\text{-}sem\ c \equiv \lambda s. \{t. (Some\ c, s) \dashv\!\!\rightarrow (None, t)\}$

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

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

definition $SEM :: 'a\ com \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
 $SEM\ c\ S \equiv \bigcup sem\ c\ `S$

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

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

Proofs

declare $ann\text{-}transition\text{-}transition.intros$ [intro]

inductive-cases $transition\text{-}cases$:

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

lemma $Parallel\text{-}empty\text{-}lemma$ [rule-format (no-asm)]:
 $(Parallel\ [], s) \dashv\!\!\rightarrow (Parallel\ Ts, t) \longrightarrow Ts = [] \wedge n = 0 \wedge s = t$
apply(induct n)
apply(simp (no-asm))
apply clarify
apply(drule rel-pow-Suc-D2)
apply(force elim:transition-cases)
done

lemma $Parallel\text{-}AllNone\text{-}lemma$ [rule-format (no-asm)]:
 $All\text{-}None\ Ss \longrightarrow (Parallel\ Ss, s) \dashv\!\!\rightarrow (Parallel\ Ts, t) \longrightarrow Ts = Ss \wedge n = 0 \wedge s = t$
apply(induct n)
apply(simp (no-asm))
apply clarify
apply(drule rel-pow-Suc-D2)
apply clarify
apply(erule transition-cases, simp-all)
apply(force dest:nth-mem simp add:All-None-def)
done

```

lemma Parallel-AllNone:  $All\text{-}None\ Ts \implies (SEM\ (Parallel\ Ts)\ X) = X$ 
apply (unfold SEM-def sem-def)
apply auto
apply (drule rtrancl-imp-UN-rel-pow)
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:
  
$$\begin{aligned} & \llbracket (c1, s1) -P* \rightarrow (Parallel\ Ts, s2); All\text{-}None\ Ts; \\ & \quad (c2, s2) -P* \rightarrow (Parallel\ Ss, s3); All\text{-}None\ Ss \rrbracket \\ & \implies (Seq\ c1\ c2, s1) -P* \rightarrow (Parallel\ Ss, s3) \end{aligned}$$

apply (erule converse-rtrancl-induct2)
apply (force intro: converse-rtrancl-into-rtrancl) +
done

```

```

lemma L3-5ii-lemma2 [rule-format (no-asm)]:
  
$$\begin{aligned} & \forall c1\ c2\ s\ t. (Seq\ c1\ c2, s) -Pn \rightarrow (Parallel\ Ts, t) \longrightarrow \\ & \quad (All\text{-}None\ Ts) \longrightarrow (\exists y\ m\ Rs. (c1, s) -P* \rightarrow (Parallel\ Rs, y) \wedge \\ & \quad (All\text{-}None\ Rs) \wedge (c2, y) -Pm \rightarrow (Parallel\ Ts, t) \wedge m \leq n) \end{aligned}$$

apply (induct n)
apply (force)
apply (safe dest!: rel-pow-Suc-D2)
apply (erule transition-cases, simp-all)
apply (fast intro!: le-SucI)
apply (fast intro!: le-SucI elim!: rel-pow-imp-rtrancl converse-rtrancl-into-rtrancl)
done

```

```

lemma L3-5ii-lemma3:
  
$$\begin{aligned} & \llbracket (Seq\ c1\ c2, s) -P* \rightarrow (Parallel\ Ts, t); All\text{-}None\ Ts \rrbracket \implies \\ & \quad (\exists y\ Rs. (c1, s) -P* \rightarrow (Parallel\ Rs, y) \wedge All\text{-}None\ Rs \\ & \quad \wedge (c2, y) -P* \rightarrow (Parallel\ Ts, t)) \end{aligned}$$

apply (drule rtrancl-imp-UN-rel-pow)
apply (fast dest: L3-5ii-lemma2 rel-pow-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 rel-pow-E2)
  apply simp-all
  apply(erule transition-cases)
  apply simp-all
  apply(erule rel-pow-E2)
  apply(simp add: Id-def)
  apply(erule transition-cases,simp-all)
  apply clarify
  apply(erule transition-cases,simp-all)
  apply(erule rel-pow-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-rel-pow L3-5v-lemma1)
done

```

```

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

lemma L3-5v-lemma4 [rule-format]:
 $\forall s. (\text{While } b \text{ i } c, s) - Pn \rightarrow (\text{Parallel } Ts, t) \longrightarrow \text{All-None } Ts \longrightarrow$ 
 $(\exists k. (\text{fwhile } b \text{ c } k, s) - P* \rightarrow (\text{Parallel } Ts, t))$ 
apply(rule nat-less-induct)
apply safe
apply(erule rel-pow-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-rel-pow)
apply clarify
apply(erule rel-pow-E2)
apply simp-all
apply(erule transition-cases,simp-all)
apply(fast dest: Parallel-empty-lemma)

```

done

```

lemma L3-5v: SEM (While b i c) = (λx. (∪ k. SEM (fwhile b c k) x))
apply (rule ext)
apply (simp add: SEM-def sem-def)
apply safe
apply (drule rtrancl-imp-UN-rel-pow, 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 ((\exists) \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**

```

consts assertions :: 'a ann-com  $\Rightarrow$  ('a assn) set
primrec
  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}

```

```

consts atomics :: 'a ann-com  $\Rightarrow$  ('a assn  $\times$  'a com) set
primrec

```

```

  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)}

```

```

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

```

consts $post :: 'a \text{ ann-triple-op} \Rightarrow 'a \text{ assn}$

primrec $post \ (c, q) = q$

definition $interfree\text{-}aux :: ('a \text{ ann-com-op} \times 'a \text{ assn} \times 'a \text{ ann-com-op}) \Rightarrow bool$

where

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

definition $interfree :: (('a \text{ ann-triple-op}) \text{ list}) \Rightarrow bool$ **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

$oghoare :: 'a \text{ assn} \Rightarrow 'a \text{ com} \Rightarrow 'a \text{ assn} \Rightarrow bool \ ((3 \parallel - \ -// -// -) \ [90, 55, 90] \ 50)$

and $ann\text{-}hoare :: 'a \text{ ann-com} \Rightarrow 'a \text{ assn} \Rightarrow bool \ ((2 \vdash -// -) \ [60, 90] \ 45)$

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $AnnBasic = oghoare\text{-}ann\text{-}hoare.AnnBasic$

lemmas $AnnSeq = oghoare\text{-}ann\text{-}hoare.AnnSeq$

lemmas $AnnCond1 = oghoare\text{-}ann\text{-}hoare.AnnCond1$

lemmas $AnnCond2 = oghoare\text{-}ann\text{-}hoare.AnnCond2$

lemmas $AnnWhile = oghoare\text{-}ann\text{-}hoare.AnnWhile$

lemmas $AnnAwait = oghoare\text{-}ann\text{-}hoare.AnnAwait$

lemmas $AnnConseq = oghoare\text{-}ann\text{-}hoare.AnnConseq$

lemmas $Parallel = oghoare\text{-}ann\text{-}hoare.Parallel$

lemmas $Basic = oghoare\text{-}ann\text{-}hoare.Basic$

lemmas $Seq = oghoare\text{-}ann\text{-}hoare.Seq$

lemmas $Cond = oghoare\text{-}ann\text{-}hoare.Cond$

lemmas $While = oghoare\text{-}ann\text{-}hoare.While$

lemmas $Conseq = oghoare\text{-}ann\text{-}hoare.Conseq$

1.5.1 Soundness of the System for Atomic Programs

lemma $Basic\text{-}ntran\ [rule\text{-}format]:$

$(Basic\ f,\ s) \text{--} Pn \rightarrow (Parallel\ Ts,\ t) \rightarrow All\text{--}None\ Ts \rightarrow t = f\ s$

apply $(induct\ n)$

apply $(simp\ (no\text{-}asm))$

apply $(fast\ dest: rel\text{-}pow\text{-}Suc\text{-}D2\ Parallel\text{-}empty\text{-}lemma\ elim: transition\text{-}cases)$

done

lemma $SEM\text{-}fwhile: SEM\ S\ (p \cap b) \subseteq p \implies SEM\ (fwhile\ b\ S\ k)\ p \subseteq (p \cap \neg b)$

apply $(induct\ k)$

apply $(simp\ (no\text{-}asm)\ add: L3\text{-}5v\text{-}lemma3)$

apply $(simp\ (no\text{-}asm)\ add: L3\text{-}5iv\ L3\text{-}5ii\ Parallel\text{-}empty)$

apply $(rule\ conjI)$

apply $(blast\ dest: L3\text{-}5i)$

apply $(simp\ add: SEM\text{-}def\ sem\text{-}def\ id\text{-}def)$

apply $(blast\ dest: Basic\text{-}ntran\ rtrancl\text{-}imp\text{-}UN\text{-}rel\text{-}pow)$

done

lemma $atom\text{-}hoare\text{-}sound\ [rule\text{-}format]:$

$\| \neg\ p\ c\ q \rightarrow atom\text{-}com(c) \rightarrow \| =\ p\ c\ q$

apply $(unfold\ com\text{-}validity\text{-}def)$

apply $(rule\ oghoare\text{-}induct)$

apply $simp\text{-}all$

— $Basic$

apply $(simp\ add: SEM\text{-}def\ sem\text{-}def)$

apply $(fast\ dest: rtrancl\text{-}imp\text{-}UN\text{-}rel\text{-}pow\ Basic\text{-}ntran)$

— 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) -1→ (c', s')
  (Some (AnnBasic r f),s) -1→ (c', s')
  (Some (AnnSeq c1 c2), s) -1→ (c', s')
  (Some (AnnCond1 r b c1 c2), s) -1→ (c', s')
  (Some (AnnCond2 r b c), s) -1→ (c', s')
  (Some (AnnWhile r b I c), s) -1→ (c', s')
  (Some (AnnAwait r b c),s) -1→ (c', s')

```

Strong Soundness for Component Programs:

lemma *ann-hoare-case-analysis* [rule-format]:

```

  defines I: I ≡ λC q'.
    ((∀ r f. C = AnnBasic r f → (∃ q. r ⊆ {s. f s ∈ q} ∧ q ⊆ q')) ∧
    (∀ c0 c1. C = AnnSeq c0 c1 → (∃ q. q ⊆ q' ∧ ⊢ c0 pre c1 ∧ ⊢ c1 q)) ∧
    (∀ r b c1 c2. C = AnnCond1 r b c1 c2 → (∃ q. q ⊆ q' ∧
    r ∩ b ⊆ pre c1 ∧ ⊢ c1 q ∧ r ∩ ¬b ⊆ pre c2 ∧ ⊢ c2 q)) ∧
    (∀ r b c. C = AnnCond2 r b c →
    (∃ q. q ⊆ q' ∧ r ∩ b ⊆ pre c ∧ ⊢ c q ∧ r ∩ ¬b ⊆ q)) ∧
    (∀ r i b c. C = AnnWhile r b i c →
    (∃ q. q ⊆ q' ∧ r ⊆ i ∧ i ∩ b ⊆ pre c ∧ ⊢ c i ∧ i ∩ ¬b ⊆ q)) ∧
    (∀ r b c. C = AnnAwait r b c → (∃ q. q ⊆ q' ∧ ⊢ (r ∩ b) c q)))
  shows ⊢ C q' → I C q'
  apply(rule ann-hoare-induct)
  apply (simp-all add: I)
  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 ∩ {(x,y). True}) = (transition)

```

  apply force
done

```

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

```

  (co, s) -1→ (co', t) → (∀ c. co = Some c → s ∈ pre c →

```


$(\forall q. \vdash c \ q \longrightarrow (\text{if } co' = \text{None} \text{ then } t \in q \text{ else } t \in \text{pre}(\text{the } co') \wedge \vdash (\text{the } co') \ q)))$
apply(rule ann-transition-transition.induct [THEN conjunct1])
apply simp-all
 — Basic
 apply clarify
 apply(frule 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: 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)]:
   $\llbracket \forall i < \text{length } Ts. (\text{if } \text{com}(Ts!i) = \text{None} \text{ then } b \in \text{post}(Ts!i)) \rrbracket$ 

```

$else\ b \in pre(the(com(Ts!i))) \wedge \vdash the(com(Ts!i))\ post(Ts!i);$
 $com(Ts\ !\ i) = Some(AnnSeq\ c0\ c1); i < length\ Ts; interfree\ Ts\] \implies$
 $(\forall\ ia < length\ Ts. (if\ com(Ts[i:=(Some\ c0,\ pre\ c1)]!\ ia) = None$
 $then\ b \in post(Ts[i:=(Some\ c0,\ pre\ c1)]!\ ia)$
 $else\ b \in pre(the(com(Ts[i:=(Some\ c0,\ pre\ c1)]!\ ia))) \wedge$
 $\vdash the(com(Ts[i:=(Some\ c0,\ pre\ c1)]!\ ia))\ post(Ts[i:=(Some\ c0,\ pre\ c1)]!\ ia)))$
 $\wedge interfree\ (Ts[i:= (Some\ c0,\ pre\ c1)])$
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]:

$(Some\ c,\ b) - 1 \rightarrow (co,\ t) \longrightarrow$
 $(\forall\ Ts. i < length\ Ts \longrightarrow com(Ts\ !\ i) = Some\ c \longrightarrow$
 $(\forall\ i < length\ Ts. (if\ com(Ts\ !\ i) = None\ then\ b \in post(Ts!i)$
 $else\ b \in pre(the(com(Ts!i))) \wedge \vdash the(com(Ts!i))\ post(Ts!i))) \longrightarrow$
 $interfree\ Ts \longrightarrow$
 $(\forall\ j. j < length\ Ts \wedge i \neq j \longrightarrow (if\ com(Ts!j) = None\ then\ t \in post(Ts!j)$
 $else\ t \in pre(the(com(Ts!j))) \wedge \vdash the(com(Ts!j))\ post(Ts!j))))$
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]:

$$\begin{aligned}
& \llbracket (Ts',s) -P* \rightarrow (Rs',t); Ts' = (Parallel\ Ts); interfree\ Ts; \\
& \forall i. i < length\ Ts \longrightarrow (\exists c\ q. (Ts\ !\ i) = (Some\ c, q) \wedge s \in (pre\ c) \wedge \vdash c\ q) \rrbracket \implies \\
& \forall Rs. Rs' = (Parallel\ Rs) \longrightarrow (\forall j. j < length\ Rs \longrightarrow \\
& \quad (if\ com(Rs\ !\ j) = None\ then\ t \in post(Ts\ !\ j) \\
& \quad \quad else\ t \in pre(the(com(Rs\ !\ j))) \wedge \vdash the(com(Rs\ !\ j))\ post(Ts\ !\ j))) \wedge interfree\ Rs \\
& \llbracket erule\ rtrancl-induct2 \rrbracket \\
& \text{apply } clarify \\
& \text{— Base} \\
& \text{apply } force \\
& \text{— Induction step} \\
& \text{apply } clarify \\
& \text{apply } (drule\ Parallel-length-post-PStar) \\
& \text{apply } clarify \\
& \text{apply } (ind-cases\ (Parallel\ Ts, s) -P1 \rightarrow (Parallel\ Rs, t)\ \text{for } Ts\ s\ Rs\ t) \\
& \text{apply } (rule\ conjI) \\
& \text{apply } clarify \\
& \text{apply } (case-tac\ i=j) \\
& \text{apply } (simp\ split\ del: split-if) \\
& \text{apply } (erule\ Strong-Soundness-aux-aux,simp+) \\
& \text{apply } force \\
& \text{apply } force \\
& \text{apply } (simp\ split\ del: split-if) \\
& \text{apply } (erule\ Parallel-Strong-Soundness-aux-aux) \\
& \text{apply } (simp-all\ add: split\ del: split-if) \\
& \text{apply } force \\
& \text{apply } (rule\ interfree-lemma) \\
& \text{apply } simp-all
\end{aligned}$$


```

done

lemma *Parallel-Strong-Soundness*:

$\llbracket (Parallel\ Ts, s) - P^* \rightarrow (Parallel\ Rs, t); interfree\ Ts; j < length\ Rs; \\ \forall i. i < length\ Ts \longrightarrow (\exists c\ q. Ts\ !\ i = (Some\ c, q) \wedge s \in pre\ c \wedge \vdash c\ q) \rrbracket \Longrightarrow \\ if\ com(Rs\ !\ j) = None\ then\ t \in post(Ts\ !\ j)\ else\ t \in pre\ (the(com(Rs\ !\ j)))$
apply (*drule Parallel-Strong-Soundness-aux*)
apply *simp+*
done

lemma *oghoare-sound* [*rule-format*]: $\llbracket -\ p\ c\ q \longrightarrow \rrbracket = p\ c\ q$

apply (*unfold com-validity-def*)
apply (*rule oghoare-induct*)
apply (*rule TrueI*) +
— Parallel
apply (*simp add: SEM-def sem-def*)
apply *clarify*
apply (*frule Parallel-length-post-PStar*)
apply *clarify*
apply (*drule-tac j=xb in Parallel-Strong-Soundness*)
apply *clarify*
apply *simp*
apply *force*
apply *simp*
apply (*erule-tac V = $\forall i. ?P\ i$ 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-rel-pow 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 An-
nAwait AnnConseq

```

```

lemmas oghoare-intros=Parallel Basic Seq Cond While Conseq

```

```

lemma ParallelConseqRule:

```

```

   $\llbracket p \subseteq (\bigcap_{i \in \{i. i < \text{length } Ts\}}. \text{pre}(\text{the}(\text{com}(Ts ! i))));$ 
   $\llbracket - (\bigcap_{i \in \{i. i < \text{length } Ts\}}. \text{pre}(\text{the}(\text{com}(Ts ! i))))$ 
   $(\text{Parallel } Ts)$ 
   $(\bigcap_{i \in \{i. i < \text{length } Ts\}}. \text{post}(Ts ! i));$ 
   $(\bigcap_{i \in \{i. i < \text{length } Ts\}}. \text{post}(Ts ! i)) \subseteq q \rrbracket$ 
   $\implies \llbracket - p (\text{Parallel } Ts) q$ 
apply (rule Conseq)
prefer 2
apply fast
apply assumption+
done

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

apply(rule Seq)
apply fast+
done

```

```

lemma CondRule:

```

```

   $\llbracket p \subseteq \{s. (s \in b \implies s \in w) \wedge (s \notin b \implies s \in w')\}; \llbracket - w \ c1 \ q; \llbracket - w' \ c2 \ q \rrbracket$ 
   $\implies \llbracket - p (\text{Cond } b \ c1 \ c2) q$ 
apply(rule Cond)
apply(rule Conseq)
prefer 4 apply(rule Conseq)
apply simp-all
apply force+
done

```

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

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

lemma *AnnskipRule*:
 $r \subseteq q \implies \vdash (AnnBasic \ r \ id) \ q$
apply(rule *AnnBasic*)
apply *simp*
done

lemma *AnnwaitRule*:
 $\llbracket (r \cap b) \subseteq q \rrbracket \implies \vdash (AnnAwait \ r \ b \ (Basic \ id)) \ q$
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*:
 $\forall (R, r) \in (atoms \ a). \parallel - (q \cap R) \ r \ q \implies interfree-aux(None, \ q, \ Some \ a)$
apply(*simp add:interfree-aux-def*)
apply(*force elim:oghoare-sound*)
done

lemma *interfree-aux-rule3*:
 $(\forall (R, r) \in (atoms \ a). \parallel - (q \cap R) \ r \ q \wedge (\forall p \in (assertions \ c). \parallel - (p \cap R) \ r \ p))$
 $\implies interfree-aux(Some \ c, \ q, \ Some \ a)$
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:*

$\llbracket - \ (q \cap r) \ (\text{Basic } f) \ q \rrbracket \implies \text{interfree-aux}(\text{None}, q, \text{Some } (\text{AnnBasic } r \ f))$

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:*

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

lemma AnnCond2-atomics:
 $\text{interfree-aux } (\text{Any}, q, \text{Some } a) \Longrightarrow \text{interfree-aux}(\text{Any}, q, \text{Some } (\text{AnnCond2 } r \ b \ a))$
by(simp add: interfree-aux-def)

lemma AnnWhile-atomics: $\text{interfree-aux } (\text{Any}, q, \text{Some } a) \Longrightarrow \text{interfree-aux}(\text{Any}, q, \text{Some } (\text{AnnWhile } r \ b \ i \ a))$
by(simp add: interfree-aux-def)

lemma Annatom-atomics:
 $\llbracket - \ (q \cap r) \ a \ q \Longrightarrow \text{interfree-aux } (\text{None}, q, \text{Some } (\text{AnnAwait } r \ \{x. \text{True}\} \ a))$
by(simp add: interfree-aux-def oghoare-sound)

lemma AnnAwait-atomics:
 $\llbracket - \ (q \cap (r \cap b)) \ a \ q \Longrightarrow \text{interfree-aux } (\text{None}, q, \text{Some } (\text{AnnAwait } r \ b \ a))$
by(simp add: interfree-aux-def oghoare-sound)

definition interfree-swap :: ('a ann-triple-op * ('a ann-triple-op) list) \Rightarrow bool
where
 $\text{interfree-swap} == \lambda(x, xs). \forall y \in \text{set } xs. \text{interfree-aux } (\text{com } x, \text{post } x, \text{com } y) \wedge \text{interfree-aux}(\text{com } y, \text{post } y, \text{com } x)$

lemma interfree-swap-Empty: $\text{interfree-swap } (x, [])$
by(simp add: interfree-swap-def)

lemma interfree-swap-List:
 $\llbracket \text{interfree-aux } (\text{com } x, \text{post } x, \text{com } y); \text{interfree-aux } (\text{com } y, \text{post } y, \text{com } x); \text{interfree-swap } (x, xs) \rrbracket \Longrightarrow \text{interfree-swap } (x, y \# xs)$
by(simp add: interfree-swap-def)

lemma interfree-swap-Map: $\forall k. i \leq k \wedge k < j \longrightarrow \text{interfree-aux } (\text{com } x, \text{post } x, c \ k) \wedge \text{interfree-aux } (c \ k, Q \ k, \text{com } x) \Longrightarrow \text{interfree-swap } (x, \text{map } (\lambda k. (c \ k, Q \ k)) [i..<j])$
by(force simp add: interfree-swap-def less-diff-conv)

lemma interfree-Empty: $\text{interfree } []$
by(simp add: interfree-def)

lemma interfree-List:
 $\llbracket \text{interfree-swap}(x, xs); \text{interfree } xs \rrbracket \Longrightarrow \text{interfree } (x \# xs)$
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:
  ( $\forall i j. a \leq i \wedge i < b \wedge a \leq j \wedge j < b \wedge i \neq j \longrightarrow \text{interfree-aux } (c \ i, Q \ i, c \ j)$ )
 $\implies \text{interfree } (\text{map } (\lambda k. (c \ k, Q \ k)) [a..<b])$ 
by(force simp add: interfree-def less-diff-conv)

```

```

definition map-ann-hoare :: ( $'a \text{ ann-com-op} * 'a \text{ assn}$ ) list  $\Rightarrow$  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:  $[\vdash c \ q ; [\vdash] \ xs] \implies [\vdash] (\text{Some } c, q) \# xs$ 
apply(simp add: map-ann-hoare-def)
apply clarify
apply(case-tac i, simp+)
done

```

```

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

```

```

lemma ParallelRule:  $[\vdash] \ Ts ; \text{interfree } Ts \implies \parallel - (\bigcap i \in \{i. i < \text{length } Ts\}. \text{pre}(\text{the}(\text{com}(Ts!i))))$ 
 $\text{Parallel } Ts$ 
 $(\bigcap i \in \{i. i < \text{length } Ts\}. \text{post}(Ts!i))$ 
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:  $\text{length } [] = 0 \wedge \text{length } (x \# xs) = \text{Suc}(\text{length } xs)$ 
by simp
lemma list-lemmas:  $\text{length } [] = 0 \wedge \text{length } (x \# xs) = \text{Suc}(\text{length } xs)$ 
 $\wedge (x \# xs) ! 0 = x \wedge (x \# xs) ! \text{Suc } n = xs ! n$ 
by simp
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-def Collect-conj-eq length-map length-upt
length-append list-length

ML <<
val before-interfree-simp-tac = (simp-tac (HOL-basic-ss addsimps [thm com.simps,
thm post.simps]))

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

val ParallelConseq = (simp-tac (HOL-basic-ss addsimps (thms ParallelConseq-list)@(thms
my-simp-list)))
>>

```

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

```

ML <<
fun conjI-Tac tac i st = st |>
  ( (EVERY [rtac conjI i,
    conjI-Tac 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 \ll

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

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

and ParallelTac i = EVERY[rtac (@{thm ParallelRule}) i,
interfree-Tac (i+1),
MapAnn-Tac i]

and MapAnn-Tac i st = st |>
(FIRST[rtac (@{thm MapAnnEmpty}) i,
EVERY[rtac (@{thm MapAnnList}) i,
MapAnn-Tac (i+1),
AnnHoareRuleTac i],

$EVERY[rtac (@\{thm MapAnnMap\}) i,$
 $rtac (@\{thm allI\}) i, rtac (@\{thm impI\}) i,$
 $AnnHoareRuleTac i]]$

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

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

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

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

```

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

```

The final tactic is given the name *oghoare*:

```

ML <<
  val oghoare-tac = SUBGOAL (fn (-, i) =>
    (HoareRuleTac 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 << val annhoare-tac = SUBGOAL (fn (-, i) =>
  (AnnHoareRuleTac i))

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

method-setup disjE-tac = <<
  Scan.succeed (K (SIMPLE-METHOD' (disjE-Tac (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)

```

```

syntax (xsymbols)
  -Assert     :: 'a ⇒ 'a set                     (({!-}) [0] 1000)

```

```

translations
  .{b}. ↦ CONST Collect <<b>>

```

```

parse-translation <<
  let
    fun quote-tr [t] = Syntax.quote-tr @{syntax-const -antiquote} t
    | quote-tr ts = raise TERM (quote-tr, ts);
  in [(@{syntax-const -quote}, 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* \ll '(-update-name *x* (λ -. *a*)) \gg
r '*x* := *a* \rightarrow *CONST AnnBasic r* \ll '(-update-name *x* (λ -. *a*)) \gg

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 (-//<-> [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
((0*IF* - / *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
((0*WHILE* - / *INV* - // *DO* - / *OD*) [0, 0, 0] 61)
-*While* :: 'a bexp \Rightarrow 'a com \Rightarrow 'a com
((0*WHILE* - // *DO* - / *OD*) [0, 0] 61)

translations

IF *b THEN* *c1 ELSE* *c2 FI* \rightarrow *CONST Cond* .{*b*}. *c1 c2*
IF *b THEN* *c FI* \Rightarrow *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* \Rightarrow *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*

$r \text{ AWAIT } b \text{ THEN } c \text{ END} \rightarrow \text{CONST AnnAwait } r .\{b\} . c$
 $r \langle c \rangle \Rightarrow r \text{ AWAIT } \text{CONST True} \text{ THEN } c \text{ END}$
 $r \text{ WAIT } b \text{ END} \Rightarrow r \text{ AWAIT } b \text{ THEN SKIP END}$

nonterminals

prgs

syntax

$\text{-PAR} :: \text{prgs} \Rightarrow 'a \quad (\text{COBEGIN} // - // \text{COEND} [57] \ 56)$
 $\text{-prg} :: ['a, 'a] \Rightarrow \text{prgs} \quad (-// - [60, 90] \ 57)$
 $\text{-prgs} :: ['a, 'a, \text{prgs}] \Rightarrow \text{prgs} \quad (-// - // - [60, 90, 57] \ 57)$

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

translations

$\text{-prg } c \ q \Rightarrow [(\text{CONST Some } c, \ q)]$
 $\text{-prgs } c \ q \ ps \Rightarrow (\text{CONST Some } c, \ q) \# \ ps$
 $\text{-PAR } ps \Rightarrow \text{CONST Parallel } ps$

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

print-translation \ll

let
 $\text{fun quote-tr}' f (t :: ts) =$
 $\quad \text{Term.list-comb } (f \ \$ \ \text{Syntax.quote-tr}' @\{\text{syntax-const -antiquote}\} \ t, \ ts)$
 $\quad | \text{quote-tr}' - - = \text{raise Match};$

 $\text{fun annquote-tr}' f (r :: t :: ts) =$
 $\quad \text{Term.list-comb } (f \ \$ \ r \ \$ \ \text{Syntax.quote-tr}' @\{\text{syntax-const -antiquote}\} \ t, \ ts)$
 $\quad | \text{annquote-tr}' - - = \text{raise Match};$

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

 $\text{fun bexp-tr}' \text{name} ((\text{Const } (@\{\text{const-syntax Collect}\}, -) \ \$ \ t) :: ts) =$
 $\quad \text{quote-tr}' (\text{Syntax.const name}) (t :: ts)$
 $\quad | \text{bexp-tr}' - - = \text{raise Match};$

 $\text{fun annbexp-tr}' \text{name} (r :: (\text{Const } (@\{\text{const-syntax Collect}\}, -) \ \$ \ t) :: ts) =$
 $\quad \text{annquote-tr}' (\text{Syntax.const name}) (r :: t :: ts)$
 $\quad | \text{annbexp-tr}' - - = \text{raise Match};$

 $\text{fun K-tr}' (\text{Abs } (-, -, t)) =$
 $\quad \text{if null (loose-bnos } t) \text{ then } t \text{ else raise Match}$
 $\quad | \text{K-tr}' (\text{Abs } (-, -, \text{Abs } (-, -, t) \ \$ \ \text{Bound } 0)) =$
 $\quad \text{if null (loose-bnos } t) \text{ then } t \text{ else raise Match}$
 $\quad | \text{K-tr}' - - = \text{raise Match};$

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

```

f)      quote-tr' (Syntax.const @{syntax-const -Assign} $ Syntax.update-name-tr'
      (Abs (x, dummyT, K-tr' k) :: ts)
      | assign-tr' - = raise Match;

fun annassign-tr' (r :: Abs (x, -, f $ k $ Bound 0) :: ts) =
  quote-tr' (Syntax.const @{syntax-const -AnnAssign} $ r $ Syntax.update-name-tr'
f)      (Abs (x, dummyT, K-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}, assert-tr'),
   (@{const-syntax Basic}, assign-tr'),
   (@{const-syntax Cond}, bexp-tr' -Cond),
   (@{const-syntax While}, bexp-tr' -While-inv),
   (@{const-syntax AnnBasic}, annassign-tr'),
   (@{const-syntax AnnWhile}, annbexp-tr' -AnnWhile),
   (@{const-syntax AnnAwait}, annbexp-tr' -AnnAwait),
   (@{const-syntax AnnCond1}, annbexp-tr' -AnnCond1),
   (@{const-syntax AnnCond2}, annbexp-tr' -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*:

```

||- .{ 'pr1=0 ∧ ¬ 'in1 ∧ 'pr2=0 ∧ ¬ 'in2 }.
COBEGIN .{ 'pr1=0 ∧ ¬ 'in1 }.
WHILE True INV .{ 'pr1=0 ∧ ¬ 'in1 }.
DO
  .{ 'pr1=0 ∧ ¬ 'in1 }. ⟨ 'in1:=True,, 'pr1:=1 ⟩;;
  .{ 'pr1=1 ∧ 'in1 }. ⟨ 'hold:=1,, 'pr1:=2 ⟩;;
  .{ 'pr1=2 ∧ 'in1 ∧ ('hold=1 ∨ 'hold=2 ∧ 'pr2=2) }.
  AWAIT (¬ 'in2 ∨ ¬ ('hold=1)) THEN 'pr1:=3 END;;
  .{ 'pr1=3 ∧ 'in1 ∧ ('hold=1 ∨ 'hold=2 ∧ 'pr2=2) }.
  ⟨ 'in1:=False,, 'pr1:=0 ⟩
OD .{ 'pr1=0 ∧ ¬ 'in1 }.
||
.{ 'pr2=0 ∧ ¬ 'in2 }.
WHILE True INV .{ 'pr2=0 ∧ ¬ 'in2 }.
DO
  .{ 'pr2=0 ∧ ¬ 'in2 }. ⟨ 'in2:=True,, 'pr2:=1 ⟩;;
  .{ 'pr2=1 ∧ 'in2 }. ⟨ 'hold:=2,, 'pr2:=2 ⟩;;
  .{ 'pr2=2 ∧ 'in2 ∧ ('hold=2 ∨ ('hold=1 ∧ 'pr1=2)) }.
  AWAIT (¬ 'in1 ∨ ¬ ('hold=2)) THEN 'pr2:=3 END;;
  .{ 'pr2=3 ∧ 'in2 ∧ ('hold=2 ∨ ('hold=1 ∧ 'pr1=2)) }.
  ⟨ 'in2:=False,, 'pr2:=0 ⟩
OD .{ 'pr2=0 ∧ ¬ 'in2 }.
COEND
.{ 'pr1=0 ∧ ¬ 'in1 ∧ 'pr2=0 ∧ ¬ '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*:

```

||- .{ True }.
'flag1:=False,, 'flag2:=False,,

```

```

COBEGIN  $\neg 'flag1$ .
  WHILE True
  INV  $\neg 'flag1$ .
  DO  $\neg 'flag1$ .  $\langle 'flag1 := True, 'after1 := False \rangle;;$ 
     $\langle 'flag1 \wedge \neg 'after1 \rangle. \langle 'turn := 1, 'after1 := True \rangle;;$ 
     $\langle 'flag1 \wedge 'after1 \wedge ('turn = 1 \vee 'turn = 2) \rangle.$ 
      WHILE  $\neg('flag2 \longrightarrow 'turn = 2)$ 
      INV  $\langle 'flag1 \wedge 'after1 \wedge ('turn = 1 \vee 'turn = 2) \rangle.$ 
      DO  $\langle 'flag1 \wedge 'after1 \wedge ('turn = 1 \vee 'turn = 2) \rangle. SKIP OD;;$ 
       $\langle 'flag1 \wedge 'after1 \wedge ('flag2 \wedge 'after2 \longrightarrow 'turn = 2) \rangle.$ 
       $'flag1 := False$ 
    OD
   $\langle False \rangle.$ 
||
 $\neg 'flag2$ .
  WHILE True
  INV  $\neg 'flag2$ .
  DO  $\neg 'flag2$ .  $\langle 'flag2 := True, 'after2 := False \rangle;;$ 
     $\langle 'flag2 \wedge \neg 'after2 \rangle. \langle 'turn := 2, 'after2 := True \rangle;;$ 
     $\langle 'flag2 \wedge 'after2 \wedge ('turn = 1 \vee 'turn = 2) \rangle.$ 
      WHILE  $\neg('flag1 \longrightarrow 'turn = 1)$ 
      INV  $\langle 'flag2 \wedge 'after2 \wedge ('turn = 1 \vee 'turn = 2) \rangle.$ 
      DO  $\langle 'flag2 \wedge 'after2 \wedge ('turn = 1 \vee 'turn = 2) \rangle. SKIP OD;;$ 
       $\langle 'flag2 \wedge 'after2 \wedge ('flag1 \wedge 'after1 \longrightarrow 'turn = 1) \rangle.$ 
       $'flag2 := False$ 
    OD
   $\langle False \rangle.$ 
COEND
 $\langle False \rangle.$ 
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*:

$\parallel - \langle i \neq j \rangle.$

$'out := True$,,

COBEGIN $\langle i \neq j \rangle.$

WHILE True INV $\langle i \neq j \rangle.$

DO $\langle i \neq j \rangle. AWAIT 'out THEN 'out := False, 'who := i END;;$

$\langle \neg 'out \wedge 'who = i \wedge i \neq j \rangle. 'out := True OD$

$\langle False \rangle.$

\parallel

```

    .{i≠j}.
    WHILE True INV .{i≠j}.
    DO .{i≠j}. AWAIT 'out THEN 'out:=False,, 'who:=j END;;
    .{¬'out ∧ 'who=j ∧ i≠j}. 'out:=True OD
    .{False}.
  COEND
  .{False}.
apply oghoare
— 38 vc
apply auto
done

```

Peterson's Algorithm III: Parameterized version:

lemma *Semaphores-parameterized-mutex*:

```

0 < n ⇒ || - .{True}.
'out:=True ,,
COBEGIN
SCHEME [0 ≤ i < n]
  .{True}.
  WHILE True INV .{True}.
  DO .{True}. AWAIT 'out THEN 'out:=False,, 'who:=i END;;
  .{¬'out ∧ 'who=i}. 'out:=True OD
  .{False}.
COEND
  .{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*:

```

[[ 0 < n; I = <<n=length 'turn ∧ 0 < 'nextv ∧ (∀ k l. k < n ∧ l < n ∧ k ≠ l
  → 'turn!k < 'num ∧ ('turn!k = 0 ∨ 'turn!k ≠ 'turn!l))>> ]]
⇒ || - .{n=length 'turn}.
'index:=0,,
WHILE 'index < n INV .{n=length 'turn ∧ (∀ i < 'index. 'turn!i=0)}.
DO 'turn:='turn['index:=0],, 'index:='index +1 OD,,
'num:=1 ,, 'nextv:=1 ,,
COBEGIN
SCHEME [0 ≤ i < n]
  .{'I}.

```

```

    WHILE True INV .{ 'I }.
    DO .{ 'I }. { 'turn := 'turn[i := 'num],, 'num := 'num + 1 };;
    .{ 'I }. WAIT 'turn != 'nextv END;;
    .{ 'I ∧ 'turn != 'nextv }. 'nextv := 'nextv + 1
    OD
    .{ False }.
COEND
    .{ False }.
apply oghoare
— 35 vc
apply simp-all
— 21 vc
apply(tactic ⟨ ALLGOALS (clarify-tac @ {claset}) ⟩)
— 11 vc
apply simp-all
apply(tactic ⟨ ALLGOALS (clarify-tac @ {claset}) ⟩)
— 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=i in allE)
apply fastsimp
— 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:

$$\llbracket I1 \rrbracket \leq a \leq 'x \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$$

$\wedge (\neg 'found \wedge a < 'x \longrightarrow f('x) \neq 0) \gg ;$
 $I2 = \ll 'y \leq a+1 \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$
 $\wedge (\neg 'found \wedge 'y \leq a \longrightarrow f('y) \neq 0) \gg \rrbracket \implies$
 $\parallel - .\{\exists u. f(u)=0\}.$
 $'turn:=1,, 'found:= False,,$
 $'x:=a,, 'y:=a+1 ,,$
COBEGIN $.\{'I1\}.$
WHILE $\neg 'found$
INV $.\{'I1\}.$
DO $.\{a \leq 'x \wedge ('found \longrightarrow 'y \leq a \wedge f('y)=0) \wedge (a < 'x \longrightarrow f('x) \neq 0)\}.$
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;;
 $.\{'I1 \wedge 'found\}.$
 $'turn:=2$
 $.\{'I1 \wedge 'found\}.$
 \parallel
 $.\{'I2\}.$
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:*

$\ll I1 = \ll a \leq 'x \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$
 $\wedge (\neg 'found \wedge a < 'x \longrightarrow f('x) \neq 0) \gg ;$
 $I2 = \ll 'y \leq a+1 \wedge ('found \longrightarrow (a < 'x \wedge f('x)=0) \vee ('y \leq a \wedge f('y)=0))$
 $\wedge (\neg 'found \wedge 'y \leq a \longrightarrow f('y) \neq 0) \gg \rrbracket \implies$

```

||- .{ $\exists u. f(u)=0$ }.
'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  $\wedge$  'found}.
||
.{'I2}.
  WHILE  $\neg$ 'found
  INV .{'I2}.
  DO .{'y  $\leq a+1 \wedge ('found \longrightarrow a < 'x \wedge f('x)=0) \wedge ('y \leq a \longrightarrow f('y) \neq 0)$ }.
    < 'y:=('y - 1),, IF  $f('y)=0$  THEN 'found:=True ELSE SKIP FI >
  OD
  .{'I2  $\wedge$  'found}.
COEND
.f('x)=0  $\vee$  f('y)=0}.
apply oghoare
— 20 vc
apply auto
— auto takes aprox. 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 $\dots < n$

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 $b=b \text{ div } n*n + b \bmod n$)

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

apply(subgoal-tac $a=a \text{ div } n*n + a \bmod n$)

prefer 2

apply(simp add: mod-div-equality [symmetric])

apply(subgoal-tac $b - a \leq b - c$)

prefer 2 **apply** arith

apply(drule le-less-trans)

back

```

  apply assumption
  apply(frul2 less-not-refl2)
  apply(drul2 less-imp-le)
  apply (drul2-tac m = a and k = n in div-le-mono)
  apply(safe)
  apply(frul2-tac b = b and a = a and n = n in nat-lemma2, assumption, as-
    sumption)
  apply assumption
  apply(drul2 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 < length a ∧ 0 < length 'buffer ∧ length 'b = length a» ;
  I = «(∀ k < 'ins. 'outs ≤ k → (a ! k) = 'buffer ! (k mod (length 'buffer))) ∧
    'outs ≤ 'ins ∧ 'ins - 'outs ≤ length 'buffer» ;
  I1 = «'I ∧ 'li ≤ length a» ;
  p1 = «'I1 ∧ 'li = 'ins» ;
  I2 = «'I ∧ (∀ k < 'lj. (a ! k) = ('b ! k)) ∧ 'lj ≤ length a» ;
  p2 = «'I2 ∧ 'lj = 'outs» ]] ⇒
||- .{ 'INIT }.
'ins := 0,, 'outs := 0,, 'li := 0,, 'lj := 0,,
COBEGIN .{ 'p1 ∧ 'INIT }.
  WHILE 'li < length a
    INV .{ 'p1 ∧ 'INIT }.
  DO .{ 'p1 ∧ 'INIT ∧ 'li < length a }.
    'vx := (a ! 'li);
    .{ 'p1 ∧ 'INIT ∧ 'li < length a ∧ 'vx = (a ! 'li) }.
    WAIT 'ins - 'outs < length 'buffer END;;
    .{ 'p1 ∧ 'INIT ∧ 'li < length a ∧ 'vx = (a ! 'li)
      ∧ 'ins - 'outs < length 'buffer }.
    'buffer := (list-update 'buffer ('ins mod (length 'buffer)) 'vx);
    .{ 'p1 ∧ 'INIT ∧ 'li < length a
      ∧ (a ! 'li) = ('buffer ! ('ins mod (length 'buffer)))
      ∧ 'ins - 'outs < length 'buffer }.

```

```

      'ins := 'ins + 1;;
      .{ 'I1 ∧ 'INIT ∧ ('li + 1) = 'ins ∧ 'li < length a }.
      'li := 'li + 1
    OD
    .{ 'p1 ∧ 'INIT ∧ 'li = length a }.
  ||
  .{ 'p2 ∧ '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 @ {claset}) >>)
— 112 subgoals left
apply (simp-all (no-asm))
— 97 subgoals left
apply (tactic << ALLGOALS (conjI-Tac (K all-tac)) >>)
— 930 subgoals left
apply (tactic << ALLGOALS (clarify-tac @ {claset}) >>)
— 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 (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-(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=setsum (b(j := (Suc 0))) {0..<n} in ssubst)
apply(frule-tac b=b in Example2-lemma2-aux)
apply(erule-tac t=setsum b {0..<n} in ssubst)

```

```

apply(subgoal-tac Suc (setsum b {0..j} + b j + ( $\sum_{i=0..<n} \text{Suc } j. b (\text{Suc } j + i)$ ))=
(setsum b {0..j} + Suc (b j) + ( $\sum_{i=0..<n} \text{Suc } j. b (\text{Suc } j + i)$ )))
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 ⇒ nat
  x :: nat

```

```

lemma Example-2:  $0 < n \implies$ 
 $\| - . \{ 'x = 0 \wedge (\sum_{i=0..<n} 'c \ i) = 0 \}.$ 
  COBEGIN
    SCHEME [ $0 \leq i < n$ ]
     $. \{ 'x = (\sum_{i=0..<n} 'c \ i) \wedge 'c \ i = 0 \}.$ 
     $\langle 'x := 'x + (\text{Suc } 0),, 'c := 'c \ (i := (\text{Suc } 0)) \rangle$ 
     $. \{ 'x = (\sum_{i=0..<n} 'c \ i) \wedge 'c \ i = (\text{Suc } 0) \}.$ 
  COEND
   $. \{ 'x = n \}.$ 
apply oghore
apply (simp-all cong del: strong-setsum-cong)
apply (tactic  $\ll \text{ALLGOALS } (\text{clarify-tac } @\{\text{claset}\}) \gg$ )
apply (simp-all cong del: strong-setsum-cong)
  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*

types

nodes = *node list*

edge = $\text{nat} \times \text{nat}$

edges = *edge list*

consts *Roots* :: *nat set*

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

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

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

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

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

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

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

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

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

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

$$\forall x \in \text{Roots}\}$$

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*]:

$$\llbracket \text{Roots} \subseteq \text{Blacks } M; \forall i < \text{length } E. \neg \text{BtoW}(E!i, M) \rrbracket$$

$$\implies 1 < \text{length } \text{path} \longrightarrow (\text{path}!(\text{length } \text{path} - 1)) \in \text{Roots} \longrightarrow$$

$$(\forall i < \text{length } \text{path} - 1. (\exists j. j < \text{length } E \wedge E!j = (\text{path}!(\text{Suc } i), \text{path}!i)))$$

$$\longrightarrow M!(\text{path}!0) = \text{Black}$$

apply (*induct-tac path*)
apply *force*
apply *clarify*
apply *simp*
apply (*case-tac list*)
apply *force*
apply *simp*
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*:

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

$$\implies \text{Reach } E \subseteq \text{Blacks } M$$

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


```

apply clarify
apply(rule conjI)
apply(subgoal-tac  $0 < \text{length path} - \text{Suc } 0$ )
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]:
   $P (n::\text{nat}) \longrightarrow (\exists m. P m \wedge (\forall i. i < m \longrightarrow \neg P i))$ 
apply(rule nat-less-induct)
apply clarify
apply(case-tac  $\forall m. m < n \longrightarrow \neg P m$ )
apply auto
done

```

```

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

```

```

lemma Ex-last-occurrence:
   $\llbracket P (n::\text{nat}); n \leq l \rrbracket \implies (\exists m. P (l - m) \wedge (\forall i. i < m \longrightarrow \neg P (l - i)))$ 
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  $\forall z < \text{length path}. \text{fst}(E!R) \neq \text{path}!z$ )
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(drule nth-drop)
apply simp
apply(subgoal-tac  $\text{length path} - \text{Suc } m + \text{nat} = \text{length path} - \text{Suc } 0$ )
  prefer 2 apply arith
apply simp
apply clarify
apply(case-tac i)
  apply(force simp add: nth-list-update)
apply simp
apply(subgoal-tac  $(\text{length path} - \text{Suc } m) + \text{nata} \leq \text{length path}$ )
  prefer 2 apply arith
apply(subgoal-tac  $(\text{length path} - \text{Suc } m) + (\text{Suc nata}) \leq \text{length path}$ )
  prefer 2 apply arith
apply simp
apply(erule-tac  $x = \text{length path} - \text{Suc } m + \text{nata}$  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  $\text{Suc nata} \leq \text{nat}$ )
  prefer 2 apply arith
apply(drule-tac  $n = \text{Suc nata}$  in Compl-lemma)
apply clarify
using [[linarith-split-limit = 0]]
apply force
using [[linarith-split-limit = 9]]
apply(drule leI)

```

```

apply(subgoal-tac Suc (length path - Suc m + nata)=(length path - Suc 0) -
(m - Suc nata))
apply(erule-tac x = m - (Suc nata) in allE)
apply(case-tac m)
  apply simp
  apply simp
apply simp
done

```

Graph 3

```

declare min-max.inf-absorb1 [simp] min-max.inf-absorb2 [simp]

```

lemma *Graph3*:

```

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

$\llbracket T \in \text{Reach } E; \text{Roots} \subseteq \text{Blacks } M; I \leq \text{length } E; T < \text{length } M; R < \text{length } E;$

$\forall i < I. \neg BtoW(E!i, M); R < I; M!fst(E!R) = Black; M!T \neq Black \rrbracket \implies$
 $(\exists r. I \leq r \wedge r < \text{length } E \wedge BtoW(E[R := (fst(E!R), T)]!r, M))$
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* $\exists m < \text{length } path. M!(path!m) = Black$)
apply (*erule exE*)
apply (*drule-tac* $P = \lambda m. m < \text{length } path \wedge M!(path!m) = Black$ **in** *Ex-first-occurrence*)
apply *clarify*
apply (*case-tac* *ma*)
apply *force*
apply *simp*
apply (*case-tac* $\text{length } path$)
apply *force*
apply *simp*
apply (*erule-tac* $P = \lambda i. i < \text{nat} \longrightarrow ?P\ i$ **and** $x = \text{nat}$ **in** *allE*)
apply *simp*
apply *clarify*
apply (*erule-tac* $P = \lambda i. i < \text{Suc } \text{nat} \longrightarrow ?P\ i$ **and** $x = \text{nat}$ **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* $x = \text{length } path - 1$ **in** *allE*)
apply (*case-tac* $\text{length } path$)
apply *force*
apply *force*
done

declare *min-max.inf-absorb1* [*simp del*] *min-max.inf-absorb2* [*simp del*]

Graph 5

lemma *Graph5*:

$\rrbracket T \in \text{Reach } E ; \text{Roots} \subseteq \text{Blacks } M; \forall i < R. \neg BtoW(E!i, M); T < \text{length } M;$
 $R < \text{length } E; M!fst(E!R) = Black; M!snd(E!R) = Black; M!T \neq Black \rrbracket$
 $\implies (\exists r. R < r \wedge r < \text{length } E \wedge BtoW(E[R := (fst(E!R), T)]!r, M))$
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  $x = \text{nat}$  in allE)
apply simp
apply clarify
apply(erule-tac  $P = \lambda i. i < \text{Suc } \text{nat} \longrightarrow ?P\ i$  and  $x = \text{nat}$  in allE)
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 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

```

Other lemmas about graphs

lemma *Graph6*:

```

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

```

lemma *Graph7*:

```

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

```

lemma *Graph8*:

```

 $\llbracket \text{Proper-Roots}(M) \rrbracket \implies \text{Proper-Roots}(M[T := a])$ 
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 :: nat
  z :: 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 :: nat T :: 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 Mut-init :: gar-coll-state  $\Rightarrow$  bool where
  Mut-init  $\equiv \ll T \in \text{Reach } 'E \wedge R < \text{length } 'E \wedge T < \text{length } 'M \gg$ 

```

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 Redirect-Edge :: gar-coll-state ann-com where
  Redirect-Edge  $\equiv \cdot \{ 'Mut-init \wedge 'z \}. \langle 'E := 'E[R := (\text{fst } 'E!R), T] \rangle, 'z := (\neg 'z) \rangle$ 

```

```

definition Color-Target :: gar-coll-state ann-com where
  Color-Target  $\equiv \cdot \{ 'Mut-init \wedge \neg 'z \}. \langle 'M := 'M[T := \text{Black}] \rangle, 'z := (\neg 'z) \rangle$ 

```

```

definition Mutator :: gar-coll-state ann-com where
  Mutator  $\equiv$ 
     $\cdot \{ 'Mut-init \wedge 'z \}.$ 
    WHILE True INV  $\cdot \{ 'Mut-init \wedge 'z \}.$ 
    DO Redirect-Edge ;; Color-Target OD

```

Correctness of the mutator

```

lemmas mutator-defs = Mut-init-def Redirect-Edge-def Color-Target-def

```

```

lemma Redirect-Edge:
   $\vdash \text{Redirect-Edge } \text{pre}(\text{Color-Target})$ 
apply (unfold mutator-defs)
apply annhoare
apply (simp-all)

```



```

apply(force elim:Graph2)
done

```

```

lemma Color-Target:
   $\vdash \text{Color-Target} .\{ 'Mut-init \wedge 'z \}.$ 
apply (unfold mutator-defs)
apply annhoare
apply(simp-all)
done

```

```

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

```

2.2.2 The Collector

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

```

consts  $M\text{-init} :: \text{nodes}$ 

```

```

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

```

```

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

```

```

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

```

```

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

```

Blackening the roots

```

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

```

.{ 'Proper $\wedge (\forall i < 'ind + 1. i \in Roots \longrightarrow 'M!i = Black) \wedge 'ind < length\ 'M$ }.
 'ind := 'ind + 1
 OD

lemma *Blacken-Roots*:
 $\vdash Blacken-Roots \{ 'Proper \wedge Roots \subseteq Blacks\ 'M \}$.
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**
 $PBInv \equiv \ll \lambda ind. 'obc < Blacks\ 'M \vee (\forall i < ind. \neg BtoW\ ('E!i, 'M) \vee$
 $(\neg 'z \wedge i = R \wedge (snd\ ('E!R)) = T \wedge (\exists r. ind \leq r \wedge r < length\ 'E \wedge BtoW\ ('E!r, 'M)))) \gg$

definition *Propagate-Black-aux* :: *gar-coll-state* *ann-com* **where**
 $Propagate-Black-aux \equiv$
 $\{ 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M \}$.
 'ind := 0;;
 $\{ 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M \wedge 'ind = 0 \}$.

WHILE 'ind < length 'E
 INV .{ 'Proper $\wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$
 $\wedge 'PBInv\ 'ind \wedge 'ind \leq length\ 'E$ }.
 DO .{ 'Proper $\wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$
 $\wedge 'PBInv\ 'ind \wedge 'ind < length\ 'E$ }.
 IF 'M!(fst ('E!'ind)) = Black THEN
 .{ 'Proper $\wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$
 $\wedge 'PBInv\ 'ind \wedge 'ind < length\ 'E \wedge 'M!fst('E!'ind) = Black$ }.
 'M := 'M[snd ('E!'ind) := Black];;
 .{ 'Proper $\wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$
 $\wedge 'PBInv\ ('ind + 1) \wedge 'ind < length\ 'E$ }.
 'ind := 'ind + 1
 FI
 OD

lemma *Propagate-Black-aux*:
 $\vdash Propagate-Black-aux$
 $\{ 'Proper \wedge Roots \subseteq Blacks\ 'M \wedge 'obc \subseteq Blacks\ 'M \wedge 'bc \subseteq Blacks\ 'M$
 $\wedge ('obc < Blacks\ 'M \vee 'Safe) \}$.
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 (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**

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

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

Propagate-Black \equiv

.{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M }.

'ind := 0;;

.{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M \wedge 'ind = 0 }.

WHILE 'ind < length 'E

INV .{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge 'PBIInv 'ind \wedge 'ind \leq length 'E }.

DO .{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge 'PBIInv 'ind \wedge 'ind < length 'E }.

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

.{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge 'PBIInv 'ind \wedge 'ind < length 'E \wedge ('M!(fst ('E! 'ind))) = Black }.

'k := (snd ('E! 'ind));;

.{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge 'PBIInv 'ind \wedge 'ind < length 'E \wedge ('M!(fst ('E! 'ind))) = Black
 \wedge 'Auxk }.

< 'M := 'M['k := Black],, 'ind := 'ind + 1 >

ELSE .{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge 'PBIInv 'ind \wedge 'ind < length 'E }.

< IF ('M!(fst ('E! 'ind))) \neq Black THEN 'ind := 'ind + 1 FI >

FI

OD

lemma *Propagate-Black*:

\vdash *Propagate-Black*

.{ 'Proper \wedge Roots \subseteq Blacks 'M \wedge 'obc \subseteq Blacks 'M \wedge 'bc \subseteq Blacks 'M
 \wedge ('obc < Blacks 'M \vee 'Safe) }.

apply (unfold *Propagate-Black-def* *PBIInv-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 (ind x)≤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≤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 (ind x)≤r)
        apply fast
        apply arith
      apply (simp add: BtoW-def)
    apply (simp add: BtoW-def)
  — last

```

```

apply clarify
apply simp
apply(subgoal-tac 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 $\equiv \ll \lambda ind. \{i. i < ind \wedge 'Ma!i = Black\} \subseteq 'bc \gg$

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*:

```

  ⊢ Count
  .{ 'Proper ∧ Roots ⊆ Blacks 'M
    ∧ 'obc ⊆ Blacks 'Ma ∧ Blacks 'Ma ⊆ 'bc ∧ 'bc ⊆ Blacks 'M ∧ length 'Ma = length
    'M
    ∧ ('obc < Blacks 'Ma ∨ 'Safe)}.
apply(unfold Count-def)
apply annhoare
apply(simp-all add:CountInv-def Graph6 Graph7 Graph8 Graph12 Blacks-def collector-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

consts *Append-to-free* :: nat × edges ⇒ edges

axioms

```

Append-to-free0: length (Append-to-free (i, e)) = length e
Append-to-free1: Proper-Edges (m, e)
  ⇒ Proper-Edges (m, Append-to-free(i, e))
Append-to-free2: i ∉ Reach e
  ⇒ n ∈ Reach (Append-to-free(i, e)) = ( n = i ∨ n ∈ Reach e)

```

definition *AppendInv* :: gar-coll-state ⇒ nat ⇒ bool **where**

AppendInv ≡ «λind. ∀ i < length 'M. ind ≤ i ⟶ i ∈ Reach 'E ⟶ 'M!i = Black»

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

```

Append ≡
.{ 'Proper ∧ Roots ⊆ Blacks 'M ∧ 'Safe}.
'ind := 0;;
.{ 'Proper ∧ Roots ⊆ Blacks 'M ∧ 'Safe ∧ 'ind = 0}.
WHILE 'ind < length 'M
  INV .{ 'Proper ∧ 'AppendInv 'ind ∧ 'ind ≤ length 'M}.
DO .{ 'Proper ∧ 'AppendInv 'ind ∧ 'ind < length 'M}.
  IF 'M!'ind = Black THEN
    .{ 'Proper ∧ 'AppendInv 'ind ∧ 'ind < length 'M ∧ 'M!'ind = Black}.
    'M := 'M['ind := White]

```

```

ELSE .{ 'Proper  $\wedge$  'AppendInv 'ind  $\wedge$  'ind < length 'M  $\wedge$  'ind  $\notin$  Reach 'E }.
      'E := Append-to-free('ind, 'E)
FI;;
.{ 'Proper  $\wedge$  'AppendInv ('ind+1)  $\wedge$  'ind < length 'M }.
  'ind := 'ind+1
OD

```

lemma *Append*:

```

 $\vdash$  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  $\equiv$ 
.{ 'Proper }.
  WHILE True INV .{ 'Proper }.
  DO
    Blacken-Roots;;
    .{ 'Proper  $\wedge$  Roots  $\subseteq$  Blacks 'M }.
      'obc := {};
    .{ 'Proper  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {} }.
      'bc := Roots;;
    .{ 'Proper  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {}  $\wedge$  'bc = Roots }.
      'Ma := M-init;;
    .{ 'Proper  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {}  $\wedge$  'bc = Roots  $\wedge$  'Ma = M-init }.
      WHILE 'obc  $\neq$  'bc
        INV .{ '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) }.
        DO .{ 'Proper  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M }.

```



```

      'obc:='bc;;
      Propagate-Black;;
      .{ 'Proper  $\wedge$  Roots $\subseteq$ Blacks 'M  $\wedge$  'obc $\subseteq$ Blacks 'M  $\wedge$  'bc $\subseteq$ Blacks 'M
         $\wedge$  ('obc < Blacks 'M  $\vee$  'Safe)}.
      'Ma:='M;;
      .{ '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)}.
      'bc:={};;
      Count
    OD;;
  Append
OD

```

```

lemma Collector:
   $\vdash$  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 mutator-defs)
apply(tactic << TRYALL (interfree-aux-tac) >>))
— 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 @{claset}) >>))
apply(simp-all add:Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12)
apply(tactic << TRYALL (etac disjE) >>))

```

```

apply simp-all
apply(tactic ⟨ TRYALL(EVERY '[rtac disjI2,rtac subset-trans,etac @{thm Graph3},force-tac
@{clasimpset}, assume-tac]) ⟩)
apply(tactic ⟨ TRYALL(EVERY '[rtac disjI2,etac subset-trans,rtac @{thm Graph9},force-tac
@{clasimpset}]) ⟩)
apply(tactic ⟨ TRYALL(EVERY '[rtac disjI1,etac @{thm psubset-subset-trans},rtac
@{thm Graph9},force-tac @{clasimpset}]) ⟩)
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 mutator-defs)
apply(tactic ⟨ TRYALL (interfree-aux-tac) ⟩)
— 64 subgoals left
apply(simp-all add:nth-list-update Invariants Append-to-free0) +
apply(tactic⟨ TRYALL (clarify-tac @{claset}) ⟩)
— 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 ∧ 'Mut-init ∧ '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 $Mul\text{-}mut\text{-}init :: mul\text{-}gar\text{-}coll\text{-}state \Rightarrow nat \Rightarrow bool$ **where**
 $Mul\text{-}mut\text{-}init \equiv \llbracket \lambda n. n = length \ 'Muts \wedge (\forall i < n. R \ (\ 'Muts!i) < length \ 'E$
 $\wedge T \ (\ 'Muts!i) < length \ 'M) \rrbracket$

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

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

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

```

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
  .{ 'Mul-mut-init n  $\wedge$  Z ( '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 .{ 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-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-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  $\llbracket \text{TRYALL } (\text{interfree-aux-tac}) \rrbracket$ )
  apply (tactic  $\llbracket \text{ALLGOALS } (\text{clarify-tac } @\{\text{claset}\}) \rrbracket$ )
  apply (simp-all add: nth-list-update)
done

```

Modular Parameterized Mutators

lemma *Mul-Parameterized-Mutators*: $0 < n \implies$

```

   $\llbracket - . \{ ' \text{Mul-mut-init } n \wedge (\forall i < n. Z ( ' \text{Muts! } i)) \} .$ 
  COBEGIN
  SCHEME  $[0 \leq j < n]$ 
    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 $\equiv \ll \text{length } (\text{filter } (\lambda i. \neg Z\ i \wedge 'M!(T\ i) \neq \text{Black})\ 'Muts) \gg$

consts *M-init* :: *nodes*

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

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

definition *Safe* :: *mul-gar-coll-state* \Rightarrow *bool* **where**
Safe $\equiv \ll \text{Reach } 'E \subseteq \text{Blacks } 'M \gg$

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* }.
 ' *ind* := 0;;
 .{ ' *Mul-Proper* *n* \wedge ' *ind* = 0 }.
 WHILE ' *ind* < length ' *M*
 INV .{ ' *Mul-Proper* *n* \wedge ($\forall i < 'ind. i \in \text{Roots} \longrightarrow 'M!i = \text{Black}$) \wedge ' *ind* \leq length ' *M* }.
 DO .{ ' *Mul-Proper* *n* \wedge ($\forall i < 'ind. i \in \text{Roots} \longrightarrow 'M!i = \text{Black}$) \wedge ' *ind* < length ' *M* }.
 IF ' *ind* \in Roots THEN
 .{ ' *Mul-Proper* *n* \wedge ($\forall i < 'ind. i \in \text{Roots} \longrightarrow 'M!i = \text{Black}$) \wedge ' *ind* < length ' *M* \wedge ' *ind* \in Roots }.
 ' *M* := ' *M* [' *ind* := Black] FI;;
 .{ ' *Mul-Proper* *n* \wedge ($\forall i < 'ind + 1. i \in \text{Roots} \longrightarrow 'M!i = \text{Black}$) \wedge ' *ind* < length ' *M* }.
 ' *ind* := ' *ind* + 1
 OD

lemma *Mul-Blacken-Roots*:

```

  ⊢ Mul-Blacken-Roots n
  .{ ' Mul-Prop n ∧ Roots ⊆ 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* ⇒ *bool* **where**
Mul-PBInv ≡ « ' *Safe* ∨ ' *obc* ⊆ *Blacks* ' *M* ∨ ' *l* < ' *Queue*
 ∨ (∀ *i* < ' *ind*. ¬ *BtoW* (' *E*! *i*, ' *M*)) ∧ ' *l* ≤ ' *Queue* »

definition *Mul-Auxk* :: *mul-gar-coll-state* ⇒ *bool* **where**
Mul-Auxk ≡ « ' *l* < ' *Queue* ∨ ' *M*! *k* ≠ *Black* ∨ ¬ *BtoW* (' *E*! ' *ind*, ' *M*) ∨ ' *obc* ⊆ *Blacks*
 ' *M* »

definition *Mul-Propagate-Black* :: *nat* ⇒ *mul-gar-coll-state* *ann-com* **where**
Mul-Propagate-Black *n* ≡
 .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M* ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ (' *Safe* ∨ ' *l* ≤ ' *Queue* ∨ ' *obc* ⊆ *Blacks* ' *M*) }.
 ' *ind* := 0;;
 .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M*
 ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ *Blacks* ' *M* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ (' *Safe* ∨ ' *l* ≤ ' *Queue* ∨ ' *obc* ⊆ *Blacks* ' *M*) ∧ ' *ind* = 0 }.
WHILE ' *ind* < *length* ' *E*
INV .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M*
 ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ ' *Mul-PBInv* ∧ ' *ind* ≤ *length* ' *E* }.
DO .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M*
 ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ ' *Mul-PBInv* ∧ ' *ind* < *length* ' *E* }.
IF ' *M*! (*fst* (' *E*! ' *ind*)) = *Black* **THEN**
 .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M*
 ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ ' *Mul-PBInv* ∧ (' *M*! *fst* (' *E*! ' *ind*)) = *Black* ∧ ' *ind* < *length* ' *E* }.
 ' *k* := *snd* (' *E*! ' *ind*);;
 .{ ' *Mul-Prop* *n* ∧ *Roots* ⊆ *Blacks* ' *M*
 ∧ ' *obc* ⊆ *Blacks* ' *M* ∧ ' *bc* ⊆ *Blacks* ' *M*
 ∧ (' *Safe* ∨ ' *obc* ⊆ *Blacks* ' *M* ∨ ' *l* < ' *Queue* ∨ (∀ *i* < ' *ind*. ¬ *BtoW* (' *E*! *i*, ' *M*))
 ∧ ' *l* ≤ ' *Queue* ∧ ' *Mul-Auxk*) ∧ ' *k* < *length* ' *M* ∧ ' *M*! *fst* (' *E*! ' *ind*) = *Black*
 ∧ ' *ind* < *length* ' *E* }.
 ⟨ ' *M* := ' *M* [' *k* := *Black*], ' *ind* := ' *ind* + 1 ⟩

$ELSE \{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge 'Mul-PBInv \wedge 'ind < length \text{ ' } E \}.$
 $\langle IF \text{ ' } M!(fst \text{ (' } E! \text{ ' } ind)) \neq Black \text{ THEN ' } ind := 'ind + 1 \text{ FI} \rangle FI$
 OD

lemma *Mul-Propagate-Black:*

$\vdash Mul-Propagate-Black \text{ } n$
 $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M \wedge 'obc \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } M \vee 'l < 'Queue \wedge ('l \leq 'Queue \vee 'obc \subseteq Blacks$
 $\text{ ' } M)) \}.$
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 $\equiv \ll \lambda ind. \{i. i < ind \wedge 'Ma!i=Black\} \subseteq 'bc \gg$

definition *Mul-Count* :: *nat* \Rightarrow *mul-gar-coll-state* *ann-com* **where**
Mul-Count *n* \equiv
 $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks \text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'bc = \{\}$ }.
 $'ind := 0;$
 $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks \text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'bc = \{\} \wedge 'ind = 0$ }.
WHILE $'ind < length \text{ ' } M$
 $INV \{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M \wedge 'Mul-CountInv \text{ ' } ind$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks$
 $\text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'ind \leq length \text{ ' } M$ }.
DO $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M \wedge 'Mul-CountInv \text{ ' } ind$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks \text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'ind < length \text{ ' } M$ }.
IF $'M! 'ind = Black$
THEN $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M \wedge 'Mul-CountInv \text{ ' } ind$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks$
 $\text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'ind < length \text{ ' } M \wedge 'M! 'ind = Black$ }.
 $'bc := insert \text{ ' } ind \text{ ' } bc$
 $FI;$
 $\{ 'Mul-Prop\text{er } n \wedge Roots \subseteq Blacks \text{ ' } M$
 $\wedge 'obc \subseteq Blacks \text{ ' } Ma \wedge Blacks \text{ ' } Ma \subseteq Blacks \text{ ' } M \wedge 'bc \subseteq Blacks \text{ ' } M$
 $\wedge length \text{ ' } Ma = length \text{ ' } M \wedge 'Mul-CountInv \text{ ' } (ind+1)$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \text{ ' } Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks \text{ ' } M))$
 $\wedge 'q < n+1 \wedge 'ind < length \text{ ' } M$ }.
 $'ind := 'ind + 1$

OD

lemma *Mul-Count*:

```

  ⊢ Mul-Count n
  .{ 'Mul-Prop 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 }.
  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

consts *Append-to-free* :: *nat* × *edges* ⇒ *edges*

axioms

Append-to-free0: *length* (*Append-to-free* (*i*, *e*)) = *length e*

Append-to-free1: *Proper-Edges* (*m*, *e*)

⇒ *Proper-Edges* (*m*, *Append-to-free*(*i*, *e*))

Append-to-free2: *i* ∉ *Reach e*

⇒ *n* ∈ *Reach* (*Append-to-free*(*i*, *e*)) = (*n* = *i* ∨ *n* ∈ *Reach e*)

definition *Mul-AppendInv* :: *mul-gar-coll-state* ⇒ *nat* ⇒ *bool* **where**

Mul-AppendInv ≡ << λ*ind*. (∀ *i*. *ind* ≤ *i* → *i* < *length* 'M → *i* ∈ *Reach* 'E → 'M!*i*=Black)>>

definition *Mul-Append* :: *nat* ⇒ *mul-gar-coll-state* *ann-com* **where**

Mul-Append n ≡

.{ 'Mul-*Proper n* ∧ *Roots* ⊆ *Blacks* 'M ∧ 'Safe }.

'ind:=0;;

.{ 'Mul-*Proper n* ∧ *Roots* ⊆ *Blacks* 'M ∧ 'Safe ∧ 'ind=0 }.

WHILE 'ind < *length* 'M

INV .{ 'Mul-*Proper n* ∧ 'Mul-AppendInv 'ind ∧ 'ind ≤ *length* 'M }.

DO .{ 'Mul-*Proper n* ∧ 'Mul-AppendInv 'ind ∧ 'ind < *length* 'M }.

IF 'M!'ind=Black THEN

.{ 'Mul-*Proper n* ∧ 'Mul-AppendInv 'ind ∧ 'ind < *length* 'M ∧ 'M!'ind=Black }.

'M:= 'M['ind:=White]

ELSE

.{ 'Mul-*Proper n* ∧ 'Mul-AppendInv 'ind ∧ 'ind < *length* 'M ∧ 'ind ∉ *Reach*

'E }.

'E:=*Append-to-free*('ind, 'E)

FI;;

.{ 'Mul-*Proper n* ∧ 'Mul-AppendInv ('ind+1) ∧ 'ind < *length* 'M }.

'ind:='ind+1

OD

lemma *Mul-Append*:

⊢ *Mul-Append n*

.{ 'Mul-*Proper 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  $\equiv$ 
    .{ 'Mul-Propser n }.
    WHILE True INV .{ 'Mul-Propser n }.
    DO
      Mul-Blacken-Roots n ;;
      .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M }.
      'obc := {};
      .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {} }.
      'bc := Roots;
      .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {}  $\wedge$  'bc = Roots }.
      'l := 0;
      .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'obc = {}  $\wedge$  'bc = Roots  $\wedge$  'l = 0 }.
      WHILE 'l < n + 1
        INV .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M  $\wedge$ 
          ( 'Safe  $\vee$  ( 'l  $\leq$  'Queue  $\vee$  'bc  $\subset$  Blacks 'M )  $\wedge$  'l < n + 1 ) }.
        DO .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
           $\wedge$  ( 'Safe  $\vee$  'l  $\leq$  'Queue  $\vee$  'bc  $\subset$  Blacks 'M ) }.
          'obc := 'bc;
          Mul-Propagate-Black n;;
          .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M
             $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
             $\wedge$  ( 'Safe  $\vee$  'obc  $\subset$  Blacks 'M  $\vee$  'l < 'Queue
               $\wedge$  ( 'l  $\leq$  'Queue  $\vee$  'obc  $\subset$  Blacks 'M ) ) }.
            'bc := {};
            .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M
               $\wedge$  'obc  $\subseteq$  Blacks 'M  $\wedge$  'bc  $\subseteq$  Blacks 'M
               $\wedge$  ( 'Safe  $\vee$  'obc  $\subset$  Blacks 'M  $\vee$  'l < 'Queue
                 $\wedge$  ( 'l  $\leq$  'Queue  $\vee$  'obc  $\subset$  Blacks 'M ) )  $\wedge$  'bc = {} }.
                ( 'Ma := 'M,, 'q := 'Queue );
                Mul-Count n;;
                .{ 'Mul-Propser 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  $\wedge$  Blacks 'Ma  $\subseteq$  'bc
                   $\wedge$  ( 'Safe  $\vee$  'obc  $\subset$  Blacks 'Ma  $\vee$  'l < 'q  $\wedge$  ( 'q  $\leq$  'Queue  $\vee$  'obc  $\subset$  Blacks 'M ) )
                   $\wedge$  'q < n + 1 }.
                  IF 'obc = 'bc THEN
                    .{ 'Mul-Propser 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  $\wedge$  Blacks 'Ma  $\subseteq$  'bc
                       $\wedge$  ( 'Safe  $\vee$  'obc  $\subset$  Blacks 'Ma  $\vee$  'l < 'q  $\wedge$  ( 'q  $\leq$  'Queue  $\vee$  'obc  $\subset$  Blacks 'M ) )
                       $\wedge$  'q < n + 1  $\wedge$  'obc = 'bc }.
                      'l := 'l + 1
                    ELSE .{ 'Mul-Propser n  $\wedge$  Roots  $\subseteq$  Blacks 'M

```

$\wedge 'obc \subseteq Blacks \ 'Ma \wedge Blacks \ 'Ma \subseteq Blacks \ 'M \wedge 'bc \subseteq Blacks \ 'M$
 $\wedge length \ 'Ma = length \ 'M \wedge Blacks \ 'Ma \subseteq 'bc$
 $\wedge ('Safe \vee 'obc \subseteq Blacks \ 'Ma \vee 'l < 'q \wedge ('q \leq 'Queue \vee 'obc \subseteq Blacks$
 $\ 'M))$
 $\wedge 'q < n+1 \wedge 'obc \neq 'bc\}.$
 $'l := 0 \ FI$
 $OD;;$
 $Mul-Append \ n$
 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*:
 $\vdash \text{Mul-Collector } n$
 $\cdot \{False\}.$
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*]:
 $\forall i. (\neg P \ (list!i) \vee P \ j) \wedge i < length \ list$
 $\longrightarrow length(filter \ P \ list) \leq length(filter \ P \ (list[i:=j]))$
apply(*induct-tac* *list*)
apply(*simp*)
apply(*clarify*)
apply(*case-tac* *i*)
apply(*simp*)
apply(*simp*)
done

lemma *less-length-filter-update* [*rule-format*]:
 $\forall i. P \ j \wedge \neg(P \ (list!i)) \wedge i < length \ list$

```

    → length(filter P list) < length(filter P (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  $\ll$  ALLGOALS (clarify-tac  $\@ \{ \text{claset} \}$ )  $\gg$ )
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  $\ll$  ALLGOALS (clarify-tac  $\@ \{ \text{claset} \}$ )  $\gg$ )
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* \ll *ALLGOALS* (*clarify-tac* @{*claset*}) \gg)
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* \ll *ALLGOALS* (*clarify-tac* @{*claset*}) \gg)
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* \ll *TRYALL* (*interfree-aux-tac*) \gg)
— 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* \ll *TRYALL* (*clarify-tac* @{*claset*}) \gg)
apply(*simp-all add:Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12*)
apply(*tactic* \ll *TRYALL* (*rtac conjI*) \gg)
apply(*tactic* \ll *TRYALL* (*rtac impI*) \gg)
apply(*tactic* \ll *TRYALL* (*etac disjE*) \gg)
apply(*tactic* \ll *TRYALL* (*etac conjE*) \gg)
apply(*tactic* \ll *TRYALL* (*etac disjE*) \gg)
apply(*tactic* \ll *TRYALL* (*etac disjE*) \gg)
 — 72 subgoals left
apply(*simp-all add:Graph6 Graph7 Graph8 Append-to-free0 Append-to-free1 Graph12*)
 — 35 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*rtac disjI1*, *rtac subset-trans*, *etac* @{*thm Graph3*}, *force-tac* @{*clasimpset*}, *assume-tac*] \gg)
 — 28 subgoals left
apply(*tactic* \ll *TRYALL* (*etac conjE*) \gg)
apply(*tactic* \ll *TRYALL* (*etac disjE*) \gg)
 — 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* \ll *TRYALL*(*EVERY* '[*REPEAT o* (*rtac disjI2*), *etac* (*thm subset-psubset-trans*), *etac* (*thm Graph11*), *force-tac* @{*clasimpset*}] \gg)
 — 41 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*rtac disjI2*, *rtac disjI1*, *etac* @{*thm le-trans*}, *force-tac* (@{*claset*}, @{*simpset*} *addsimps* [@{*thm Queue-def*}, @{*thm less-Suc-eq-le*}, @{*thm le-length-filter-update*}]]) \gg)
 — 35 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*rtac disjI2*, *rtac disjI1*, *etac* (*thm psubset-subset-trans*), *rtac* (*thm Graph9*), *force-tac* @{*clasimpset*}] \gg)
 — 31 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*rtac disjI2*, *rtac disjI1*, *etac* (*thm subset-psubset-trans*), *etac* (*thm Graph11*), *force-tac* @{*clasimpset*}] \gg)
 — 29 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*REPEAT o* (*rtac disjI2*), *etac* (*thm subset-psubset-trans*), *etac* (*thm subset-psubset-trans*), *etac* (*thm Graph11*), *force-tac* @{*clasimpset*}] \gg)
 — 25 subgoals left
apply(*tactic* \ll *TRYALL*(*EVERY* '[*rtac disjI2*, *rtac disjI2*, *rtac disjI1*, *etac* @{*thm le-trans*}, *force-tac* (@{*claset*}, @{*simpset*} *addsimps* [@{*thm Queue-def*}, @{*thm less-Suc-eq-le*}, @{*thm le-length-filter-update*}]]) \gg)
 — 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*: $j < n \implies$
 $\text{interfree-aux } (\text{Some } (\text{Mul-Mutator } j \ n), \{\}, \text{Some } (\text{Mul-Collector } n))$
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* $\ll \text{TRYALL } (\text{interfree-aux-tac}) \gg$)
— 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*:
 $\| - . \{ ' \text{Mul-Prop} \text{er } n \wedge ' \text{Mul-mut-init } n \wedge (\forall i < n. Z (' \text{Muts! } i)) \} .$
COBEGIN
 $\text{Mul-Collector } n$
 $. \{ \text{False} \} .$
 $\|$
SCHEME $[0 \leq j < n]$
 $\text{Mul-Mutator } j \ n$
 $. \{ \text{False} \} .$
COEND
 $. \{ \text{False} \} .$
apply *oghoare*
— 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*.

types

'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*

types *'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

types *'a conf* = ((*'a com*) *option*) \times *'a*

inductive-set

etran :: (*'a conf* \times *'a conf*) *set*

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

where

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

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

Component transitions

inductive-set

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

where

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

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

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

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

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

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

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

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

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

monos $rtrancl\ mono$

3.2.2 Semantics of Parallel Programs

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

inductive-set

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

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 $\text{par-ctran}' :: ['a \text{ par-conf}, 'a \text{ par-conf}] \Rightarrow \text{bool} \ (- \text{pc} \rightarrow - [81,81] \ 80)$
where
 $P \text{ -pc} \rightarrow Q \equiv (P, Q) \in \text{par-ctran}$
 $| \text{ParComp}: \llbracket i < \text{length } Ps; (Ps!i, s) \text{ -c} \rightarrow (r, t) \rrbracket \Longrightarrow (Ps, s) \text{ -pc} \rightarrow (Ps[i:=r], t)$
lemma $\text{par-ctranE}: c \text{ -pc} \rightarrow c' \Longrightarrow$
 $(\bigwedge i \text{ Ps } s \ r \ t. c = (Ps, s) \Longrightarrow c' = (Ps[i := r], t) \Longrightarrow i < \text{length } Ps \Longrightarrow$
 $(Ps ! i, s) \text{ -c} \rightarrow (r, t) \Longrightarrow P) \Longrightarrow P$
by ($\text{induct } c, \text{induct } c', \text{erule } \text{par-ctran.cases}, \text{blast}$)

3.2.3 Computations

Sequential computations

types $'a \text{ confs} = ('a \text{ conf}) \text{ list}$

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

where

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

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

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

Parallel computations

types $'a \text{ par-confs} = ('a \text{ par-conf}) \text{ list}$

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

where

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

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

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

3.2.4 Modular Definition of Computation

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

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

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

where

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

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

3.2.5 Equivalence of Both Definitions.

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

apply *simp*

apply (*induct xs, simp+*)

apply (*case-tac xs*)

apply *simp-all*

done

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]:
   $\forall s \ Q \ t \ xs. ((Some \ a, s), \ Q, t) \in ctran \longrightarrow (Q, t) \# xs \in cptn-mod$ 
   $\longrightarrow (Some \ a, s) \# (Q, t) \# xs \in cptn-mod$ 
  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(drule div-seq,simp)
  apply force
  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]:

```

```

  ∀ b a. b# c1 ∈ cptn ⟶ a# c2 ∈ cptn ⟶ (b# c1)!length c1 = a ⟶ b# c1 @ c2 ∈ 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:  $\llbracket xs \neq []; fst(xs!(length\ xs - (Suc\ 0))) = None \rrbracket$ 
  ⟹  $fst((map\ (lift\ P)\ xs)!(length\ (map\ (lift\ P)\ xs) - (Suc\ 0))) = (Some\ P)$ 
apply(case-tac ( $xs \neq []$ ))
apply (simp add:lift-def)
done

```

```

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

```

```

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:  $xs \neq [] \implies$ 
   $snd(((map\ (lift\ P)\ xs)!(length\ (map\ (lift\ P)\ xs) - (Suc\ 0)))) = snd(xs!(length\ xs$ 
   $- (Suc\ 0)))$ 
apply(case-tac ( $xs \neq []$ ))
apply (simp add:lift-def)
done

```

```

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:  $i < length\ xs \implies snd(lift\ Q\ (xs\ !\ i)) = snd\ (xs\ !\ i)$ 
apply(case-tac  $xs\ !\ i$ )
apply(simp add:lift-def)
done

```

```

lemma cptn-if-cptn-mod:  $c \in \text{cptn-mod} \implies c \in \text{cptn}$ 
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(case-tac  $xs \neq []$ )
  apply(drule-tac  $P=P1$  in last-lift)
  apply(rule last-fst-esp)
  apply (simp add:last-length)
  apply(simp add:Cons-lift del:map.simps)
  apply(rule conjI, clarify, simp)
  apply(case-tac  $((\text{Some } P0, s) \# xs) \neq []$ )
  apply clarify
  apply (simp add:lift-def last-length)
apply (simp add:last-length)
— While1
apply(rule CptnComp)
apply(rule WhileT,simp)
apply(drule-tac  $P=\text{While } b \text{ } P$  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(case-tac xs≠[])
apply(drule-tac P=While b P in last-lift)
  apply(rule last-fst-esp,simp add:last-length)
apply(simp add:Cons-lift del:map.simps)
apply(rule conjI, clarify, simp)
apply(case-tac (((Some P, s) # xs) ! length xs))
apply clarify
apply (simp add:last-length lift-def)
apply simp
done

theorem cptn-iff-cptn-mod: (c ∈ cptn) = (c ∈ 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.

types 'a rgformula = 'a com × 'a set × ('a × 'a) set × ('a × 'a) set × 'a set

definition *assum* :: ('a set × ('a × 'a) set) ⇒ ('a confs) set **where**
assum ≡ λ(pre, rely). {c. snd(c!0) ∈ pre ∧ (∀ i. Suc i < length c ⟶
 c!i -e⟶ c!(Suc i) ⟶ (snd(c!i), snd(c!Suc i)) ∈ rely)}

definition *comm* :: (('a × 'a) set × 'a set) ⇒ ('a confs) set **where**
comm ≡ λ(guar, post). {c. (∀ i. Suc i < length c ⟶
 c!i -c⟶ c!(Suc i) ⟶ (snd(c!i), snd(c!Suc i)) ∈ guar) ∧
 (fst (last c) = None ⟶ snd (last c) ∈ post)}

definition *com-validity* :: 'a com ⇒ 'a set ⇒ ('a × 'a) set ⇒ ('a × 'a) set ⇒ 'a set ⇒ bool

(| = - sat [-, -, -, -] [60,0,0,0,0] 45) **where**
 | = P sat [pre, rely, guar, post] ≡
 ∀ s. cp (Some P) s ∩ assum(pre, rely) ⊆ comm(guar, post)

3.3.2 Validity for Parallel Programs.

definition *All-None* :: (('a com) option) list ⇒ 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**
 $par-assum \equiv \lambda(pre, rely). \{c. snd(c!0) \in pre \wedge (\forall i. Suc\ i < length\ c \longrightarrow$
 $c!i -pe \longrightarrow c!Suc\ i \longrightarrow (snd(c!i), snd(c!Suc\ i)) \in rely)\}$

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

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

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**
 $same-length\ c\ clist \equiv (\forall i < length\ clist. length(clist!i) = length\ c)$

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

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

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

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

Some previous lemmas

lemma $list-eq-if\ [rule-format]:$

$\forall ys. xs = ys \longrightarrow (length\ xs = length\ ys) \longrightarrow (\forall i < length\ xs. xs!i = ys!i)$

apply $(induct\ xs)$

apply $simp$

apply $clarify$

done

```

lemma list-eq: (length xs = 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:  $\llbracket ys!0 = a; ys \neq [] \rrbracket \implies ys = (a \# (tl \ ys))$ 
apply(case-tac ys)
  apply simp+
done

```

```

lemma nth-tl-if [rule-format]:  $ys \neq [] \longrightarrow ys!0 = a \longrightarrow P \ ys \longrightarrow P \ (a \# (tl \ ys))$ 
apply(induct ys)
  apply simp+
done

```

```

lemma nth-tl-onlyif [rule-format]:  $ys \neq [] \longrightarrow ys!0 = a \longrightarrow P \ (a \# (tl \ ys)) \longrightarrow P \ ys$ 
apply(induct ys)
  apply simp+
done

```

```

lemma seq-not-eq1: Seq c1 c2  $\neq$  c1
apply(rule com.induct)
  apply simp-all
  apply clarify
done

```

```

lemma seq-not-eq2: Seq c1 c2  $\neq$  c2
apply(rule com.induct)
  apply simp-all
  apply clarify
done

```

```

lemma if-not-eq1: Cond b c1 c2  $\neq$  c1
apply(rule com.induct)
  apply simp-all
  apply clarify
done

```

```

lemma if-not-eq2: Cond b c1 c2  $\neq$  c2
apply(rule com.induct)
  apply simp-all
  apply clarify
done

```

```

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 $c: (P,s) -c \rightarrow (Q,t)$
 shows $P \neq Q$ using c
 by (induct $x1 \equiv (P,s)$ $x2 \equiv (Q,t)$ arbitrary: $P \ s \ Q \ t$) auto

lemma *prog-not-eq-in-ctran [simp]*: $\neg (P,s) -c \rightarrow (P,t)$
 apply clarify
 apply (drule *prog-not-eq-in-ctran-aux*)
 apply simp
 done

lemma *prog-not-eq-in-par-ctran-aux [rule-format]*: $(P,s) -pc \rightarrow (Q,t) \implies (P \neq Q)$
 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]*: $\neg (P,s) -pc \rightarrow (P,t)$
 apply clarify
 apply (drule *prog-not-eq-in-par-ctran-aux*)
 apply simp
 done

lemma *tl-in-cptn*: $\llbracket a \# xs \in \text{cptn}; xs \neq [] \rrbracket \implies xs \in \text{cptn}$
 apply (force elim: *cptn.cases*)
 done

lemma *tl-zero [rule-format]*:
 $P (ys! \text{Suc } j) \longrightarrow \text{Suc } j < \text{length } ys \longrightarrow ys \neq [] \longrightarrow P (tl(ys)!j)$
 apply (induct ys)
 apply simp-all
 done

3.3.4 The Semantics is Compositional

lemma *aux-if [rule-format]*:
 $\forall xs \ s \ \text{clist}. (\text{length } \text{clist} = \text{length } xs \wedge (\forall i < \text{length } xs. (xs!i, s) \# \text{clist}!i \in \text{cptn})$
 $\wedge ((xs, s) \# ys \propto \text{map } (\lambda i. (\text{fst } i, s) \# \text{snd } i) (\text{zip } xs \ \text{clist}))$
 $\longrightarrow (xs, s) \# ys \in \text{par-cptn})$
 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  $P=\lambda j. ?H\ j \longrightarrow (?P\ j \vee ?Q\ j)$  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  $P=\lambda j. (?H\ j) \longrightarrow (snd\ (?d\ j))=(snd\ (?e\ j))$  in allE,simp)
  prefer 2
  apply(simp add:same-program-def)
  apply(erule-tac  $x=1$  and  $P=\lambda j. ?H\ j \longrightarrow (fst\ (?s\ j))=(?t\ j)$  in allE,simp)
  apply(rule nth-equalityI,simp)
  apply clarify
  apply(case-tac  $i=ia,simp,simp$ )
  apply(erule-tac  $x=ia$  and  $P=\lambda j. ?H\ j \longrightarrow ?I\ j \longrightarrow ?J\ j$  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  $P=\lambda j. ?H\ j \longrightarrow (snd\ (?d\ j))=(snd\ (?e\ j))$  in allE)
    apply(erule-tac  $x=ia$  and  $P=\lambda j. ?H\ j \longrightarrow ?I\ j \longrightarrow ?J\ j$  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= $\lambda j. ?H j \longrightarrow (snd (?d j))=(snd (?e j))$ 
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= $\lambda j. ?H j \longrightarrow (fst (?s j))=(?t j)$  in
allE,simp)
  apply(rule nth-equalityI,simp,simp)
  apply(force simp add:length-Suc-conv)
  apply(rule allI,rule impI)
  apply(erule-tac x=Suc j and P= $\lambda j. ?H j \longrightarrow (?I j \vee ?J j)$  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= $\lambda j. ?H j \longrightarrow ?I j \longrightarrow ?J j$  in allE,erule impE,assumption)
  apply(erule-tac x=l and P= $\lambda j. ?H j \longrightarrow ?I j \longrightarrow ?J j$  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= $\lambda j. (?H j \longrightarrow (snd (?d j))=(snd (?e j)))$  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= $\lambda j. ?H j \longrightarrow (snd (?d j))=(snd (?e j))$  in allE)
  apply(erule-tac x=ia and P= $\lambda j. ?H j \longrightarrow ?I j \longrightarrow ?J j$  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  $\longrightarrow$  (length (?s j) = ?t) in allE,force)
    apply simp
  apply(erule-tac P= $\lambda j$ . ?H j  $\longrightarrow$  ?I j  $\longrightarrow$  ?J 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  $\longrightarrow$  (length (?s j) = ?t) in allE,force)
  apply(rule nth-tl-if)
    apply(erule-tac x=l and P= $\lambda j$ . ?H j  $\longrightarrow$  (length (?s j) = ?t) in allE,force)
    apply(erule-tac x=l in allE, erule impE, assumption,
      erule-tac x=1 and P= $\lambda j$ . ?H j  $\longrightarrow$  (snd (?d j))=(snd (?e j)) in allE)
    apply(erule-tac x=l and P= $\lambda j$ . ?H j  $\longrightarrow$  ?I j  $\longrightarrow$  ?J 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  $\longrightarrow$  (length (?s j) = ?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 $\in$ etran 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  $\longrightarrow$  (snd (?d j))=(snd (?e j)) in allE)
  apply(erule-tac x=ia and P= $\lambda j$ . ?H j  $\longrightarrow$  ?I j  $\longrightarrow$  ?J 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  $\longrightarrow$  (length (?s j) = ?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  $\longrightarrow$  (length (?s j) = ?t) in allE,force)
apply force
apply(erule-tac x=ia and P= $\lambda j$ . ?H j  $\longrightarrow$  (length (?s j) = ?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  $x=\text{map } tl \text{ clist}$  in exI,simp)
apply(rule conjI)
  apply clarify
apply(erule-tac  $x=i$  and  $P=\lambda j. ?H j \longrightarrow (?I ?s j) \in \text{cptn}$  in allE,simp)
apply(erule cptn.cases)
  apply(simp add:same-length-def)
  apply(erule-tac  $x=i$  and  $P=\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
apply(simp add:same-state-def)
apply(erule-tac  $x=i$  in allE, erule impE, assumption,
  erule-tac  $x=1$  and  $P=\lambda j. ?H j \longrightarrow (\text{snd } (?d j))=(\text{snd } (?e j))$  in allE,simp)
apply(erule-tac  $x=i$  and  $P=\lambda j. ?H j \longrightarrow ?J j \in \text{etran}$  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  $x=\text{Suc}(\text{Suc } \text{nat})$  and  $P=\lambda j. ?H j \longrightarrow (\text{snd } (?d j))=(\text{snd } (?e j))$ 
in allE,simp)
apply(rule tl-zero)
  apply(simp)
  apply force
apply(erule-tac  $x=i$  and  $P=\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  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  $P=\lambda j. ?H j \longrightarrow (\text{fst } (?s j))=(?t j)$  in
allE,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  $P=\lambda j. ?H j \longrightarrow (?I j \vee ?J j)$  in allE,simp)
apply(erule disjE)
apply clarify
apply(rule-tac  $x=i$  in exI,simp)
apply(rule conjI)
  apply(erule-tac  $x=i$  and  $P=\lambda i. ?H i \longrightarrow ?J i \in \text{etran}$  in allE, erule impE,
assumption)
  apply(erule etranE,simp)
  apply(erule-tac  $x=i$  in allE, erule impE, assumption,
  erule-tac  $x=1$  and  $P=\lambda j. (?H j) \longrightarrow (\text{snd } (?d j))=(\text{snd } (?e j))$  in allE,simp)
apply(rule nth-tl-if)

```

```

    apply(erule-tac x=i and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply simp
  apply(erule tl-zero,force)
  apply(erule-tac x=i and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply clarify
  apply(erule-tac x=l and P= $\lambda i$ .  $?H i \longrightarrow ?J i \in \text{etran}$  in allE, erule impE,
assumption)
  apply(erule etranE,simp)
  apply(erule-tac x=l in allE, erule impE, assumption,
    erule-tac x=1 and P= $\lambda j$ .  $(?H j) \longrightarrow (\text{snd } (?d j)) = (\text{snd } (?e j))$  in allE,simp)
  apply(rule nth-tl-if)
  apply(erule-tac x=l and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply simp
  apply(rule tl-zero,force)
  apply force
  apply(erule-tac x=l and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply(rule disjI2)
  apply simp
  apply clarify
  apply(case-tac j,simp)
  apply(rule tl-zero)
  apply(erule-tac x=i and P= $\lambda i$ .  $?H i \longrightarrow ?J i \in \text{etran}$  in allE, erule impE,
assumption)
  apply(erule-tac x=i and P= $\lambda i$ .  $?H i \longrightarrow ?J i \in \text{etran}$  in allE, erule impE,
assumption)
  apply(force elim:etranE intro:Env)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply simp
  apply(rule tl-zero)
  apply(rule tl-zero,force)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
  apply force
  apply(erule-tac x=i and P= $\lambda j$ .  $?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE,force)
done

```

lemma less-Suc-0 [iff]: $(n < \text{Suc } 0) = (n = 0)$
by auto

lemma aux-onlyif [rule-format]: $\forall xs s. (xs, s) \# ys \in \text{par-cptn} \longrightarrow$
 $(\exists \text{clist}. (\text{length clist} = \text{length } xs) \wedge$
 $(xs, s) \# ys \propto \text{map } (\lambda i. (\text{fst } i, s) \# (\text{snd } i)) (\text{zip } xs \text{ clist}) \wedge$
 $(\forall i < \text{length } xs. (xs!i, s) \# (\text{clist}!i) \in \text{cptn}))$
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 (λj. (P!j, t)#(clist!j)) [0..<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=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=(map (λj. (Ps!j, ta)#(clist!j)) [0..<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=λj. ?H j → (?I j ∈ cptn) 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=nat and P= $\lambda j. ?H j \longrightarrow (fst (?a j))=((?b j))$  in allE)
  apply(case-tac 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= $\lambda j. ?H j \longrightarrow (?P j \vee ?Q j)$  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= $\lambda j. ?H j \longrightarrow (?P j) \in etran$  in allE, erule impE,

```

```

assumption)
apply(case-tac i=ia,simp,simp)
done

lemma one-iff-aux:  $xs \neq [] \implies (\forall ys. ((xs, s) \# ys \in \text{par-cptn}) =$ 
  ( $\exists \text{clist. length clist} = \text{length } xs \wedge$ 
  ( $(xs, s) \# ys \propto \text{map } (\lambda i. (\text{fst } i, s) \# (\text{snd } i)) (\text{zip } xs \text{ clist})) \wedge$ 
  ( $\forall i < \text{length } xs. (xs!i, s) \# (\text{clist}!i) \in \text{cptn})) =$ 
  ( $\text{par-cp } (xs) \text{ } s = \{c. \exists \text{clist. } (\text{length } \text{clist}) = (\text{length } xs) \wedge$ 
  ( $\forall i < \text{length } \text{clist. } (\text{clist}!i) \in \text{cp}(xs!i) \text{ } s) \wedge c \propto \text{clist}\})$ )
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(erule-tac x=list in allE)
apply clarify
apply simp
apply(rule-tac x=map ( $\lambda i. (\text{fst } i, s) \# \text{snd } i$ ) (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= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  in all-dupE)
apply(subgoal-tac a = xs)
apply(subgoal-tac b = s,simp)
prefer 3
apply(erule-tac x=0 and P= $\lambda j. ?H j \longrightarrow (\text{fst } (?s j)) = ((?t j))$  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= $\lambda j. ?H j \longrightarrow (\forall i. ?T i \longrightarrow (\text{snd } (?d j i)) = (\text{snd } (?e j i)))$  in allE,simp)
apply(erule-tac x=0 and P= $\lambda j. ?H j \longrightarrow (\text{snd } (?d j)) = (\text{snd } (?e j))$  in allE,simp)
apply(erule-tac x=list in allE)
apply(rule-tac x=map tl clist in exI,simp)
apply(rule conjI)

```



```

apply clarify
apply(case-tac j,simp)
  apply(erule-tac x=i in allE, erule impE, assumption,
    erule-tac x=0 and P= $\lambda j. ?H j \longrightarrow (\text{snd } (?d j))=(\text{snd } (?e j))$  in allE,simp)
apply(erule-tac x=i in allE, erule impE, assumption,
  erule-tac x=Suc nat and P= $\lambda j. ?H j \longrightarrow (\text{snd } (?d j))=(\text{snd } (?e j))$  in
allE)
apply(erule-tac x=i and P= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  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= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE)
apply(case-tac clist!i,simp,simp)
apply(thin-tac ?H = ( $\exists i. ?J i$ ))
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,simp)
apply(rule conjI)
apply(erule-tac x=i in allE)
apply(simp add:cp-def)
apply(erule-tac x=i and P= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  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= $\lambda j. ?H j \longrightarrow ?I j \longrightarrow ?J j$  in allE)
apply clarify
apply(simp add:cp-def)
apply(erule-tac x=l and P= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE)
apply(case-tac clist!l,simp,simp)
apply simp
apply(rule conjI)
apply(erule-tac x=i and P= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  in allE)
apply(case-tac clist!i,simp,simp)
apply clarify
apply(erule-tac x=l and P= $\lambda j. ?H j \longrightarrow ?I j \longrightarrow ?J j$  in allE)
apply(erule-tac x=l and P= $\lambda j. ?H j \longrightarrow (\text{length } (?s j) = ?t)$  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  $\longrightarrow$  (length (?s j) = ?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) $\in$ etran 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  $\longrightarrow$  (length (?s j) = ?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  $\longrightarrow$  (length (?s j) = ?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  $\longrightarrow$  (snd (?d j))=(snd (?e j)) in allE)
apply(case-tac j,simp)
apply(erule-tac x=i and P= $\lambda j$ . ?H j  $\longrightarrow$  (length (?s j) = ?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  $\longrightarrow$  (length (?s j) = ?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  $P=\lambda j. ?H\ j \longrightarrow (length\ (?s\ j) = ?t)$  in  $allE$ )
  apply(case-tac  $clist!i, simp, simp$ )
  apply(erule-tac  $x=i$  and  $P=\lambda j. ?H\ j \longrightarrow (length\ (?s\ j) = ?t)$  in  $allE$ )
  apply(case-tac  $clist!i, simp, simp$ )
apply clarify
apply(erule-tac  $x=l$  and  $P=\lambda j. ?H\ j \longrightarrow ?I\ j \longrightarrow ?J\ j$  in  $allE$ )
apply(erule-tac  $x=l$  and  $P=\lambda j. ?H\ j \longrightarrow (length\ (?s\ j) = ?t)$  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  $P=\lambda j. ?H\ j \longrightarrow (length\ (?s\ j) = ?t)$  in  $allE, force$ )
  apply force
apply(erule-tac  $x=i$  and  $P=\lambda j. ?H\ j \longrightarrow (length\ (?s\ j) = ?t)$  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  $x=map\ (\lambda i. (fst\ i, s) \# snd\ i)\ (zip\ xs\ clist)$  in  $allE$ )
apply simp
apply clarify
apply(simp add:cp-def)
done

theorem one: xs $\neq$ []  $\implies$ 
  par-cp  $xs\ s = \{c. \exists\ clist. (length\ clist)=(length\ xs) \wedge$ 
     $(\forall\ i<length\ clist. (clist!i) \in cp(xs!i)\ s) \wedge c \propto clist\}$ 
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] le-sup-iff [simp del]
declare Cons-eq-map-conv [iff]

```

definition *stable* :: '*a* set \Rightarrow ('*a* \times '*a*) set \Rightarrow bool **where**
stable $\equiv \lambda f g. (\forall x y. x \in f \longrightarrow (x, y) \in g \longrightarrow y \in f)$

inductive

rghoare :: [*'a com*, '*a* set, ('*a* \times '*a*) set, ('*a* \times '*a*) set, '*a* set] \Rightarrow bool
 $(\vdash - \text{sat } [-, -, -, -] [60, 0, 0, 0, 0] \ 45)$

where

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

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

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

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

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

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

$$\Longrightarrow \vdash P \text{ sat } [pre, rely, guar, post]$$

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**
 $Post \ x \equiv snd(snd(snd(snd \ x)))$

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

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

inductive

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

where

$Parallel:$

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

3.5 Soundness

Some previous lemmas

lemma $tl\text{-}of\text{-}assum\text{-}in\text{-}assum:$

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

apply $(simp \text{ add:assum-def})$

apply $clarify$

apply $(rule \text{ conjI})$

apply $(erule\text{-}tac \ x=0 \text{ in } allE)$

apply $(simp \ (no\text{-}asm\text{-}use) \text{ only:stable-def})$

apply $(erule \ allE, erule \ allE, erule \ impE, assumption, erule \ mp)$

apply $(simp \ \text{add:Env})$

```

apply clarify
apply(erule-tac  $x = \text{Suc } i$  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  $i, \text{simp}+$ )
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  $i, \text{simp}+$ )
done

```

```

lemma takecptn-is-cptn [rule-format, elim!]:
   $\forall j. c \in \text{cptn} \longrightarrow \text{take } (\text{Suc } j) \ c \in \text{cptn}$ 
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  $j, \text{simp}+$ )
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:  $\llbracket x \# xs \in \text{cptn}; xs \neq [] \rrbracket \implies xs \in \text{cptn}$ 
apply(subgoal-tac 1 < length (x # xs))
apply(drule dropcptn-is-cptn,simp+)
done

```

```

lemma not-ctran-None [rule-format]:
   $\forall s. (None, s) \# xs \in \text{cptn} \longrightarrow (\forall i < \text{length } xs. ((None, s) \# xs)!i -e\rightarrow xs!i)$ 
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]:  $[] \notin \text{cptn}$ 
apply(force elim:cptn.cases)
done

```

```

lemma etran-or-ctran [rule-format]:
   $\forall m \ i. x \in \text{cptn} \longrightarrow m \leq \text{length } x$ 
   $\longrightarrow (\forall i. \text{Suc } i < m \longrightarrow \neg x!i -c\rightarrow x!\text{Suc } i) \longrightarrow \text{Suc } i < m$ 
   $\longrightarrow x!i -e\rightarrow x!\text{Suc } i$ 
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 ( $\forall i. \text{Suc } i < \text{nat} \longrightarrow ((P, t) \# xs) ! i, xs ! i) \notin \text{ctran}$ ))
apply force
apply clarify
apply(erule-tac  $x = \text{Suc } ia$  in allE,simp)
apply(erule-tac  $x = 0$  and  $P = \lambda j. ?H j \longrightarrow (?J j) \notin \text{ctran}$  in allE,simp)
done

```

```

lemma etran-or-ctran2 [rule-format]:
   $\forall i. \text{Suc } i < \text{length } x \longrightarrow x \in \text{cptn} \longrightarrow (x!i -c \longrightarrow x!\text{Suc } i \longrightarrow \neg x!i -e \longrightarrow x!\text{Suc } i)$ 
   $\vee (x!i -e \longrightarrow x!\text{Suc } i \longrightarrow \neg x!i -c \longrightarrow x!\text{Suc } i)$ 
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:
   $\llbracket x \in \text{cptn}; \text{Suc } i < \text{length } x; x!i -c \longrightarrow x!\text{Suc } i \rrbracket \Longrightarrow \neg x!i -e \longrightarrow x!\text{Suc } i$ 
by(drule etran-or-ctran2,simp-all)

```

```

lemma etran-or-ctran2-disjI2:
   $\llbracket x \in \text{cptn}; \text{Suc } i < \text{length } x; x!i -e \longrightarrow x!\text{Suc } i \rrbracket \Longrightarrow \neg x!i -c \longrightarrow x!\text{Suc } i$ 
by(drule etran-or-ctran2,simp-all)

```

```

lemma not-ctran-None2 [rule-format]:
   $\llbracket (\text{None}, s) \# xs \in \text{cptn}; i < \text{length } xs \rrbracket \Longrightarrow \neg ((\text{None}, s) \# xs) ! i -c \longrightarrow xs ! i$ 
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]:
   $\forall j k. x \in \text{cptn} \longrightarrow \text{stable } p \text{ rely} \longrightarrow j \leq k \longrightarrow k < \text{length } x \longrightarrow \text{snd}(x!j) \in p \longrightarrow$ 

```


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

apply *clarify*
apply(*erule-tac* $x = \text{Suc } i$ **and** $P = \lambda j. (?H\ j) \longrightarrow (?J\ j) \longrightarrow (?T\ j) \in \text{rely}$ **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*]:
 $\forall s\ i. x \in \text{cptn} \longrightarrow x!0 = (\text{Some } (\text{Basic } f), s) \longrightarrow$
 $\text{Suc } i < \text{length } x \longrightarrow x!i -c \rightarrow x!\text{Suc } i \longrightarrow$
 $(\forall j. \text{Suc } j < \text{length } x \longrightarrow i \neq j \longrightarrow x!j -e \rightarrow x!\text{Suc } j)$
apply(*induct* x, simp)
apply *simp*
apply *clarify*
apply(*erule* $\text{cptn.cases}, \text{simp}$)
apply(*case-tac* $i, \text{simp}+$)
apply *clarify*
apply(*case-tac* j, simp)
apply(*rule* *Env*)
apply *simp*
apply *clarify*
apply *simp*
apply(*case-tac* i)
apply(*case-tac* $j, \text{simp}, \text{simp}$)
apply(*erule* $\text{ctran.cases}, \text{simp-all}$)
apply(*force elim: not-ctran-None*)
apply(*ind-cases* $((\text{Some } (\text{Basic } f), sa), Q, t) \in \text{ctran}$ **for** $sa\ Q\ t$)
apply *simp*
apply(*drule-tac* $i = \text{nat}$ **in** $\text{not-ctran-None}, \text{simp}$)
apply(*erule* $\text{etranE}, \text{simp}$)
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* $\text{cptn.cases}, \text{simp}$)
apply(*case-tac* $i, \text{simp}, \text{simp}$)
apply(*erule-tac* $x = \text{nat}$ **in** allE, simp)
apply *clarify*
apply(*rule-tac* $x = \text{Suc } j$ **in** $\text{exI}, \text{simp}, \text{simp}$)
apply *clarify*
apply(*case-tac* $i, \text{simp}, \text{simp}$)
apply(*rule-tac* $x = 0$ **in** exI, simp)
done

lemma *Basic-sound*:

```

     $\llbracket pre \subseteq \{s. f \mid s \in post\}; \{(s, t). s \in pre \wedge t = f s\} \subseteq guar;$ 
     $stable\ pre\ rely; stable\ post\ rely \rrbracket$ 
 $\implies \models Basic\ f\ sat\ [pre, rely, guar, post]$ 
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)
    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  $\forall j. ?H\ j$ )
apply(force elim:ctran.cases)
apply clarify
apply(simp add:cp-def)
apply clarify
apply(frule-tac i=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=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=Some (Basic f) in sym,simp)
apply(case-tac x!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=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 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]:
 $\forall s\ i. x \in \text{cptn} \longrightarrow x ! 0 = (\text{Some } (\text{Await } b\ c), s) \longrightarrow$
 $\text{Suc } i < \text{length } x \longrightarrow x!i \text{ --c--> } x!\text{Suc } i \longrightarrow$
 $(\forall j. \text{Suc } j < \text{length } x \longrightarrow i \neq j \longrightarrow x!j \text{ --e--> } x!\text{Suc } j)$
apply (induct $x, \text{simp}+$)
apply clarify
apply (erule $\text{cptn.cases}, \text{simp}$)
apply (case-tac $i, \text{simp}+$)
apply clarify
apply (case-tac j, simp)
apply (rule *Env*)
apply *simp*
apply clarify
apply *simp*
apply (case-tac i)
apply (case-tac $j, \text{simp}, \text{simp}$)
apply (erule $\text{ctran.cases}, \text{simp-all}$)
apply (force elim: *not-ctran-None*)
apply (ind-cases $((\text{Some } (\text{Await } b\ c), sa), Q, t) \in \text{ctran} \text{ for } sa\ Q\ t, \text{simp})$)
apply (drule-tac $i = \text{nat}$ in *not-ctran-None, simp*)
apply (erule *etranE*, *simp*)
done

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

lemma *Star-imp-cptn*:
 $(P, s) \text{ --c*--> } (R, t) \implies \exists l \in \text{cp } P\ s. (\text{last } l) = (R, t)$
 $\wedge (\forall i. \text{Suc } i < \text{length } l \longrightarrow l!i \text{ --c--> } l!\text{Suc } i)$
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:
  
$$\begin{aligned}
& \llbracket \text{stable pre rely; stable post rely;} \\
& \forall V. \vdash P \text{ sat } [\text{pre} \cap b \cap \{s. s = V\}, \{(s, t). s = t\}, \\
& \quad \text{UNIV}, \{s. (V, s) \in \text{guar}\} \cap \text{post}] \wedge \\
& \models P \text{ sat } [\text{pre} \cap b \cap \{s. s = V\}, \{(s, t). s = t\}, \\
& \quad \text{UNIV}, \{s. (V, s) \in \text{guar}\} \cap \text{post}] \rrbracket \\
& \implies \models \text{Await } b \text{ } P \text{ sat } [\text{pre}, \text{rely}, \text{guar}, \text{post}]
\end{aligned}$$

  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 $\in$  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= $\lambda i. ?H \ i \longrightarrow (?J \ i, ?I \ i) \in \text{ctran}$  in allE,simp)
  apply(erule etranE,simp)

```

```

  apply simp
  apply clarify
  apply(simp add:cp-def)
  apply clarify
  apply(frul-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 add:cptn-not-empty)+)
  apply clarify
  apply(simp add:assum-def)
  apply clarify
  apply(frul-tac j=0 and k=j and p=pre in stability,simp-all)
    apply(erul-tac x=i in allE,simp)
    apply(erul-tac i=j in unique-ctran-Await,force,simp-all)
  apply(case-tac x!j)
  apply clarify
  apply simp
  apply(drul-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(drul Star-imp-cptn)
  apply clarify
  apply(erul-tac x=sa in allE)
  apply clarify
  apply(erul-tac x=sa in allE)
  apply(drul-tac c=l in subsetD)
    apply (simp add:cp-def)
    apply clarify
    apply(erul-tac x=i and P= $\lambda i. ?H i \longrightarrow (?J i, ?I i) \in \text{ctran}$  in allE,simp)
    apply(erul etranE,simp)
  apply simp
  apply clarify
  apply(frul-tac j=Suc j and k=length x - 1 and p=post in stability,simp-all)
    apply(case-tac x,simp+)
    apply(erul-tac x=i in allE)
  apply(erul-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*:

```

  [[ stable pre rely;  $\models P1 \text{ sat } [pre \cap b, \text{ rely}, \text{ guar}, \text{ post}]$ ;
     $\models P2 \text{ sat } [pre \cap \neg b, \text{ rely}, \text{ guar}, \text{ post}]$ ;  $\forall s. (s,s) \in \text{guar}$  ]]
   $\implies \models (\text{Cond } b \ P1 \ P2) \text{ sat } [pre, \text{ rely}, \text{ guar}, \text{ post}]$ 
  apply(unfold com-validity-def)

```

```

apply clarify
apply (simp add:cp-def comm-def)
apply (case-tac  $\exists i. \text{Suc } i < \text{length } x \wedge x!i - c \rightarrow x!\text{Suc } i$ )
prefer 2
apply simp
apply clarify
apply (frule-tac  $j=0$  and  $k=\text{length } x - 1$  and  $p=\text{pre}$  in stability,simp+)
  apply (case-tac  $x,\text{simp+}$ )
  apply (simp add:assum-def)
  apply (simp add:assum-def)
  apply (erule-tac  $m=\text{length } x$  in etran-or-ctran,simp+)
apply (case-tac  $x, (\text{simp add:last-length})+$ )
apply (erule exE)
apply (drule-tac  $n=i$  and  $P=\lambda i. ?H\ i \wedge (?J\ i, ?I\ i) \in \text{ctran}$  in Ex-first-occurrence)
apply clarify
apply (simp add:assum-def)
apply (frule-tac  $j=0$  and  $k=m$  and  $p=\text{pre}$  in stability,simp+)
  apply (erule-tac  $m=\text{Suc } m$  in etran-or-ctran,simp+)
apply (erule ctran.cases,simp-all)
apply (erule-tac  $x=\text{sa}$  in allE)
apply (drule-tac  $c=\text{drop } (\text{Suc } m)\ x$  in subsetD)
  apply simp
  apply clarify
apply simp
apply clarify
apply (case-tac  $i \leq m$ )
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  $x=i - (\text{Suc } m)$  and  $P=\lambda j. ?H\ j \longrightarrow ?J\ j \longrightarrow (?I\ j) \in \text{guar}$  in allE)
apply (subgoal-tac  $(\text{Suc } m) + (i - \text{Suc } m) \leq \text{length } x$ )
apply (subgoal-tac  $(\text{Suc } m) + \text{Suc } (i - \text{Suc } m) \leq \text{length } x$ )
apply (rotate-tac -2)
apply simp
apply arith
apply arith
apply (case-tac  $\text{length } (\text{drop } (\text{Suc } m)\ x),\text{simp}$ )
apply (erule-tac  $x=\text{sa}$  in allE)
back
apply (drule-tac  $c=\text{drop } (\text{Suc } m)\ x$  in subsetD,simp)
  apply clarify
apply simp
apply clarify
apply (case-tac  $i \leq m$ )
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= $\lambda j. ?H\ j \longrightarrow ?J\ j \longrightarrow (?I\ j) \in guar$  in
allE)
  apply(subgoal-tac (Suc m)+(i - Suc m)  $\leq$  length x)
  apply(subgoal-tac (Suc m)+Suc (i - Suc m)  $\leq$  length x)
  apply(rotate-tac -2)
  apply simp
  apply arith
  apply arith
done

```

Soundness of the Sequential rule

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

```

lemma last-lift-not-None: fst ((lift Q) ((x#xs)!(length xs)))  $\neq$  None
  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

```

declare map-eq-Cons-conv [simp del] Cons-eq-map-conv [simp del]

lemma Seq-sound1 [rule-format]:

```

  x  $\in$  cptn-mod  $\implies \forall s\ P. x \neq 0 \implies$  (Some (Seq P Q), s)  $\longrightarrow$ 
  ( $\forall i < \text{length } x. \text{fst}(x[i]) \neq \text{Some } Q$ )  $\longrightarrow$ 
  ( $\exists xs \in \text{cp } (\text{Some } P)\ s. x = \text{map } (\text{lift } Q)\ xs$ )
  apply(erule cptn-mod.induct)
  apply(unfold cp-def)
  apply safe
  apply simp-all
  apply(simp add:lift-def)
  apply(rule-tac x=[(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=(Some Pa, sa) # (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=Suc i in allE, simp)
  apply(ind-cases ((Some (Seq Pa Q), sa), None, t)  $\in$  ctran for Pa sa t)
  apply(rule-tac x=(Some P, sa) # xs in exI, simp add:cptn-iff-cptn-mod lift-def)
  apply(erule-tac x=length xs in allE, simp)
  apply(simp only:Cons-lift-append)
  apply(subgoal-tac length xs < length ((Some P, sa) # xs))

```



```

apply(simp only :nth-append length-map last-length nth-map)
apply(case-tac last((Some P, sa) # xs))
apply(simp add:lift-def)
apply simp
done
declare map-eq-Cons-conv [simp del] Cons-eq-map-conv [simp del]

lemma Seq-sound2 [rule-format]:
   $x \in \text{cptn} \implies \forall s P i. x!0 = (\text{Some } (\text{Seq } P \ Q), s) \longrightarrow i < \text{length } x$ 
   $\longrightarrow \text{fst}(x!i) = \text{Some } Q \longrightarrow$ 
   $(\forall j < i. \text{fst}(x!j) \neq (\text{Some } Q)) \longrightarrow$ 
   $(\exists xs \ ys. xs \in \text{cp } (\text{Some } P) \ s \wedge \text{length } xs = \text{Suc } i$ 
   $\wedge \ ys \in \text{cp } (\text{Some } Q) \ (\text{snd}(xs \ !i)) \wedge x = (\text{map } (\text{lift } Q) \ xs) @ \text{tl } ys)$ 
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 ( $\forall j < \text{nat}. \text{fst } (((\text{Some } (\text{Seq } Pa \ Q), t) \# xs) \ !j) \neq \text{Some } Q$ ), clarify)
prefer 2
apply force
apply(case-tac xsa, simp, simp)
apply(rule-tac  $x = (\text{Some } Pa, sa) \# (\text{Some } Pa, t) \# \text{list in } exI, \text{simp}$ )
apply(rule conjI, erule CptnEnv)
apply(simp (no-asm-use) add:lift-def)
apply(rule-tac  $x = ys$  in  $exI, \text{simp}$ )
apply(ind-cases ((Some (Seq Pa Q), sa), t)  $\in$  ctran for Pa sa t)
apply simp
apply(rule-tac  $x = (\text{Some } Pa, sa) \# [(None, ta)]$  in  $exI, \text{simp}$ )
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  $x = (\text{Some } Q, ta) \# xs$  in  $exI, \text{simp}$  add:lift-def)
apply(case-tac nat, simp+)
apply(force)
apply(case-tac i, simp+)
apply(case-tac nat, simp+)
apply(erule-tac  $x = \text{Suc } nata$  in allE, simp)
apply clarify
apply(subgoal-tac ( $\forall j < \text{Suc } nata. \text{fst } (((\text{Some } (\text{Seq } P2 \ Q), ta) \# xs) \ !j) \neq \text{Some } Q$ ), clarify)
prefer 2
apply clarify
apply force
apply(rule-tac  $x = (\text{Some } Pa, sa) \# (\text{Some } P2, ta) \# (\text{tl } xsa)$  in  $exI, \text{simp}$ )

```

```

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:map.simps)
  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)))  $\neq$  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:
   $\llbracket \models P \text{ sat } [pre, rely, guar, mid]; \models Q \text{ sat } [mid, rely, guar, post] \rrbracket$ 
   $\implies \models Seq P Q \text{ sat } [pre, rely, guar, post]$ 
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(frule-tac Seq-sound1,force)
  apply force
  apply clarify
  apply(erule-tac x=s in allE,simp)
  apply(drule-tac c=xs in subsetD,simp add:cp-def cptn-iff-cptn-mod)
  apply(simp add:assum-def)
  apply clarify
  apply(erule-tac  $P = \lambda j. ?H j \longrightarrow ?J j \longrightarrow ?I j$  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  $P = \lambda j. ?H\ j \longrightarrow ?J\ j \longrightarrow ?I\ j$  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:map.simps)
apply(subgoal-tac (map (lift Q) (( $a, b$ )  $\#$  list)) $\neq []$ )
  apply(drule last-conv-nth)
  apply (simp del:map.simps)
  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  $x=\text{snd}(xs!m)$  in allE)
apply(drule-tac  $c=ys$  in subsetD, simp add:cp-def assum-def)
  apply(case-tac  $xs \neq []$ )
  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 length-append)
done

```

```

lemma assum-after-body:
  
$$\begin{aligned} & \llbracket \models P \text{ sat } [pre \cap b, \text{ rely}, \text{ guar}, pre]; \\ & (Some\ P, s) \# xs \in \text{cptn-mod}; \text{fst}(\text{last}((Some\ P, s) \# xs)) = \text{None}; s \in b; \\ & (Some\ (While\ b\ P), s) \# (Some\ (Seq\ P\ (While\ b\ P)), s) \# \\ & \quad \text{map}(\text{lift}\ (While\ b\ P))\ xs @ ys \in \text{assum}\ (pre, \text{rely}) \rrbracket \\ & \implies (Some\ (While\ b\ P), \text{snd}(\text{last}((Some\ P, s) \# xs))) \# ys \in \text{assum}\ (pre, \text{rely}) \end{aligned}$$

  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:map.simps)
  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 del:map.simps)
  apply(simp add:last-length)
  apply(erule mp)
  apply(case-tac last xs)
  apply(simp add:lift-def)
apply(simp add:Cons-lift-append nth-append snd-lift del:map.simps)
done

lemma While-sound-aux [rule-format]:
  
$$\llbracket pre \cap - b \subseteq post; \models P \text{ sat } [pre \cap b, rely, guar, pre]; \forall s. (s, s) \in guar; \text{stable } pre \text{ rely}; \text{stable } post \text{ rely}; x \in \text{cptn-mod} \rrbracket$$


$$\implies \forall s \text{ xs. } x = (\text{Some}(\text{While } b \text{ } P), s) \# xs \longrightarrow x \in \text{assum}(pre, rely) \longrightarrow x \in \text{comm}(guar, post)$$

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(thin-tac P = While b P  $\longrightarrow$  ?Q)
  apply(rule ctran-in-comm,simp)
  apply(simp add:Cons-lift del:map.simps)
  apply(simp add:comm-def del:map.simps)
  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)  $-c \longrightarrow t$  for ba t)
  apply simp
  apply simp
  apply(simp add:snd-lift del:map.simps)
  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:map.simps)
  apply clarify
  apply(erule-tac x=Suc ia in allE,simp add:snd-lift del:map.simps)
  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:map.simps)
  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)) $\neq []$ )
  apply(drule last-conv-nth)
  apply (simp del:map.simps)
  apply(simp only:last-lift-not-None)
  apply simp
— WhileMore
  apply(thin-tac P = While b P  $\longrightarrow$  ?Q)

```

```

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:map.simps 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:map.simps 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:map.simps last.simps)
apply(thin-tac ∀ i. i < length ys → ?P i)
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:map.simps last.simps)
apply clarify
apply(erule-tac x=Suc ia in allE,simp add:nth-append snd-lift del:map.simps
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:map.simps)
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 ≥ length xs
apply(subgoal-tac i-length xs <length ys)
prefer 2
apply arith

```



```

apply(erule-tac  $x=i-length\ xs$  in allE,clarify)
apply(case-tac  $i=length\ xs$ )
  apply (simp add:nth-append snd-lift del:map.simps 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)
—  $i > length\ xs$ 
apply(case-tac  $i=length\ xs$ )
  apply arith
apply(simp add:nth-append del:map.simps last.simps)
apply(rotate-tac -3)
apply(subgoal-tac  $i- Suc\ (length\ xs)=nat$ )
  prefer 2
  apply arith
apply simp
— last=None
apply clarify
apply(case-tac ys)
  apply(simp add:Cons-lift del:map.simps last.simps)
  apply(subgoal-tac (map (lift (While b P)) ((Some P, sa) # xs))≠[])
  apply(drule last-conv-nth)
  apply (simp del:map.simps)
  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:map.simps last.simps)
  apply(simp add:nth-append del:last.simps)
  apply(subgoal-tac ((Some (While b P), snd (last ((Some P, sa) # xs))) # a #
list)≠[])
  apply(drule last-conv-nth)
  apply (simp del:map.simps last.simps)
apply simp
apply simp
done

```

lemma *While-sound*:

$\llbracket stable\ pre\ rely; pre \cap -\ b \subseteq post; stable\ post\ rely;$
 $\models P\ sat\ [pre \cap b, rely, guar, pre]; \forall s. (s,s) \in guar \rrbracket$
 $\implies \models While\ b\ P\ sat\ [pre, rely, guar, post]$

```

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 clarify
apply(case-tac i,simp+)
apply(case-tac x,simp+)
done

```

Soundness of the Rule of Consequence

```

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

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

$\implies \forall j \ i. \ i < \text{length } \text{clist} \wedge \text{Suc } j < \text{length } x \longrightarrow (\text{clist}!i!j) -c \rightarrow (\text{clist}!i!\text{Suc } j)$
 $\longrightarrow (\text{snd}(\text{clist}!i!j), \text{snd}(\text{clist}!i!\text{Suc } j)) \in \text{Guar}(xs!i)$
apply(*unfold par-cp-def*)
apply(*rule ccontr*)
— By contradiction:
apply(*simp del: Un-subset-iff*)
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= $\lambda j. \exists i. ?H \ i \ j$ 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($\text{Pre}(xs!i), \text{Rely}(xs!i)$)*)
— but this contradicts $\models \sigma$ - i sat [*pre-i, rely-i, guar-i, post-i*]
apply(*erule-tac x=i and P= $\lambda i. ?H \ i \longrightarrow \models (?J \ i) \text{ sat } [?I \ i, ?K \ i, ?M \ i, ?N \ i]$ 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= $\lambda j. ?I \ j \wedge ?J \ j \longrightarrow ?H \ j$ in allE*)
apply(*simp add:conjoin-def same-length-def*)
apply(*simp add:assum-def del: Un-subset-iff*)
apply(*rule conjI*)
apply(*erule-tac x=i and P= $\lambda j. ?H \ j \longrightarrow ?I \ j \in \text{cp } (?K \ j) (?J \ j)$ in allE*)
apply(*simp add:cp-def par-assum-def*)
apply(*drule-tac c=s in subsetD, simp*)
apply *simp*
apply *clarify*
apply(*erule-tac x=i and P= $\lambda j. ?H \ j \longrightarrow ?M \cup \text{UNION } (?S \ j) (?T \ j) \subseteq (?L \ j)$ in allE*)
apply(*simp del: Un-subset-iff*)
apply(*erule subsetD*)
apply *simp*
apply(*simp add:conjoin-def compat-label-def*)
apply *clarify*
apply(*erule-tac x=ia and P= $\lambda j. ?H \ j \longrightarrow (?P \ j) \vee ?Q \ j$ in allE, simp*)
— each etran in σ -1[0.. m] corresponds to
apply(*erule disjE*)
— a c-tran in some σ -{ ib }
apply *clarify*
apply(*case-tac i=ib, simp*)
apply(*erule etranE, simp*)
apply(*erule-tac x=ib and P= $\lambda i. ?H \ i \longrightarrow (?I \ i) \vee (?J \ i)$ in allE*)

```

apply (erule etranE)
apply(case-tac ia=m,simp)
apply simp
apply(erule-tac x=ia and P= $\lambda j. ?H j \longrightarrow (\forall i. ?P i j)$  in allE)
apply(subgoal-tac ia<m,simp)
prefer 2
apply arith
apply(erule-tac x=ib and P= $\lambda j. (?I j, ?H j) \in ctran \longrightarrow (?P i j)$  in allE,simp)
apply(simp add:same-state-def)
apply(erule-tac x=i and P= $\lambda j. (?T j) \longrightarrow (\forall i. (?H j i) \longrightarrow (snd (?d j i))=(snd (?e j i)))$  in all-dupE)
apply(erule-tac x=ib and P= $\lambda j. (?T j) \longrightarrow (\forall i. (?H j i) \longrightarrow (snd (?d j i))=(snd (?e j i)))$  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]:

```

 $\llbracket xs \neq []; \forall i < \text{length } xs. rely \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. Guar (xs ! j))$ 
 $\subseteq Rely (xs ! i);$ 
 $pre \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. Pre (xs ! i));$ 
 $\forall i < \text{length } xs.$ 
 $\models Com (xs ! i) \text{ sat } [Pre (xs ! i), Rely (xs ! i), Guar (xs ! i), Post (xs ! i)];$ 
 $\text{length } xs = \text{length } clist; x \in \text{par-cp } (ParallelCom \text{ } xs) \text{ } s; x \in \text{par-assum}(pre, rely);$ 
 $\forall i < \text{length } clist. clist ! i \in cp (Some(Com(xs!i))) \text{ } s; x \propto clist \rrbracket$ 
 $\implies \forall j i. i < \text{length } clist \wedge Suc j < \text{length } x \longrightarrow (clist ! i ! j) -e \longrightarrow (clist ! i ! Suc j)$ 
 $\longrightarrow (snd(clist ! i ! j), snd(clist ! i ! Suc j)) \in rely \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. Guar (xs ! j))$ 
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= $\lambda j. ?H j \longrightarrow (?J j \wedge (\exists i. ?P i j)) \vee ?I j$  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= $\lambda i. ?H i \longrightarrow (?I i) \vee (?J i)$  in allE,simp)
apply(erule-tac x=j and P= $\lambda j. \forall i. ?S j i \longrightarrow (?I j i, ?H j i) \in ctran \longrightarrow (?P i j)$  in allE)
apply(erule-tac x=ia and P= $\lambda j. ?S j \longrightarrow (?I j, ?H j) \in ctran \longrightarrow (?P j)$  in allE)
apply(simp add:same-state-def)
apply(erule-tac x=i and P= $\lambda j. (?T j) \longrightarrow (\forall i. (?H j i) \longrightarrow (snd (?d j i))=(snd$ 

```

$(?e\ j\ i)))$ **in** *all-dupE*)
apply(*erule-tac* $x=ia$ **and** $P=\lambda j. (?T\ j) \longrightarrow (\forall i. (?H\ j\ i) \longrightarrow (snd\ (?d\ j\ i))=(snd\ (?e\ j\ i)))$) **in** *allE,simp*)
done

lemma four:

$\llbracket xs \neq []; \forall i < length\ xs. rely \cup (\bigcup j \in \{j. j < length\ xs \wedge j \neq i\}. Guar\ (xs\ !\ j))$
 $\subseteq Rely\ (xs\ !\ i);$
 $(\bigcup j \in \{j. j < length\ xs\}. Guar\ (xs\ !\ j)) \subseteq guar;$
 $pre \subseteq (\bigcap i \in \{i. i < length\ xs\}. Pre\ (xs\ !\ i));$
 $\forall i < length\ xs.$
 $\models Com\ (xs\ !\ i)\ sat\ [Pre\ (xs\ !\ i), Rely\ (xs\ !\ i), Guar\ (xs\ !\ i), Post\ (xs\ !\ i)];$
 $x \in par\text{-}cp\ (ParallelCom\ xs)\ s; x \in par\text{-}assum\ (pre, rely); Suc\ i < length\ x;$
 $x\ !\ i \text{--}pc \rightarrow x\ !\ Suc\ i\rrbracket$
 $\implies (snd\ (x\ !\ i), snd\ (x\ !\ Suc\ i)) \in guar$
apply(*simp add: ParallelCom-def del: Un-subset-iff*)
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 del: Un-subset-iff*)
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= $\lambda j. ?H\ j \longrightarrow fst\ (?I\ j)=(?J\ j)$ in all-dupE*)
apply(*erule-tac x=Suc i and P= $\lambda j. ?H\ j \longrightarrow fst\ (?I\ j)=(?J\ j)$ in allE*)
apply(*erule par-ctranE,simp*)
apply(*erule-tac x=i and P= $\lambda j. \forall i. ?S\ j\ i \longrightarrow (?I\ j\ i, ?H\ j\ i) \in ctran \longrightarrow (?P\ i\ j)$ in allE*)
apply(*erule-tac x=ia and P= $\lambda j. ?S\ j \longrightarrow (?I\ j, ?H\ j) \in ctran \longrightarrow (?P\ j)$ in allE*)
apply(*rule-tac x=ia in exI*)
apply(*simp add:same-state-def*)
apply(*erule-tac x=ia and P= $\lambda j. (?T\ j) \longrightarrow (\forall i. (?H\ j\ i) \longrightarrow (snd\ (?d\ j\ i))=(snd\ (?e\ j\ i)))$ in all-dupE,simp*)
apply(*erule-tac x=ia and P= $\lambda j. (?T\ j) \longrightarrow (\forall i. (?H\ j\ i) \longrightarrow (snd\ (?d\ j\ i))=(snd\ (?e\ j\ i)))$ in allE,simp*)
apply(*erule-tac x=i and P= $\lambda j. ?H\ j \longrightarrow (snd\ (?d\ j))=(snd\ (?e\ j))$ in all-dupE*)
apply(*erule-tac x=i and P= $\lambda j. ?H\ j \longrightarrow (snd\ (?d\ j))=(snd\ (?e\ j))$ in all-dupE,simp*)

```

apply(erule-tac  $x = \text{Suc } i$  and  $P = \lambda j. ?H j \longrightarrow (\text{snd } (?d j)) = (\text{snd } (?e j))$  in
  allE, simp)
apply(erule mp)
apply(subgoal-tac  $r = \text{fst}(clist ! ia ! \text{Suc } i)$ , simp)
apply(drule-tac  $i = ia$  in list-eq-if)
back
apply simp-all
done

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

lemma five:
   $\llbracket xs \neq []; \forall i < \text{length } xs. \text{rely} \cup (\bigcup j \in \{j. j < \text{length } xs \wedge j \neq i\}. \text{Guar } (xs ! j))$ 
     $\subseteq \text{Rely } (xs ! i);$ 
   $\text{pre} \subseteq (\bigcap i \in \{i. i < \text{length } xs\}. \text{Pre } (xs ! i));$ 
   $(\bigcap i \in \{i. i < \text{length } xs\}. \text{Post } (xs ! i)) \subseteq \text{post};$ 
   $\forall i < \text{length } xs.$ 
     $\models \text{Com } (xs ! i) \text{ sat } [\text{Pre } (xs ! i), \text{Rely } (xs ! i), \text{Guar } (xs ! i), \text{Post } (xs ! i)];$ 
     $x \in \text{par-cp } (\text{ParallelCom } xs) s; x \in \text{par-assum } (\text{pre}, \text{rely});$ 
     $\text{All-None } (\text{fst } (\text{last } x)) \rrbracket \implies \text{snd } (\text{last } x) \in \text{post}$ 
apply(simp add: ParallelCom-def del: Un-subset-iff)
apply(subgoal-tac (map (Some  $\circ$  fst) xs)  $\neq []$ )
prefer 2
apply simp
apply(erule rev-subsetD)
apply(erule one [THEN equalityD1])
apply(erule subsetD)
apply(simp del: Un-subset-iff)
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  $x = i$  and  $P = \lambda i. ?H i \longrightarrow \models (?J i) \text{ sat } [?I i, ?K i, ?M i, ?N i]$  in
  allE, erule impE, assumption)
apply(simp add: com-validity-def)
apply(erule-tac  $x = s$  in allE)
apply(erule-tac  $x = i$  and  $P = \lambda j. ?H j \longrightarrow (?I j) \in \text{cp } (?J j) s$  in allE, simp)
apply(drule-tac  $c = clist ! i$  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  $P = \lambda j. ?H j \longrightarrow \text{fst } (?I j) = (?J j)$  in allE)
apply (simp add: All-None-def same-length-def)
apply(erule-tac  $x = i$  and  $P = \lambda j. ?H j \longrightarrow \text{length } (?J j) = (?K j)$  in allE)
apply(subgoal-tac  $\text{length } x - 1 < \text{length } x$ , simp)
apply(case-tac  $x \neq []$ )
apply(simp add: last-conv-nth)
apply(erule-tac  $x = clist ! i$  in ballE)
apply(simp add: same-state-def)

```

```

    apply(subgoal-tac clist!i≠[])
    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=ia and P=λj. (?T j) → (∀ i. (?H j i) → (snd (?d j i))=(snd
    (?e j i))) in allE,simp)
    apply(erule-tac x=0 and P=λj. ?H j → (snd (?d j))=(snd (?e j)) in allE)
    apply(case-tac x,simp+)
    apply (simp add:par-assum-def)
    apply clarify
    apply(drule-tac c=snd (clist ! ia ! 0) in subsetD)
    apply assumption
    apply simp
    apply clarify
    apply(erule-tac x=ia in all-dupE)
    apply(rule subsetD, erule mp, assumption)
    apply(erule-tac pre=pre and rely=rely and x=x and s=s in three)
    apply(erule-tac x=ic 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]:
  ∀ i s. x ∈ par-cp (ParallelCom []) s →
    Suc i < length x → (x ! i, x ! Suc i) ∉ par-ctran
    apply(induct-tac x)
    apply(simp add:par-cp-def ParallelCom-def)
    apply clarify
    apply(case-tac list,simp,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:
   $\vdash c \text{ SAT } [pre, rely, guar, post] \implies$ 
   $\models (\text{ParallelCom } c) \text{ SAT } [pre, rely, guar, post]$ 
apply(erule par-rghoare.induct)
apply(case-tac xs,simp)
apply(simp add:par-com-validity-def par-comm-def)
apply clarify
apply(case-tac post=UNIV,simp)
apply clarify
apply(drule ParallelEmpty)
apply assumption
apply simp
apply clarify
apply simp
apply(subgoal-tac xs $\neq$ [])
prefer 2
apply simp
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* \equiv *Basic id*

notation *Seq* ((-;;/-) [60,61] 60)

syntax

-Assign :: *idt* \Rightarrow 'b \Rightarrow 'a com ((' - := / -) [70, 65] 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 ((0IF - THEN - FI) [0,0] 56)
-While :: 'a bexp \Rightarrow 'a com \Rightarrow 'a com ((0WHILE - / DO - / OD) [0,
0] 61)
-Await :: 'a bexp \Rightarrow 'a com \Rightarrow 'a com ((0AWAIT - / THEN - / END)
[0,0] 61)
-Atom :: 'a com \Rightarrow 'a com (((<-) 61)
-Wait :: 'a bexp \Rightarrow 'a com ((0WAIT - END) 61)

translations

'x := a \rightarrow CONST Basic \ll '(-update-name x (λ -. a)) \gg
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 DO c OD \rightarrow CONST While .{b}. c
AWAIT b THEN c END \rightleftharpoons CONST Await .{b}. c
<c> \rightleftharpoons AWAIT CONST True THEN c END
WAIT b END \rightleftharpoons AWAIT b THEN SKIP END

nonterminals

prgs

syntax

-PAR :: *prgs* \Rightarrow 'a (COBEGIN // // COEND 60)
-prg :: 'a \Rightarrow *prgs* (- 57)
-prgs :: ['a, *prgs*] \Rightarrow *prgs* (- // // - [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* (SCHEME [- \leq - < -] - [0,0,0,60] 57)

translations

-prg-scheme *j i k c* \rightleftharpoons (CONST map (λ i. c) [*j*..*k*])

Translations for variables before and after a transition:

syntax

$\text{-before} :: id \Rightarrow 'a \text{ } (^{\circ}\text{-})$
 $\text{-after} :: id \Rightarrow 'a \text{ } (^{\text{a}}\text{-})$

translations

$\text{}^{\circ}x \rightleftharpoons x \text{ ' } \text{CONST fst}$
 $\text{}^{\text{a}}x \rightleftharpoons x \text{ ' } \text{CONST snd}$

print-translation \ll

```

let
  fun quote-tr' f (t :: ts) =
    Term.list-comb (f $ Syntax.quote-tr' @{\syntax-const -antiquote} t, ts)
  | quote-tr' - = raise Match;

  val assert-tr' = quote-tr' (Syntax.const @{\syntax-const -Assert});

  fun bexp-tr' name ((Const (@{const-syntax Collect}, -) $ t) :: ts) =
    quote-tr' (Syntax.const name) (t :: ts)
  | bexp-tr' - = raise Match;

  fun K-tr' (Abs (-, -, t)) =
    if null (loose-bnos t) then t else raise Match
  | K-tr' (Abs (-, -, Abs (-, -, t) $ Bound 0)) =
    if null (loose-bnos t) then t else raise Match
  | K-tr' - = raise Match;

  fun assign-tr' (Abs (x, -, f $ k $ Bound 0) :: ts) =
    quote-tr' (Syntax.const @{\syntax-const -Assign} $ Syntax.update-name-tr'
f)
      (Abs (x, dummyT, K-tr' k) :: ts)
  | assign-tr' - = raise Match;

in
  [(@{const-syntax Collect}, assert-tr'),
   (@{const-syntax Basic}, assign-tr'),
   (@{const-syntax Cond}, bexp-tr' @{\syntax-const -Cond}),
   (@{const-syntax While}, bexp-tr' @{\syntax-const -While})]
end
\gg
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 $[0 \leq i < n]$
 $(\text{'A} := \text{'A} [i := 0],$
 $\{\!\{ n < \text{length } \text{'A} \}\!\},$
 $\{\!\{ \text{length } {}^\circ A = \text{length } {}^a A \wedge {}^\circ A ! i = {}^a A ! i \}\!\},$
 $\{\!\{ \text{length } {}^\circ A = \text{length } {}^a A \wedge (\forall j < n. i \neq j \longrightarrow {}^\circ A ! j = {}^a A ! j) \}\!\},$
 $\{\!\{ \text{'A} ! i = 0 \}\!\})$
COEND
SAT $\llbracket \{\!\{ n < \text{length } \text{'A} \}\!\}, \{\!\{ {}^\circ A = {}^a A \}\!\}, \{\!\{ \text{True} \}\!\}, \{\!\{ \forall i < n. \text{'A} ! i = 0 \}\!\} \rrbracket$
apply(*rule Parallel*)
apply (*auto intro!*: *Basic*)
done

lemma *Example1-parameterized*:

$k < t \implies$
⊢ *COBEGIN*
SCHEME $[k * n \leq i < (\text{Suc } k) * n] (\text{'A} := \text{'A} [i := 0],$
 $\{\!\{ t * n < \text{length } \text{'A} \}\!\},$
 $\{\!\{ t * n < \text{length } {}^\circ A \wedge \text{length } {}^\circ A = \text{length } {}^a A \wedge {}^\circ A ! i = {}^a A ! i \}\!\},$
 $\{\!\{ t * n < \text{length } {}^\circ A \wedge \text{length } {}^\circ A = \text{length } {}^a A \wedge (\forall j < \text{length } {}^\circ A. i \neq j \longrightarrow {}^\circ A ! j =$
 ${}^a A ! j) \}\!\},$
 $\{\!\{ \text{'A} ! i = 0 \}\!\})$
COEND
SAT $\llbracket \{\!\{ t * n < \text{length } \text{'A} \}\!\},$
 $\{\!\{ t * n < \text{length } {}^\circ A \wedge \text{length } {}^\circ A = \text{length } {}^a A \wedge (\forall i < n. {}^\circ A ! (k * n + i) = {}^a A ! (k * n + i)) \}\!\},$
 $\{\!\{ t * n < \text{length } {}^\circ A \wedge \text{length } {}^\circ A = \text{length } {}^a A \wedge$
 $(\forall i < \text{length } {}^\circ A. (i < k * n \longrightarrow {}^\circ A ! i = {}^a A ! i) \wedge ((\text{Suc } k) * n \leq i \longrightarrow {}^\circ A ! i =$
 ${}^a A ! i)) \}\!\},$
 $\{\!\{ \forall i < n. \text{'A} ! (k * n + i) = 0 \}\!\} \rrbracket$
apply(*rule Parallel*)
apply *auto*
apply(*erule-tac* $x = k * n + i$ **in** *allE*)
apply(*subgoal-tac* $k * n + i < \text{length } (A \ b)$)
apply *force*
apply(*erule le-less-trans2*)
apply(*case-tac* $t, \text{simp}+$)
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
  (⟨ 'x:= 'x+1;; 'c-0:= 'c-0 + 1 ⟩,
   { 'x= 'c-0 + 'c-1 ∧ 'c-0=0 },
   {oc-0 = ac-0 ∧
    ( ox=oc-0 + oc-1
      ⟶ ax = ac-0 + ac-1 ) },
   {oc-1 = ac-1 ∧
    ( ox=oc-0 + oc-1
      ⟶ ax = ac-0 + ac-1 ) },
   { 'x= 'c-0 + 'c-1 ∧ 'c-0=1 })
||
  (⟨ 'x:= 'x+1;; 'c-1:= 'c-1+1 ⟩,
   { 'x= 'c-0 + 'c-1 ∧ 'c-1=0 },
   {oc-1 = ac-1 ∧
    ( ox=oc-0 + oc-1
      ⟶ ax = ac-0 + ac-1 ) },
   {oc-0 = ac-0 ∧
    ( ox=oc-0 + oc-1
      ⟶ ax = ac-0 + ac-1 ) },
   { 'x= 'c-0 + 'c-1 ∧ 'c-1=1 })
COEND
SAT [{ 'x=0 ∧ 'c-0=0 ∧ 'c-1=0 },
    {ox=ax ∧ oc-0= ac-0 ∧ oc-1=ac-1 },
    { True },
    { 'x=2 }]
apply(rule Parallel)
  apply simp-all
  apply clarify

```

```

apply(case-tac i)
  apply simp
  apply(rule conjI)
    apply clarify
    apply simp
    apply clarify
    apply simp
    apply(case-tac j, simp)
    apply simp
  apply simp
apply(rule conjI)
  apply clarify
  apply simp
apply clarify
apply simp
apply(subgoal-tac j=0)
  apply (simp)
  apply arith
apply clarify
apply(case-tac i, simp, simp)
apply clarify
apply simp
apply(erule-tac x=0 in all-dupE)
apply(erule-tac x=1 in allE, simp)
apply clarify
apply(case-tac i, 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(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:
   $j \leq s \implies (\sum_{i::nat=0..<j}. (b \ (s:=t)) \ i) = (\sum_{i=0..<j}. b \ i)$ 
apply (induct  $j$ )
apply (simp-all cong:setsum-cong)
done

```

```

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  $t = setsum \ (b \ (j := (Suc \ 0))) \ \{0..<n\}$ ) in ssubst
apply (frule-tac  $b = b$ ) in Example2-lemma2-aux
apply (erule-tac  $t = setsum \ b \ \{0..<n\}$ ) in ssubst
apply (subgoal-tac  $Suc \ (setsum \ b \ \{0..<j\} + b \ j + (\sum_{i=0..<n - Suc \ j}. b \ (Suc \ j + i))) = (setsum \ b \ \{0..<j\} + Suc \ (b \ j) + (\sum_{i=0..<n - Suc \ j}. b \ (Suc \ j + i)))$ )
apply (rotate-tac  $-1$ )
apply (erule ssubst)
apply (subgoal-tac  $j \leq j$ )
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:  $0 < n \implies$ 
   $\vdash COBEGIN \ SCHEME \ [0 \leq i < n]$ 
   $(\langle \ 'y := \ 'y + 1;; \ 'C := \ 'C \ (i := 1) \ \rangle,$ 
   $\{\ 'y = (\sum_{i=0..<n}. \ 'C \ i) \wedge \ 'C \ i = 0 \},$ 
   $\{\!^o C \ i = \ ^a C \ i \wedge$ 

```

```

      ( $\circ y = (\sum i=0..<n. \circ C i) \longrightarrow {}^a y = (\sum i=0..<n. {}^a C i)$ ),
       $\{\{(\forall j<n. i \neq j \longrightarrow \circ C j = {}^a C j) \wedge$ 
      ( $\circ y = (\sum i=0..<n. \circ C i) \longrightarrow {}^a y = (\sum i=0..<n. {}^a C i)$ )\},
       $\{\{y' = (\sum i=0..<n. ' C i) \wedge ' C i = 1\}\}$ )
    COEND
    SAT [ $\{\{y' = 0 \wedge (\sum i=0..<n. ' C i) = 0\}\}, \{\{C = {}^a C \wedge \circ y = {}^a y\}\}, \{\{True\}\}, \{\{y' = n\}\}]$ 
  apply (rule Parallel)
  apply force
  apply force
  apply (force)
  apply clarify
  apply simp
  apply (simp cong:setsum-ivl-cong)
  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+
  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  $a = a \div n * n + a \bmod n$ )
    prefer 2 apply (simp (no-asm-use))
  apply (subgoal-tac  $j = j \div n * n + j \bmod n$ )
    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* =

$X :: \text{nat} \Rightarrow \text{nat}$

$Y :: \text{nat} \Rightarrow \text{nat}$

lemma *Example3*: $m \bmod n = 0 \implies$

$\vdash \text{COBEGIN}$

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,

$\{\{('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < m \longrightarrow P(B!('Y\ i)) \wedge 'Y\ i \leq m+i)\}\}$,

$\{\{(\forall j < n. i \neq j \longrightarrow {}^a Y\ j \leq {}^o Y\ j) \wedge {}^o X\ i = {}^a X\ i \wedge {}^o Y\ i = {}^a Y\ i\}\}$,

$\{\{(\forall j < n. i \neq j \longrightarrow {}^o X\ j = {}^a X\ j \wedge {}^o Y\ j = {}^a Y\ j) \wedge {}^a Y\ i \leq {}^o Y\ i\}\}$,

$\{\{('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < m \longrightarrow P(B!('Y\ i)) \wedge 'Y\ i \leq m+i) \wedge (\exists j < n. 'Y\ j \leq 'X\ i)\}\}$

COEND

SAT $[\{\{ \forall i < n. 'X\ i = i \wedge 'Y\ i = m+i \}\}, \{\{ {}^o X = {}^a X \wedge {}^o Y = {}^a Y \}\}, \{\{ \text{True} \}\},$

$\{\{ \forall i < n. ('X\ i) \bmod n = i \wedge (\forall j < 'X\ i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < m \longrightarrow P(B!('Y\ i)) \wedge 'Y\ i \leq m+i) \wedge (\exists j < n. 'Y\ j \leq 'X\ i) \}\}\}$

apply(*rule Parallel*)

— 5 subgoals left

apply *force*+

apply *clarify*

apply *simp*

apply(*rule While*)

apply *force*

apply *force*

apply *force*

apply(*rule-tac pre'* $\{\{ 'X\ i \bmod n = i \wedge (\forall j. j < 'X\ i \longrightarrow j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y\ i < n * q \longrightarrow P(B!('Y\ i))) \wedge 'X\ i < 'Y\ i \}\}$ **in** *Conseq*)

apply *force*

apply(*rule subset-refl*) +

apply(*rule Cond*)

apply *force*

apply(*rule Basic*)

apply *force*

apply *fastsimp*

apply *force*

apply *force*

apply(*rule Basic*)

apply *simp*

apply *clarify*

apply *simp*

apply (*case-tac* $X\ x\ (j \bmod n) \leq j$)

apply (*drule le-imp-less-or-eq*)

apply (*erule disjE*)
apply (*drule-tac* $j=j$ **and** $n=n$ **and** $i=j \bmod n$ **and** $a=X \ x \ (j \bmod n)$ **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*: $m \bmod n = 0 \implies \vdash (\text{COBEGIN SCHEME } [0 \leq i < n]$
 (*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,
 $\{n < \text{length } 'X \wedge n < \text{length } 'Y \wedge ('X!i) \bmod n = i \wedge (\forall j < 'X!i. j \bmod n = i \longrightarrow$
 $\neg P(B!j)) \wedge ('Y!i < m \longrightarrow P(B!('Y!i))) \wedge 'Y!i \leq m + i\}$,
 $\{(\forall j < n. i \neq j \longrightarrow {}^a Y!j \leq {}^o Y!j) \wedge {}^o X!i = {}^a X!i \wedge$
 ${}^o Y!i = {}^a Y!i \wedge \text{length } {}^o X = \text{length } {}^a X \wedge \text{length } {}^o Y = \text{length } {}^a Y\}$,
 $\{(\forall j < n. i \neq j \longrightarrow {}^o X!j = {}^a X!j \wedge {}^o Y!j = {}^a Y!j) \wedge$
 ${}^a Y!i \leq {}^o Y!i \wedge \text{length } {}^o X = \text{length } {}^a X \wedge \text{length } {}^o Y = \text{length } {}^a Y\}$,
 $\{('X!i) \bmod n = i \wedge (\forall j < 'X!i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y!i < m \longrightarrow P(B!('Y!i)))$
 $\wedge 'Y!i \leq m + i) \wedge (\exists j < n. 'Y!j \leq 'X!i) \}$ *COEND*)
SAT [$\{n < \text{length } 'X \wedge n < \text{length } 'Y \wedge (\forall i < n. 'X!i = i \wedge 'Y!i = m + i) \}$,
 $\{{}^o X = {}^a X \wedge {}^o Y = {}^a Y\}$,
 $\{\text{True}\}$,
 $\{\forall i < n. ('X!i) \bmod n = i \wedge (\forall j < 'X!i. j \bmod n = i \longrightarrow \neg P(B!j)) \wedge$
 $('Y!i < m \longrightarrow P(B!('Y!i))) \wedge 'Y!i \leq m + i) \wedge (\exists j < n. 'Y!j \leq 'X!i)\}$]

apply(*rule Parallel*)

— 5 subgoals left

apply *force+*

apply *clarify*

apply *simp*

apply(*rule While*)

apply *force*

apply *force*

apply *force*

apply(*rule-tac* $\text{pre}' = \{n < \text{length } 'X \wedge n < \text{length } 'Y \wedge 'X!i \bmod n = i \wedge (\forall j.$
 $j < 'X!i \longrightarrow j \bmod n = i \longrightarrow \neg P(B!j)) \wedge ('Y!i < n * q \longrightarrow P(B!('Y!$
 $!i))) \wedge 'X!i < 'Y!i\}$ **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 ≤ 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.