Hoare Logic for Parallel Programs

Leonor Prensa Nieto

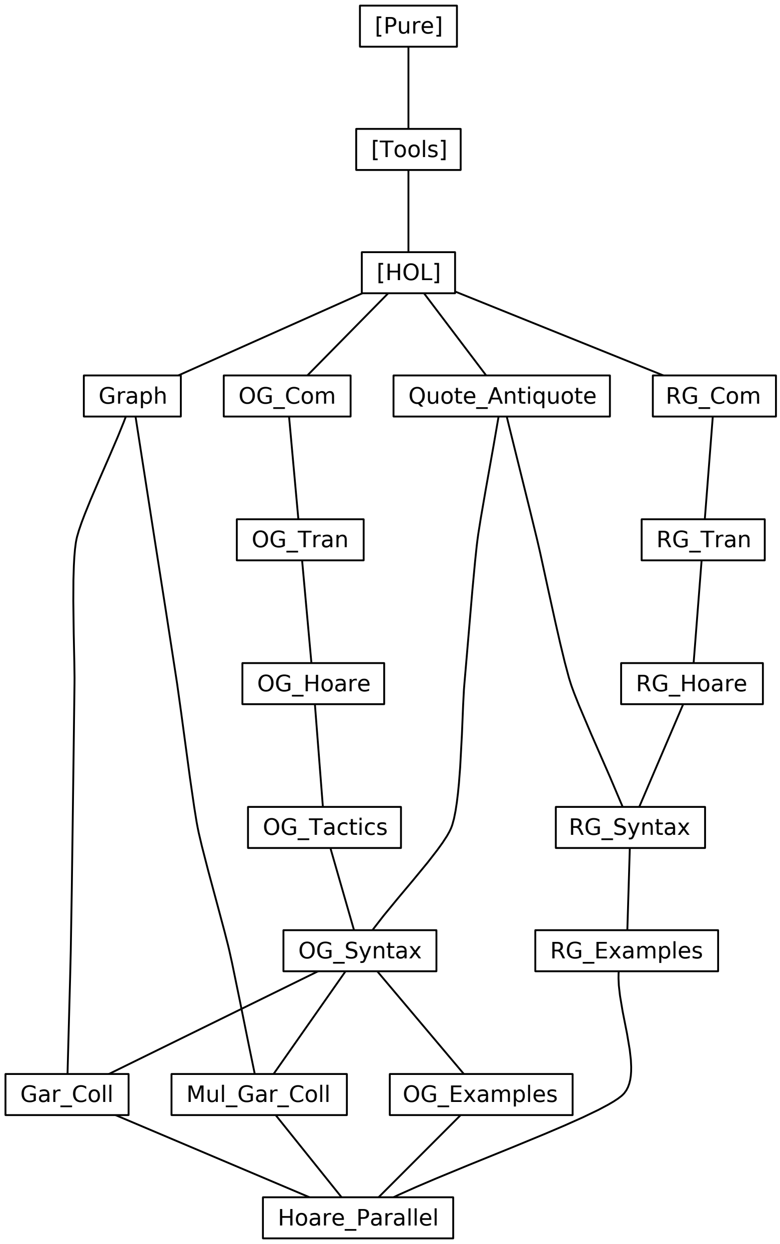
January 18, 2026

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


[Pure]
[Tools]
[HOL]
Graph
OG_Com
Quote_Antiquote
RG_Com
OG_Tran
RG_Tran
OG_Hoare
RG_Hoare
OG_Tactics
RG_Syntax
OG_Syntax
RG_Examples
Gar_Coll
Mul_Gar_Coll
OG_Examples
Hoare_Parallel

Chapter 1

The Owicki-Gries Method

1.1 Abstract Syntax

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

Type abbreviations for boolean expressions and assertions:

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

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

datatype *'a ann-com* =
  AnnBasic (*'a assn*) (*'a* ⇒ *'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* × *'a assn*) *list*
  | *Basic* (*'a* ⇒ *'a*)
  | *Seq* (*'a com*) (*'a com*)
  | *Cond* (*'a bexp*) (*'a com*) (*'a com*)
  | *While* (*'a bexp*) (*'a assn*) (*'a com*)

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

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

Well-formedness predicate for atomic programs:

primrec *atom-com* :: *'a com* ⇒ *bool* **where**
atom-com (*Parallel Ts*) = *False*
| *atom-com* (*Basic f*) = *True*
| *atom-com* (*Seq c1 c2*) = (*atom-com c1* ∧ *atom-com c2*)
| *atom-com* (*Cond b c1 c2*) = (*atom-com c1* ∧ *atom-com c2*)
| *atom-com* (*While b i c*) = *atom-com c*

end

1.2 Operational Semantics

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

type-synonym *'a ann-com-op* = (*'a ann-com*) *option*
type-synonym *'a ann-triple-op* = (*'a ann-com-op* × *'a assn*)

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

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

definition *All-None* :: *'a ann-triple-op list* ⇒ *bool* **where**
All-None Ts ≡ ∀(*c*, *q*) ∈ *set Ts*. *c* = *None*

1.2.1 The Transition Relation

inductive-set
ann-transition :: ((*'a ann-com-op* × *'a*) × (*'a ann-com-op* × *'a*)) *set*
and *transition* :: ((*'a com* × *'a*) × (*'a com* × *'a*)) *set*
and *ann-transition'* :: (*'a ann-com-op* × *'a*) ⇒ (*'a ann-com-op* × *'a*) ⇒ *bool*
(‹- −1→ -›[81,81] 100)
and *transition'* :: (*'a com* × *'a*) ⇒ (*'a com* × *'a*) ⇒ *bool*
(‹- −P1→ -›[81,81] 100)
and *transitions* :: (*'a com* × *'a*) ⇒ (*'a com* × *'a*) ⇒ *bool*
(‹- −P∗→ -›[81,81] 100)
where
con-0 −1→ *con-1* ≡ (*con-0*, *con-1*) ∈ *ann-transition*
| *con-0* −P1→ *con-1* ≡ (*con-0*, *con-1*) ∈ *transition*
| *con-0* −P∗→ *con-1* ≡ (*con-0*, *con-1*) ∈ *transition**

| *AnnBasic*: (*Some* (*AnnBasic r f*), *s*) −1→ (*None*, *f s*)

| *AnnSeq1*: (*Some c0*, *s*) −1→ (*None*, *t*) ⟹
(*Some* (*AnnSeq c0 c1*), *s*) −1→ (*Some c1*, *t*)
| *AnnSeq2*: (*Some c0*, *s*) −1→ (*Some c2*, *t*) ⟹
(*Some* (*AnnSeq c0 c1*), *s*) −1→ (*Some* (*AnnSeq c2 c1*), *t*)

| *AnnCond1T*: *s* ∈ *b* ⟹ (*Some* (*AnnCond1 r b c1 c2*), *s*) −1→ (*Some c1*, *s*)

| AnnCond1F: s ∉ b ⟹ (Some (AnnCond1 r b c1 c2), s) −1→ (Some c2, s)

| AnnCond2T: s ∈ b ⟹ (Some (AnnCond2 r b c), s) −1→ (Some c, s)
| AnnCond2F: s ∉ b ⟹ (Some (AnnCond2 r b c), s) −1→ (None, s)

| AnnWhileF: s ∉ b ⟹ (Some (AnnWhile r b i c), s) −1→ (None, s)
| AnnWhileT: s ∈ b ⟹ (Some (AnnWhile r b i c), s) −1→
(Some (AnnSeq c (AnnWhile i b i c)), s)

| AnnAwait: ⟦ s ∈ b; atom-com c; (c, s) −P∗→ (Parallel [], t) ⟧ ⟹
(Some (AnnAwait r b c), s) −1→ (None, t)

| Parallel: ⟦ i<length Ts; Ts!i = (Some c, q); (Some c, s) −1→ (r, t) ⟧
⟹ (Parallel Ts, s) −P1→ (Parallel (Ts [i:=(r, q)]), t)

| Basic: (Basic f, s) −P1→ (Parallel [], f s)

| Seq1: All-None Ts ⟹ (Seq (Parallel Ts) c, s) −P1→ (c, s)
| Seq2: (c0, s) −P1→ (c2, t) ⟹ (Seq c0 c1, s) −P1→ (Seq c2 c1, t)

| CondT: s ∈ b ⟹ (Cond b c1 c2, s) −P1→ (c1, s)
| CondF: s ∉ b ⟹ (Cond b c1 c2, s) −P1→ (c2, s)

| WhileF: s ∉ b ⟹ (While b i c, s) −P1→ (Parallel [], s)
| WhileT: s ∈ b ⟹ (While b i c, s) −P1→ (Seq c (While b i c), s)

monos *rtrancl-mono*

The corresponding abbreviations are:

abbreviation
ann-transition-n :: ('a ann-com-op × 'a) ⇒ nat ⇒ ('a ann-com-op × 'a)
⇒ bool (‹- −-→ -›[81,81] 100) **where**
con-0 −n→ con-1 ≡ (con-0, con-1) ∈ ann-transition ^^ n

abbreviation
ann-transitions :: ('a ann-com-op × 'a) ⇒ ('a ann-com-op × 'a) ⇒ bool
(‹- −∗→ -›[81,81] 100) **where**
con-0 −∗→ con-1 ≡ (con-0, con-1) ∈ ann-transition*

abbreviation
transition-n :: ('a com × 'a) ⇒ nat ⇒ ('a com × 'a) ⇒ bool
(‹- −P-→ -›[81,81,81] 100) **where**
con-0 −Pn→ con-1 ≡ (con-0, con-1) ∈ transition ^^ n

1.2.2 Definition of Semantics

definition *ann-sem* :: 'a ann-com ⇒ 'a ⇒ 'a set **where**
ann-sem c ≡ λs. {t. (Some c, s) −∗→ (None, t)}

definition *ann-SEM* :: *'a ann-com* ⇒ *'a set* ⇒ *'a set* **where**
ann-SEM c S ≡ ⋃(*ann-sem c* ‘ *S*)

definition *sem* :: *'a com* ⇒ *'a* ⇒ *'a set* **where**
sem c ≡ λ*s*. {*t*. ∃ *Ts*. (*c*, *s*) −*P*∗→ (*Parallel Ts*, *t*) ∧ *All-None Ts*}

definition *SEM* :: *'a com* ⇒ *'a set* ⇒ *'a set* **where**
SEM c S ≡ ⋃(*sem c* ‘ *S*)

abbreviation *Omega* :: *'a com* (‹Ω› *63*)
where Ω ≡ *While UNIV UNIV* (*Basic id*)

primrec *fwhile* :: *'a bexp* ⇒ *'a com* ⇒ *nat* ⇒ *'a com* **where**
fwhile b c 0 = Ω
| *fwhile b c* (*Suc n*) = *Cond b* (*Seq c* (*fwhile b c n*)) (*Basic id*)

Proofs

declare *ann-transition-transition.intros* [*intro*]
inductive-cases *transition-cases*:
(*Parallel T*,*s*) −*P1*→ *t*
(*Basic f*, *s*) −*P1*→ *t*
(*Seq c1 c2*, *s*) −*P1*→ *t*
(*Cond b c1 c2*, *s*) −*P1*→ *t*
(*While b i c*, *s*) −*P1*→ *t*

lemma *Parallel-empty-lemma* [*rule-format* (*no-asm*)]:
(*Parallel* [],*s*) −*Pn*→ (*Parallel Ts*,*t*) ⟶ *Ts*=[] ∧ *n*=*0* ∧ *s*=*t*
apply(*induct n*)
apply(*simp* (*no-asm*))
apply *clarify*
apply(*drule relpow-Suc-D2*)
apply(*force elim*:*transition-cases*)
done

lemma *Parallel-AllNone-lemma* [*rule-format* (*no-asm*)]:
All-None Ss ⟶ (*Parallel Ss*,*s*) −*Pn*→ (*Parallel Ts*,*t*) ⟶ *Ts*=*Ss* ∧ *n*=*0* ∧ *s*=*t*
apply(*induct n*)
apply(*simp* (*no-asm*))
apply *clarify*
apply(*drule relpow-Suc-D2*)
apply *clarify*
apply(*erule transition-cases*,*simp-all*)
apply(*force dest*:*nth-mem simp add*:*All-None-def*)
done

lemma *Parallel-AllNone*: *All-None Ts* ⟹ (*SEM* (*Parallel Ts*) *X*) = *X*
apply (*unfold SEM-def sem-def*)
apply *auto*

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

lemma *Parallel-empty*: *Ts*=[] ⟹ (*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*⊆*Y* ⟹ *SEM c X* ⊆ *SEM c Y*
apply (*unfold SEM-def*)
apply *force*
done

lemma *L3-5ii-lemma1*:
⟦ (*c1*, *s1*) −*P*∗→ (*Parallel Ts*, *s2*); *All-None Ts*;
(*c2*, *s2*) −*P*∗→ (*Parallel Ss*, *s3*); *All-None Ss* ⟧
⟹ (*Seq c1 c2*, *s1*) −*P*∗→ (*Parallel Ss*, *s3*)
apply(*erule converse-rtrancl-induct2*)
apply(*force intro*:*converse-rtrancl-into-rtrancl*)+
done

lemma *L3-5ii-lemma2* [*rule-format* (*no-asm*)]:
∀ *c1 c2 s t*. (*Seq c1 c2*, *s*) −*Pn*→ (*Parallel Ts*, *t*) ⟶
(*All-None Ts*) ⟶ (∃ *y m Rs*. (*c1*,*s*) −*P*∗→ (*Parallel Rs*, *y*) ∧
(*All-None Rs*) ∧ (*c2*, *y*) −*Pm*→ (*Parallel Ts*, *t*) ∧ *m* ≤ *n*)
apply(*induct n*)
apply(*force*)
apply(*safe dest*!: *relpow-Suc-D2*)
apply(*erule transition-cases*,*simp-all*)
apply (*fast intro*!: *le-SucI*)
apply (*fast intro*!: *le-SucI elim*!: *relpow-imp-rtrancl converse-rtrancl-into-rtrancl*)
done

lemma *L3-5ii-lemma3*:
⟦(*Seq c1 c2*,*s*) −*P*∗→ (*Parallel Ts*,*t*); *All-None Ts*⟧ ⟹
(∃ *y Rs*. (*c1*,*s*) −*P*∗→ (*Parallel Rs*,*y*) ∧ *All-None Rs*
∧ (*c2*,*y*) −*P*∗→ (*Parallel Ts*,*t*))
apply(*drule rtrancl-imp-UN-relpow*)
apply(*fast dest*: *L3-5ii-lemma2 relpow-imp-rtrancl*)
done

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

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

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

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

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

lemma *L3-5v-lemma2*: ⟦(Ω, *s*) −*P*∗→ (*Parallel Ts*, *t*); *All-None Ts* ⟧ ⟹ *False*
apply(*fast dest*: *rtrancl-imp-UN-relpow L3-5v-lemma1*)
done

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

done

lemma *L3-5v-lemma4* [*rule-format*]:
∀ *s*. (*While b i c*, *s*) −*Pn*→ (*Parallel Ts*, *t*) ⟶ *All-None Ts* ⟶
(∃ *k*. (*fwhile b c k*, *s*) −*P*∗→ (*Parallel Ts*, *t*))
apply(*rule nat-less-induct*)
apply *safe*
apply(*erule relpow-E2*)
apply *safe*
apply(*erule transition-cases*,*simp-all*)
apply (*rule-tac x* = *1* **in** *exI*)
apply(*force dest*: *Parallel-empty-lemma intro*: *converse-rtrancl-into-rtrancl simp add*: *Id-def*)
apply *safe*
apply(*drule L3-5ii-lemma2*)
apply *safe*
apply(*drule le-imp-less-Suc*)
apply (*erule allE* , *erule impE*,*assumption*)
apply (*erule allE* , *erule impE*, *assumption*)
apply *safe*
apply (*rule-tac x* = *k*+*1* **in** *exI*)
apply(*simp* (*no-asm*))
apply(*rule converse-rtrancl-into-rtrancl*)
apply *fast*
apply(*fast elim*: *L3-5ii-lemma1*)
done

lemma *L3-5v-lemma5* [*rule-format*]:
∀ *s*. (*fwhile b c k*, *s*) −*P*∗→ (*Parallel Ts*, *t*) ⟶ *All-None Ts* ⟶
(*While b i c*, *s*) −*P*∗→ (*Parallel Ts*,*t*)
apply(*induct k*)
apply(*force dest*: *L3-5v-lemma2*)
apply *safe*
apply(*erule converse-rtranclE*)
apply *simp-all*
apply(*erule transition-cases*,*simp-all*)
apply(*rule converse-rtrancl-into-rtrancl*)
apply(*fast*)
apply(*fast elim*!: *L3-5ii-lemma1 dest*: *L3-5ii-lemma3*)
apply(*drule rtrancl-imp-UN-relpow*)
apply *clarify*
apply(*erule relpow-E2*)
apply *simp-all*
apply(*erule transition-cases*,*simp-all*)
apply(*fast dest*: *Parallel-empty-lemma*)
done

lemma *L3-5v*: *SEM* (*While b i c*) = $(\lambda x.\ (\bigcup k.\ SEM\ (fwhile\ b\ c\ k)\ x))$
apply(*rule ext*)

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

1.3 Validity of Correctness Formulas

definition *com-validity* :: *′a assn* ⇒ *′a com* ⇒ *′a assn* ⇒ *bool* (‹(*3*‖= -// -//-)› [*90*,*55*,*90*] *50*) **where**
 ‖= *p c q* ≡ *SEM c p* ⊆ *q*

definition *ann-com-validity* :: *′a ann-com* ⇒ *′a assn* ⇒ *bool* (‹⊨ - -› [*60*,*90*] *45*) **where**
 ⊨ *c q* ≡ *ann-SEM c* (*pre c*) ⊆ *q*

end

1.4 The Proof System

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

primrec *assertions* :: *′a ann-com* ⇒ (*′a assn*) *set* **where**
 assertions (*AnnBasic r f*) = {*r*}
| *assertions* (*AnnSeq c1 c2*) = *assertions c1* ∪ *assertions c2*
| *assertions* (*AnnCond1 r b c1 c2*) = {*r*} ∪ *assertions c1* ∪ *assertions c2*
| *assertions* (*AnnCond2 r b c*) = {*r*} ∪ *assertions c*
| *assertions* (*AnnWhile r b i c*) = {*r*, *i*} ∪ *assertions c*
| *assertions* (*AnnAwait r b c*) = {*r*}

primrec *atomics* :: *′a ann-com* ⇒ (*′a assn* × *′a com*) *set* **where**
 atomics (*AnnBasic r f*) = {(*r*, *Basic f*)}
| *atomics* (*AnnSeq c1 c2*) = *atomics c1* ∪ *atomics c2*
| *atomics* (*AnnCond1 r b c1 c2*) = *atomics c1* ∪ *atomics c2*
| *atomics* (*AnnCond2 r b c*) = *atomics c*
| *atomics* (*AnnWhile r b i c*) = *atomics c*
| *atomics* (*AnnAwait r b c*) = {(*r* ∩ *b*, *c*)}

primrec *com* :: *′a ann-triple-op* ⇒ *′a ann-com-op* **where**
 com (*c*, *q*) = *c*

primrec *post* :: *′a ann-triple-op* ⇒ *′a assn* **where**
 post (*c*, *q*) = *q*

definition *interfree-aux* :: (*′a ann-com-op* × *′a assn* × *′a ann-com-op*) ⇒ *bool* **where**

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

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

inductive
$oghoare :: 'a\ assn \Rightarrow 'a\ com \Rightarrow 'a\ assn \Rightarrow bool$ (‹(3‖- -//-//-)› [90,55,90] 50)
and $ann\text{-}hoare :: 'a\ ann\text{-}com \Rightarrow 'a\ assn \Rightarrow bool$ (‹(2⊢ -// -)› [60,90] 45)
where
$AnnBasic$: $r \subseteq \{s.\ f\ s \in q\} \Longrightarrow \vdash (AnnBasic\ r\ f)\ q$

| $AnnSeq$: $[\![\vdash c0\ pre\ c1; \vdash c1\ q\ ]\!] \Longrightarrow \vdash (AnnSeq\ c0\ c1)\ q$

| $AnnCond1$: $[\![\ r \cap b \subseteq pre\ c1; \vdash c1\ q;\ r \cap -b \subseteq pre\ c2; \vdash c2\ q]\!]$
$\Longrightarrow \vdash (AnnCond1\ r\ b\ c1\ c2)\ q$
| $AnnCond2$: $[\![\ r \cap b \subseteq pre\ c; \vdash c\ q;\ r \cap -b \subseteq q\ ]\!] \Longrightarrow \vdash (AnnCond2\ r\ b\ c)\ q$

| $AnnWhile$: $[\![\ r \subseteq i;\ i \cap b \subseteq pre\ c; \vdash c\ i;\ i \cap -b \subseteq q\ ]\!]$
$\Longrightarrow \vdash (AnnWhile\ r\ b\ i\ c)\ q$

| $AnnAwait$: $[\![\ atom\text{-}com\ c;\ \|- (r \cap b)\ c\ q\ ]\!] \Longrightarrow \vdash (AnnAwait\ r\ b\ c)\ q$

| $AnnConseq$: $[\![\vdash c\ q;\ q \subseteq q'\ ]\!] \Longrightarrow \vdash c\ q'$

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

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

| Seq: $[\![\ \|- p\ c1\ r;\ \|- r\ c2\ q\ ]\!] \Longrightarrow \|- p\ (Seq\ c1\ c2)\ q$

| $Cond$: $[\![\ \|- (p \cap b)\ c1\ q;\ \|- (p \cap -b)\ c2\ q\ ]\!] \Longrightarrow \|- p\ (Cond\ b\ c1\ c2)\ q$

| $While$: $[\![\ \|- (p \cap b)\ c\ p\ ]\!] \Longrightarrow \|- p\ (While\ b\ i\ c)\ (p \cap -b)$

| $Conseq$: $[\![\ p' \subseteq p;\ \|- p\ c\ q\ ;\ q \subseteq q'\ ]\!] \Longrightarrow \|- p'\ c\ q'$

1.5 Soundness

lemmas [*cong del*] = *if-weak-cong*

lemmas *ann-hoare-induct* = *oghoare-ann-hoare.induct* [*THEN conjunct2*]
lemmas *oghoare-induct* = *oghoare-ann-hoare.induct* [*THEN conjunct1*]

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

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

1.5.1 Soundness of the System for Atomic Programs

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

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

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

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

1.5.2 Soundness of the System for Component Programs

inductive-cases *ann-transition-cases*:
$(None,s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnBasic\ r\ f),s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnSeq\ c1\ c2),\ s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnCond1\ r\ b\ c1\ c2),\ s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnCond2\ r\ b\ c),\ s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnWhile\ r\ b\ I\ c),\ s)\ -1\rightarrow\ (c',\ s')$
$(Some\ (AnnAwait\ r\ b\ c),s)\ -1\rightarrow\ (c',\ s')$

Strong Soundness for Component Programs:

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

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

lemma *Strong-Soundness-aux-aux* [*rule-format*]:
$(co,\ s)\ -1\rightarrow\ (co',\ t)\ \longrightarrow (\forall c.\ co = Some\ c \longrightarrow s\in\ pre\ c \longrightarrow$
$(\forall q.\ \vdash c\ q \longrightarrow (if\ co' = None\ then\ t{\in}q\ else\ t \in pre(the\ co') \wedge \vdash (the\ co')\ q\ )))$
apply(*rule ann-transition-transition.induct* [*THEN conjunct1*])
apply *simp-all*
— Basic
apply *clarify*
apply(*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*: $[\![$ *(Some c, s)* $-*\rightarrow$ *(co, t)*; $s \in$ *pre c*; $\vdash c\ q$ $]\!]$
$\Longrightarrow$ *if co = None then* $t \in q$ *else* $t \in$ *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*: $[\![$ (*Some c*, *s*)−∗→(*co*, *t*); *s* ∈ *pre c*; ⊢ *c q* $]\!]$
⟹ *if co* = *None then t*∈*q else t* ∈ *pre* (*the co*)
apply(*force dest*:*Strong-Soundness-aux*)
done

lemma *ann-hoare-sound*: ⊢ *c q* ⟹ ⊨ *c q*
apply (*unfold ann-com-validity-def ann-SEM-def ann-sem-def*)
apply *clarify*
apply(*drule Strong-Soundness*)
apply *simp-all*
done

1.5.3 Soundness of the System for Parallel Programs

lemma *Parallel-length-post-P1*: (*Parallel Ts*,*s*) −*P1*→ (*R′*, *t*) ⟹
(∃ *Rs*. *R′* = (*Parallel Rs*) ∧ (*length Rs*) = (*length Ts*) ∧
(∀ *i*. *i*<*length Ts* ⟶ *post*(*Rs* ! *i*) = *post*(*Ts* ! *i*)))
apply(*erule transition-cases*)
apply *simp*
apply *clarify*
apply(*case-tac i*=*ia*)
apply *simp*+
done

lemma *Parallel-length-post-PStar*: (*Parallel Ts*,*s*) −*P*∗→ (*R′*,*t*) ⟹
(∃ *Rs*. *R′* = (*Parallel Rs*) ∧ (*length Rs*) = (*length Ts*) ∧
(∀ *i*. *i*<*length Ts* ⟶ *post*(*Ts* ! *i*) = *post*(*Rs* ! *i*)))
apply(*erule rtrancl-induct2*)
apply(*simp-all*)
apply *clarify*
apply *simp*
apply(*drule Parallel-length-post-P1*)
apply *auto*
done

lemma *assertions-lemma*: *pre c* ∈ *assertions c*
apply(*rule ann-com-com.induct* [*THEN conjunct1*])
apply *auto*
done

```
lemma interfree-aux1 [rule-format]:
  (c,s) −1→ (r,t)  ⟶ (interfree-aux(c1, q1, c) ⟶ 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→ (r,t) ⟶ (interfree-aux(c, q, a)  ⟶ interfree-aux(r, q, a) )
apply (rule ann-transition-transition.induct [THEN conjunct1])
apply(force simp add:interfree-aux-def)+
done

lemma interfree-lemma: ⟦ (Some c, s) −1→ (r, t);interfree Ts ; i<length Ts;
           Ts!i = (Some c, q) ⟧ ⟹ 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:
  ⟦interfree Ts; i<length Ts; com(Ts ! i) = Some(AnnSeq c0 c1) ⟧
  ⟹  interfree (Ts[i:=(Some c0, 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)]:
 ⟦ ∀ i<length Ts. (if com(Ts!i) = None then b ∈ post(Ts!i)
  else b ∈ pre(the(com(Ts!i))) ∧ ⊢ the(com(Ts!i)) post(Ts!i));
  com(Ts ! i) = Some(AnnSeq c0 c1); i<length Ts; interfree Ts ⟧ ⟹
 (∀ ia<length Ts. (if com(Ts[i:=(Some c0, pre c1)]! ia) = None
  then b ∈ post(Ts[i:=(Some c0, pre c1)]! ia)
 else b ∈ pre(the(com(Ts[i:=(Some c0, pre c1)]! ia))) ∧
 ⊢ the(com(Ts[i:=(Some c0, pre c1)]! ia)) post(Ts[i:=(Some c0, pre c1)]! ia)))
```

```
  ∧ 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→ (co, t) ⟶
  (∀ Ts. i<length Ts ⟶ com(Ts ! i) = Some c ⟶
  (∀ i<length Ts. (if com(Ts ! i) = None then b∈post(Ts!i)
  else b∈pre(the(com(Ts!i))) ∧ ⊢ the(com(Ts!i)) post(Ts!i))) ⟶
 interfree Ts ⟶
  (∀ j. j<length Ts ∧ i≠j ⟶ (if com(Ts!j) = None then t∈post(Ts!j)
  else t∈pre(the(com(Ts!j))) ∧ ⊢ 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*]:
$[\![$(*Ts'*,*s*) −*P*∗→ (*Rs'*,*t*); *Ts'* = (*Parallel Ts*); *interfree Ts*;
∀ *i*. *i*<*length Ts* ⟶ (∃ *c q*. (*Ts* ! *i*) = (*Some c*, *q*) ∧ *s*∈(*pre c*) ∧ ⊢ *c q*) $]\!]$ ⟹
∀ *Rs*. *Rs'* = (*Parallel Rs*) ⟶ (∀ *j*. *j*<*length Rs* ⟶
(*if com*(*Rs* ! *j*) = *None then t*∈*post*(*Ts* ! *j*)
else t∈*pre*(*the*(*com*(*Rs* ! *j*))) ∧ ⊢ *the*(*com*(*Rs* ! *j*)) *post*(*Ts* ! *j*))) ∧ *interfree Rs*
apply(*erule rtrancl-induct2*)
apply *clarify*
— Base
apply *force*
— Induction step
apply *clarify*
apply(*drule Parallel-length-post-PStar*)
apply *clarify*
apply (*ind-cases* (*Parallel Ts*, *s*) −*P1*→ (*Parallel Rs*, *t*) **for** *Ts s Rs t*)
apply(*rule conjI*)
apply *clarify*
apply(*case-tac i*=*j*)
apply(*simp split del*:*if-split*)
apply(*erule Strong-Soundness-aux-aux*,*simp*+)
apply *force*
apply *force*
apply(*simp split del*: *if-split*)
apply(*erule Parallel-Strong-Soundness-aux-aux*)
apply(*simp-all add*: *split del*:*if-split*)
apply *force*
apply(*rule interfree-lemma*)
apply *simp-all*
done

lemma *Parallel-Strong-Soundness*:
$[\![$(*Parallel Ts*, *s*) −*P*∗→ (*Parallel Rs*, *t*); *interfree Ts*; *j*<*length Rs*;
∀ *i*. *i*<*length Ts* ⟶ (∃ *c q*. *Ts* ! *i* = (*Some c*, *q*) ∧ *s*∈*pre c* ∧ ⊢ *c q*) $]\!]$ ⟹
if com(*Rs* ! *j*) = *None then t*∈*post*(*Ts* ! *j*) *else t*∈*pre* (*the*(*com*(*Rs* ! *j*)))

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

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

end

1.6 Generation of Verification Conditions

theory *OG-Tactics*
imports *OG-Hoare*
begin

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

```
lemma ParallelConseqRule:
 ⟦ p ⊆ (⋂i∈{i. i<length Ts}. pre(the(com(Ts ! i))));
  ∥- (⋂i∈{i. i<length Ts}. pre(the(com(Ts ! i))))
      (Parallel Ts)
     (⋂i∈{i. i<length Ts}. post(Ts ! i));
  (⋂i∈{i. i<length Ts}. post(Ts ! i)) ⊆ q ⟧
  ⟹ ∥- p (Parallel Ts) q
apply (rule Conseq)
prefer 2
 apply fast
apply assumption+
done

lemma SkipRule: p ⊆ q ⟹ ∥- p (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 ⊆ {s. (f s)∈q} ⟹ ∥- p (Basic f) q
apply(rule oghoare-intros)
  prefer 2 apply(rule oghoare-intros)
 prefer 2 apply(rule subset-refl)
apply assumption
done

lemma SeqRule: ⟦ ∥- p c1 r; ∥- r c2 q ⟧ ⟹ ∥- p (Seq c1 c2) q
apply(rule Seq)
apply fast+
done

lemma CondRule:
 ⟦ p ⊆ {s. (s∈b ⟶ s∈w) ∧ (s∉b ⟶ s∈w')}; ∥- w c1 q; ∥- w' c2 q ⟧
  ⟹ ∥- p (Cond b c1 c2) q
apply(rule Cond)
 apply(rule Conseq)
 prefer 4 apply(rule Conseq)
apply simp-all
apply force+
done

lemma WhileRule: ⟦ p ⊆ i; ∥- (i ∩ b) c i ; (i ∩ (-b)) ⊆ q ⟧
        ⟹ ∥- 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*:
  $[\![\ atom\text{-}com(c);\ \|\!-\ r\ c\ q\ ]\!]\ \Longrightarrow\ \vdash\ (AnnAwait\ r\ \{s.\ True\}\ c)\ q$
apply(*rule AnnAwait*)
apply *simp-all*
done

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

lemma *AnnwaitRule*:
  $[\![\ (r \cap b) \subseteq q\ ]\!] \Longrightarrow\ \vdash\ (AnnAwait\ r\ b\ (Basic\ id))\ q$
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*:
  $\forall\,(R,r)\in(atomics\ a).\ \|\!-\ (q \cap R)\ r\ q \Longrightarrow$ *interfree-aux*(*None*, *q*, *Some a*)
apply(*simp add*:*interfree-aux-def*)
apply(*force elim*:*oghoare-sound*)
done

lemma *interfree-aux-rule3*:
  $(\forall\,(R,\ r)\in(atomics\ a).\ \|\!-\ (q \cap R)\ r\ q \wedge (\forall\,p\in(assertions\ c).\ \|\!-\ (p \cap R)\ r\ p))$
  $\Longrightarrow$ *interfree-aux*(*Some c*, *q*, *Some a*)
apply(*simp add*:*interfree-aux-def*)
apply(*force elim*:*oghoare-sound*)
done

lemma *AnnBasic-assertions*:
  $[\![$*interfree-aux*(*None*, *r*, *Some a*); *interfree-aux*(*None*, *q*, *Some a*)$]\!] \Longrightarrow$
    interfree-aux(*Some* (*AnnBasic r f*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnSeq-assertions*:
⟦ *interfree-aux*(*Some c1*, *q*, *Some a*); *interfree-aux*(*Some c2*, *q*, *Some a*)⟧⟹
interfree-aux(*Some* (*AnnSeq c1 c2*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnCond1-assertions*:
⟦ *interfree-aux*(*None*, *r*, *Some a*); *interfree-aux*(*Some c1*, *q*, *Some a*);
interfree-aux(*Some c2*, *q*, *Some a*)⟧⟹
interfree-aux(*Some*(*AnnCond1 r b c1 c2*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnCond2-assertions*:
⟦ *interfree-aux*(*None*, *r*, *Some a*); *interfree-aux*(*Some c*, *q*, *Some a*)⟧⟹
interfree-aux(*Some* (*AnnCond2 r b c*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnWhile-assertions*:
⟦ *interfree-aux*(*None*, *r*, *Some a*); *interfree-aux*(*None*, *i*, *Some a*);
interfree-aux(*Some c*, *q*, *Some a*)⟧⟹
interfree-aux(*Some* (*AnnWhile r b i c*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnAwait-assertions*:
⟦ *interfree-aux*(*None*, *r*, *Some a*); *interfree-aux*(*None*, *q*, *Some a*)⟧⟹
interfree-aux(*Some* (*AnnAwait r b c*), *q*, *Some a*)
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnBasic-atomics*:
‖− (*q* ∩ *r*) (*Basic f*) *q* ⟹ *interfree-aux*(*None*, *q*, *Some* (*AnnBasic r f*))
by(*simp add*: *interfree-aux-def oghoare-sound*)

lemma *AnnSeq-atomics*:
⟦ *interfree-aux*(*Any*, *q*, *Some a1*); *interfree-aux*(*Any*, *q*, *Some a2*)⟧⟹
interfree-aux(*Any*, *q*, *Some* (*AnnSeq a1 a2*))
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnCond1-atomics*:
⟦ *interfree-aux*(*Any*, *q*, *Some a1*); *interfree-aux*(*Any*, *q*, *Some a2*)⟧⟹
interfree-aux(*Any*, *q*, *Some* (*AnnCond1 r b a1 a2*))
apply(*simp add*: *interfree-aux-def*)
by *force*

lemma *AnnCond2-atomics*:

interfree-aux (*Any*, *q*, *Some a*)⟹ *interfree-aux*(*Any*, *q*, *Some* (*AnnCond2 r b a*))
by(*simp add*: *interfree-aux-def*)

lemma *AnnWhile-atomics*: *interfree-aux* (*Any*, *q*, *Some a*)
⟹ *interfree-aux*(*Any*, *q*, *Some* (*AnnWhile r b i a*))
by(*simp add*: *interfree-aux-def*)

lemma *Annatom-atomics*:
‖− (*q* ∩ *r*) *a q* ⟹ *interfree-aux* (*None*, *q*, *Some* (*AnnAwait r* {*x*. *True*} *a*))
by(*simp add*: *interfree-aux-def oghoare-sound*)

lemma *AnnAwait-atomics*:
‖− (*q* ∩ (*r* ∩ *b*)) *a q* ⟹ *interfree-aux* (*None*, *q*, *Some* (*AnnAwait r b a*))
by(*simp add*: *interfree-aux-def oghoare-sound*)

definition *interfree-swap* :: (*'a ann-triple-op* ∗ (*'a ann-triple-op*) *list*) ⇒ *bool* **where**
interfree-swap == λ(*x*, *xs*). ∀ *y*∈*set xs*. *interfree-aux* (*com x*, *post x*, *com y*)
∧ *interfree-aux*(*com y*, *post y*, *com x*)

lemma *interfree-swap-Empty*: *interfree-swap* (*x*, [])
by(*simp add*:*interfree-swap-def*)

lemma *interfree-swap-List*:
⟦ *interfree-aux* (*com x*, *post x*, *com y*);
interfree-aux (*com y*, *post y* ,*com x*); *interfree-swap* (*x*, *xs*) ⟧
⟹ *interfree-swap* (*x*, *y*#*xs*)
by(*simp add*:*interfree-swap-def*)

lemma *interfree-swap-Map*: ∀ *k*. *i*≤*k* ∧ *k*<*j* ⟶ *interfree-aux* (*com x*, *post x*, *c k*)
∧ *interfree-aux* (*c k*, *Q k*, *com x*)
⟹ *interfree-swap* (*x*, *map* (λ*k*. (*c k*, *Q k*)) [*i*..<*j*])
by(*force simp add*: *interfree-swap-def less-diff-conv*)

lemma *interfree-Empty*: *interfree* []
by(*simp add*:*interfree-def*)

lemma *interfree-List*:
⟦ *interfree-swap*(*x*, *xs*); *interfree xs* ⟧ ⟹ *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*:
(∀ *i j*. *a*≤*i* ∧ *i*<*b* ∧ *a*≤*j* ∧ *j*<*b* ∧ *i*≠*j* ⟶ *interfree-aux* (*c i*, *Q i*, *c j*))

⟹ *interfree* (*map* (λ*k*. (*c k*, *Q k*)) [*a*..<*b*])
by(*force simp add*: *interfree-def less-diff-conv*)

definition *map-ann-hoare* :: (('*a ann-com-op* ∗ '*a assn*) *list*) ⇒ *bool* (‹[⊢] -› [*0*] *45*) **where**
[⊢] *Ts* == (∀ *i*<*length Ts*. ∃ *c q*. *Ts*!*i*=(*Some c*, *q*) ∧ ⊢ *c q*)

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

lemma *MapAnnList*: ⟦ ⊢ *c q* ; [⊢] *xs* ⟧ ⟹ [⊢] (*Some c*,*q*)#*xs*
apply(*simp add*:*map-ann-hoare-def*)
apply *clarify*
apply(*case-tac i*,*simp*+)
done

lemma *MapAnnMap*:
∀ *k*. *i*≤*k* ∧ *k*<*j* ⟶ ⊢ (*c k*) (*Q k*) ⟹ [⊢] *map* (λ*k*. (*Some* (*c k*), *Q k*)) [*i*..<*j*]
apply(*simp add*: *map-ann-hoare-def less-diff-conv*)
done

lemma *ParallelRule*:⟦ [⊢] *Ts* ; *interfree Ts* ⟧
⟹ ‖− (⋂ *i*∈{*i*. *i*<*length Ts*}. *pre*(*the*(*com*(*Ts*!*i*))))
Parallel Ts
(⋂ *i*∈{*i*. *i*<*length Ts*}. *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*: −(*Collect b*) = {*x*. ¬(*b x*)}
by *fast*

lemma *list-length*: *length* []=*0* *length* (*x*#*xs*) = *Suc*(*length xs*)
by *simp-all*
lemma *list-lemmas*: *length* []=*0* *length* (*x*#*xs*) = *Suc*(*length xs*)
(*x*#*xs*) ! *0* = *x* (*x*#*xs*) ! *Suc n* = *xs* ! *n*
by *simp-all*
lemma *le-Suc-eq-insert*: {*i*. *i* <*Suc n*} = *insert n* {*i*. *i*< *n*}
by *auto*
lemmas *primrecdef-list* = *pre.simps assertions.simps atomics.simps atom-com.simps*
lemmas *my-simp-list* = *list-lemmas fst-conv snd-conv*
not-less0 refl le-Suc-eq-insert Suc-not-Zero Zero-not-Suc nat.inject
Collect-mem-eq ball-simps option.simps primrecdef-list
lemmas *ParallelConseq-list* = *INTER-eq Collect-conj-eq length-map length-upt length-append*

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

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

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

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

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

Tactic for the generation of the verification conditions

The tactic basically uses two subtactics:

HoareRuleTac is called at the level of parallel programs, it uses the ParallelTac to solve parallel composition of programs. This verification has two parts, namely, (1) all component programs are correct and (2) they are interference free. *HoareRuleTac* is also called at the level of atomic regions, i.e. ⟨ ⟩ 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 ‖− *?p c q*, in this case we have precond=False and the generated verification condition would have the form *?p* ⊆ . . . which can be solved by *rtac subset-refl*, if True we proceed to simplify it using the simplification tactics above.

ML ‹

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

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

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

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

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

```
            resolve-tac ctxt @{thms allI} i,
            resolve-tac ctxt @{thms impI} i,
            AnnHoareRuleTac ctxt i]])

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

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

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

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

```
          dest-atomics-Tac ctxt (i+1),
          dest-atomics-Tac ctxt i],
      dest-atomics-Tac ctxt i])

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

The final tactic is given the name *oghoare*:

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

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

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

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

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

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

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

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

method-setup *interfree-aux* = ‹
  Scan.succeed (*SIMPLE-METHOD′ o interfree-aux-tac*)›
  verification condition generator for interference freedom tests

Tactics useful for dealing with the generated verification conditions:

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

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

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

end

1.7 Concrete Syntax

theory *Quote-Antiquote* **imports** *Main* **begin**

syntax
  -quote :: ′*b* ⇒ (′*a* ⇒ ′*b*)   (‹(«-»)› [*0*] *1000*)
  -antiquote :: (′*a* ⇒ ′*b*) ⇒ ′*b*   (‹´-› [*1000*] *1000*)
  -Assert :: ′*a* ⇒ ′*a set*   (‹(⦃-⦄)› [*0*] *1000*)

translations
  ⦃*b*⦄ ⇀ *CONST Collect* «*b*»

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

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

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

abbreviation *Skip* :: *'a com* (‹*SKIP*› *63*)
  where *SKIP* ≡ *Basic id*

abbreviation *AnnSkip* :: *'a assn* ⇒ *'a ann-com* (‹*-//SKIP*› [*90*] *63*)
  where *r SKIP* ≡ *AnnBasic r id*

notation
  Seq (‹*-,,/ -*› [*55*, *56*] *55*) **and**
  AnnSeq (‹*-;;/ -*› [*60*,*61*] *60*)

syntax
  -Assign :: *idt* ⇒ *'b* ⇒ *'a com* (‹(*´- :=/ -*)› [*70*, *65*] *61*)
  -AnnAssign :: *'a assn* ⇒ *idt* ⇒ *'b* ⇒ *'a com* (‹(*- ´- :=/ -*)› [*90*,*70*,*65*] *61*)

translations
  ´x := a ⇀ *CONST Basic* «*´*(*-update-name x* (λ*-. a*))»
  r ´x := a ⇀ *CONST AnnBasic r* «*´*(*-update-name x* (λ*-. a*))»

syntax
  -AnnCond1 :: *'a assn* ⇒ *'a bexp* ⇒ *'a ann-com* ⇒ *'a ann-com* ⇒ *'a ann-com*
    (‹*- //IF - /THEN - /ELSE - /FI*› [*90*,*0*,*0*,*0*] *61*)
  -AnnCond2 :: *'a assn* ⇒ *'a bexp* ⇒ *'a ann-com* ⇒ *'a ann-com*
    (‹*- //IF - /THEN - /FI*› [*90*,*0*,*0*] *61*)
  -AnnWhile :: *'a assn* ⇒ *'a bexp* ⇒ *'a assn* ⇒ *'a ann-com* ⇒ *'a ann-com*
    (‹*- //WHILE - /INV - //DO -//OD*› [*90*,*0*,*0*,*0*] *61*)
  -AnnAwait :: *'a assn* ⇒ *'a bexp* ⇒ *'a com* ⇒ *'a ann-com*
    (‹*- //AWAIT - /THEN /- /END*› [*90*,*0*,*0*] *61*)
  -AnnAtom :: *'a assn* ⇒ *'a com* ⇒ *'a ann-com* (‹*-//*⟨*-*⟩› [*90*,*0*] *61*)
  -AnnWait :: *'a assn* ⇒ *'a bexp* ⇒ *'a ann-com* (‹*-//WAIT - END*› [*90*,*0*] *61*)

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

translations
  IF b THEN c1 ELSE c2 FI ⇀ *CONST Cond* ⦃*b*⦄ *c1 c2*
  IF b THEN c FI ⇌ *IF b THEN c ELSE SKIP FI*
  WHILE b INV i DO c OD ⇀ *CONST While* ⦃*b*⦄ *i c*
  WHILE b DO c OD ⇌ *WHILE b INV CONST undefined DO c OD*

  r IF b THEN c1 ELSE c2 FI ⇀ *CONST AnnCond1 r* ⦃*b*⦄ *c1 c2*
  r IF b THEN c FI ⇀ *CONST AnnCond2 r* ⦃*b*⦄ *c*
  r WHILE b INV i DO c OD ⇀ *CONST AnnWhile r* ⦃*b*⦄ *i c*
  r AWAIT b THEN c END ⇀ *CONST AnnAwait r* ⦃*b*⦄ *c*
  r ⟨*c*⟩ ⇌ *r AWAIT CONST True THEN c END*

```
  r WAIT b END ⇌ r AWAIT b THEN SKIP END

nonterminal prgs

syntax
  -PAR :: prgs ⇒ 'a              (‹COBEGIN//-//COEND› [57] 56)
  -prg :: ['a, 'a] ⇒ prgs        (‹-//-› [60, 90] 57)
  -prgs :: ['a, 'a, prgs] ⇒ prgs  (‹-//-//‖//-› [60,90,57] 57)

  -prg-scheme :: ['a, 'a, 'a, 'a, 'a] ⇒ prgs
              (‹SCHEME [- ≤ - < -] -// -› [0,0,0,60, 90] 57)

translations
  -prg c q ⇌ [(CONST Some c, q)]
  -prgs c q ps ⇌ (CONST Some c, q) # ps
  -PAR ps ⇌ CONST Parallel ps

  -prg-scheme j i k c q ⇌ CONST map (λi. (CONST Some c, q)) [j..<k]

print-translation ‹
  let
    fun quote-tr' f (t :: ts) =
          Term.list-comb (f $ Syntax-Trans.quote-tr' syntax-const ‹-antiquote› t,
ts)
      | quote-tr' - - = raise Match;

    fun annquote-tr' f (r :: t :: ts) =
          Term.list-comb (f $ r $ Syntax-Trans.quote-tr' syntax-const ‹-antiquote›
t, ts)
      | annquote-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 annbexp-tr' name (r :: (Const (const-syntax ‹Collect›, -) $ t) :: ts) =
          annquote-tr' (Syntax.const name) (r :: t :: ts)
      | annbexp-tr' - - = raise Match;

    fun assign-tr' (Abs (x, -, f $ k $ Bound 0) :: ts) =
        quote-tr' (Syntax.const syntax-const ‹-Assign› $ Syntax-Trans.update-name-tr'
f)
          (Abs (x, dummyT, Syntax-Trans.const-abs-tr' k) :: ts)
      | assign-tr' - = raise Match;

    fun annassign-tr' (r :: Abs (x, -, f $ k $ Bound 0) :: ts) =
        quote-tr' (Syntax.const syntax-const ‹-AnnAssign› $ r $ Syntax-Trans.update-name-tr'
```

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

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

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

end

1.8 Examples

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

1.8.1 Mutual Exclusion

Peterson's Algorithm I

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

record *Petersons-mutex-1* =
 pr1 :: *nat*
 pr2 :: *nat*
 in1 :: *bool*
 in2 :: *bool*
 hold :: *nat*

lemma *Petersons-mutex-1*:
 ‖− ⦃´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 {|¬´flag1|}
        WHILE True
        INV {|¬´flag1|}
        DO {|¬´flag1|} ⟨ ´flag1:=True,,´after1:=False ⟩;;
           {|´flag1 ∧ ¬´after1|} ⟨ ´turn:=1,,´after1:=True ⟩;;
           {|´flag1 ∧ ´after1 ∧ (´turn=1 ∨ ´turn=2)|}
            WHILE ¬(´flag2 ⟶ ´turn=2)
```

```
        INV ⦃´flag1 ∧ ´after1 ∧ (´turn=1 ∨ ´turn=2)⦄
        DO ⦃´flag1 ∧ ´after1 ∧ (´turn=1 ∨ ´turn=2)⦄ SKIP OD;;
       ⦃´flag1 ∧ ´after1 ∧ (´flag2 ∧ ´after2 ⟶ ´turn=2)⦄
        ´flag1:=False
     OD
    ⦃False⦄
 ∥
   ⦃¬´flag2⦄
     WHILE True
     INV ⦃¬´flag2⦄
     DO ⦃¬´flag2⦄ ⟨ ´flag2:=True,,´after2:=False ⟩;;
       ⦃´flag2 ∧ ¬´after2⦄ ⟨ ´turn:=2,,´after2:=True ⟩;;
       ⦃´flag2 ∧ ´after2 ∧ (´turn=1 ∨ ´turn=2)⦄
        WHILE ¬(´flag1 ⟶ ´turn=1)
        INV ⦃´flag2 ∧ ´after2 ∧ (´turn=1 ∨ ´turn=2)⦄
        DO ⦃´flag2 ∧ ´after2 ∧ (´turn=1 ∨ ´turn=2)⦄ SKIP OD;;
       ⦃´flag2 ∧ ´after2 ∧ (´flag1 ∧ ´after1 ⟶ ´turn=1)⦄
        ´flag2:=False
     OD
    ⦃False⦄
 COEND
 ⦃False⦄
```

apply *oghoare*
— 122 vc
apply *auto*
done

Peterson's Algorithm III: A Solution using Semaphores

record *Semaphores-mutex* =
out :: *bool*
who :: *nat*

lemma *Semaphores-mutex*:

```
 ∥− ⦃i≠j⦄
  ´out:=True ,,
  COBEGIN ⦃i≠j⦄
      WHILE True INV ⦃i≠j⦄
      DO ⦃i≠j⦄ AWAIT ´out THEN  ´out:=False,, ´who:=i END;;
         ⦃¬´out ∧ ´who=i ∧ i≠j⦄ ´out:=True OD
      ⦃False⦄
  ∥
      ⦃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!i=´nextv END;;
       ⦃´I ∧ ´turn!i=´nextv⦄ ´nextv:=´nextv+1
     OD
   ⦃False⦄
 COEND

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

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

1.8.2 Parallel Zero Search

Synchronized Zero Search. Zero-6

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

record *Zero-search* =
turn :: *nat*
found :: *bool*
x :: *nat*
y :: *nat*

lemma *Zero-search*:
$[\![I1 = « \ a \leq ´x \wedge (´found \longrightarrow (a < ´x \wedge f(´x)=0) \vee (´y \leq a \wedge f(´y)=0))$
$\wedge (\neg ´found \wedge a < ´\ x \longrightarrow f(´x) \neq 0) » ;$
$I2 = « ´y \leq a+1 \wedge (´found \longrightarrow (a < ´x \wedge f(´x)=0) \vee (´y \leq a \wedge f(´y)=0))$
$\wedge (\neg ´found \wedge ´y \leq a \longrightarrow f(´y) \neq 0) »]\!] \Longrightarrow$
$\Vert- \{\!| \exists \ u. \ f(u)=0 |\!\}$
$´turn:=1,, \ ´found:= False,,$
$´x:=a,, \ ´y:=a+1 \ ,,$
$COBEGIN \ \{\!| ´I1 |\!\}$

WHILE ¬´*found*
INV {|´*I1*|}
DO {|*a*≤´*x* ∧ (´*found* ⟶ ´*y*≤*a* ∧ *f*(´*y*)=*0*) ∧ (*a*<´*x* ⟶ *f*(´*x*)≠*0*)|}
WAIT ´*turn*=*1* *END*;;
{|*a*≤´*x* ∧ (´*found* ⟶ ´*y*≤*a* ∧ *f*(´*y*)=*0*) ∧ (*a*<´*x* ⟶ *f*(´*x*)≠*0*)|}
´*turn*:=*2*;;
{|*a*≤´*x* ∧ (´*found* ⟶ ´*y*≤*a* ∧ *f*(´*y*)=*0*) ∧ (*a*<´*x* ⟶ *f*(´*x*)≠*0*)|}
⟨ ´*x*:=´*x*+*1*,,
IF *f*(´*x*)=*0* *THEN* ´*found*:=*True* *ELSE* *SKIP* *FI*⟩
OD;;
{|´*I1* ∧ ´*found*|}
´*turn*:=*2*
{|´*I1* ∧ ´*found*|}
‖
{|´*I2*|}
WHILE ¬´*found*
INV {|´*I2*|}
DO {|´*y*≤*a*+*1* ∧ (´*found* ⟶ *a*<´*x* ∧ *f*(´*x*)=*0*) ∧ (´*y*≤*a* ⟶ *f*(´*y*)≠*0*)|}
WAIT ´*turn*=*2* *END*;;
{|´*y*≤*a*+*1* ∧ (´*found* ⟶ *a*<´*x* ∧ *f*(´*x*)=*0*) ∧ (´*y*≤*a* ⟶ *f*(´*y*)≠*0*)|}
´*turn*:=*1*;;
{|´*y*≤*a*+*1* ∧ (´*found* ⟶ *a*<´*x* ∧ *f*(´*x*)=*0*) ∧ (´*y*≤*a* ⟶ *f*(´*y*)≠*0*)|}
⟨ ´*y*:=(´*y* − *1*),,
IF *f*(´*y*)=*0* *THEN* ´*found*:=*True* *ELSE* *SKIP* *FI*⟩
OD;;
{|´*I2* ∧ ´*found*|}
´*turn*:=*1*
{|´*I2* ∧ ´*found*|}
COEND
{|*f*(´*x*)=*0* ∨ *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*:
⟦*I1*=« *a*≤´*x* ∧ (´*found* ⟶ (*a*<´*x* ∧ *f*(´*x*)=*0*) ∨ (´*y*≤*a* ∧ *f*(´*y*)=*0*))
∧ (¬´*found* ∧ *a*<´*x* ⟶ *f*(´*x*)≠*0*)»;
I2= «´*y*≤*a*+*1* ∧ (´*found* ⟶ (*a*<´*x* ∧ *f*(´*x*)=*0*) ∨ (´*y*≤*a* ∧ *f*(´*y*)=*0*))
∧ (¬´*found* ∧ ´*y*≤*a* ⟶ *f*(´*y*)≠*0*)»⟧ ⟹
‖− {|∃ *u*. *f*(*u*)=*0*|}
´*found*:= *False*,,
´*x*:=*a*,, ´*y*:=*a*+*1*,,
COBEGIN {|´*I1*|}
WHILE ¬´*found*
INV {|´*I1*|}
DO {|*a*≤´*x* ∧ (´*found* ⟶ ´*y*≤*a* ∧ *f*(´*y*)=*0*) ∧ (*a*<´*x* ⟶ *f*(´*x*)≠*0*)|}

    ⟨ ´*x*:=´*x*+*1*,,*IF f*(´*x*)=*0 THEN* ´*found*:=*True ELSE SKIP FI*⟩
   OD
   ⦃´*I1* ∧ ´*found*⦄
  ‖
   ⦃´*I2*⦄
   WHILE ¬´*found*
   INV ⦃´*I2*⦄
   DO ⦃´*y*≤*a*+*1* ∧ (´*found* ⟶ *a*<´*x* ∧ *f*(´*x*)=*0*) ∧ (´*y*≤*a* ⟶ *f*(´*y*)≠*0*)⦄
    ⟨ ´*y*:=(´*y* − *1*),,*IF f*(´*y*)=*0 THEN* ´*found*:=*True ELSE SKIP FI*⟩
   OD
   ⦃´*I2* ∧ ´*found*⦄
  COEND
  ⦃*f*(´*x*)=*0* ∨ *f*(´*y*)=*0*⦄
apply *oghoare*
— 20 vc
apply *auto*
— auto takes aprox. 2 minutes.
done

1.8.3 Producer/Consumer

Previous lemmas

lemma *nat-lemma2*: ⟦ *b* = *m*∗(*n*::*nat*) + *t*; *a* = *s*∗*n* + *u*; *t*=*u*; *b*−*a* < *n* ⟧ ⟹ *m* ≤ *s*
proof −
 assume *b* = *m*∗(*n*::*nat*) + *t a* = *s*∗*n* + *u t*=*u*
 hence (*m* − *s*) ∗ *n* = *b* − *a* **by** (*simp add*: *diff-mult-distrib*)
 also assume ... < *n*
 finally have *m* − *s* < *1* **by** *simp*
 thus *?thesis* **by** *arith*
qed

lemma *mod-lemma*: ⟦ (*c*::*nat*) ≤ *a*; *a* < *b*; *b* − *c* < *n* ⟧ ⟹ *b mod n* ≠ *a mod n*
apply(*subgoal-tac b*=*b div n*∗*n* + *b mod n*)
 prefer *2* **apply** (*simp add*: *div-mult-mod-eq* [*symmetric*])
apply(*subgoal-tac a*=*a div n*∗*n* + *a mod n*)
 prefer *2*
 apply(*simp add*: *div-mult-mod-eq* [*symmetric*])
apply(*subgoal-tac b* − *a* ≤ *b* − *c*)
 prefer *2* **apply** *arith*
apply(*drule le-less-trans*)
back
 apply *assumption*
apply(*frule less-not-refl2*)
apply(*drule less-imp-le*)
apply (*drule-tac m* = *a* **and** *k* = *n* **in** *div-le-mono*)
apply(*safe*)
apply(*frule-tac b* = *b* **and** *a* = *a* **and** *n* = *n* **in** *nat-lemma2*, *assumption*, *assumption*)

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

Producer/Consumer Algorithm

record *Producer-consumer* =
  ins :: *nat*
  outs :: *nat*
  li :: *nat*
  lj :: *nat*
  vx :: *nat*
  vy :: *nat*
  buffer :: *nat list*
  b :: *nat list*

The whole proof takes aprox. 4 minutes.

lemma *Producer-consumer*:
```
  ⟦INIT= «0<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* ***context***)›)
— 112 subgoals left
apply(*simp-all* (*no-asm*))
— 43 subgoals left
apply(*tactic* ‹*ALLGOALS* (*conjI-Tac* ***context*** (*K all-tac*))›)
— 419 subgoals left
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
— 99 subgoals left
apply(*simp-all only*:*length-0-conv* [*THEN sym*])
— 20 subgoals left
apply (*simp-all del*:*length-0-conv length-greater-0-conv add*: *nth-list-update mod-lemma*)
— 9 subgoals left
apply (*force simp add*:*less-Suc-eq*)
apply(*hypsubst-thin*, *drule sym*)
apply (*force simp add*:*less-Suc-eq*)+
done

1.8.4 Parameterized Examples

Set Elements of an Array to Zero

record *Example1* =
a :: *nat* ⇒ *nat*

lemma *Example1*:
‖− {|*True*|}
COBEGIN SCHEME [*0≤i<n*] {|*True*|} *´a*:=*´a* (*i*:=*0*) {|*´a i*=*0*|} *COEND*
{|∀ *i* < *n*. *´a i* = *0*|}
apply *oghoare*
apply *simp-all*
done

Same example with lists as auxiliary variables.

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

Increment a Variable in Parallel

First some lemmas about summation properties.

```
lemma Example2-lemma2-aux: !!b. j<n ⟹
 (∑ i=0..<n. (b i::nat)) =
 (∑ i=0..<j. b i) + b j + (∑ i=0..<n−(Suc j) . b (Suc j + i))
apply(induct n)
 apply simp-all
apply(simp add:less-Suc-eq)
 apply(auto)
apply(subgoal-tac n − j = Suc(n− Suc j))
  apply simp
apply arith
done

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

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

apply(*erule ssubst*)
apply *simp-all*
done

record *Example2* =
 c :: *nat* ⇒ *nat*
 x :: *nat*

lemma *Example-2*: $0<n \Longrightarrow$
 $\Vert-$ {| ´*x*=*0* ∧ (∑ *i*=*0*..<*n*. ´*c i*)=*0* |}
 COBEGIN
   SCHEME [*0*≤*i*<*n*]
  {| ´*x*=(∑ *i*=*0*..<*n*. ´*c i*) ∧ ´*c i*=*0* |}
   ⟨ ´*x*:=´*x*+(*Suc 0*),, ´*c*:=´*c* (*i*:=(*Suc 0*)) ⟩
  {| ´*x*=(∑ *i*=*0*..<*n*. ´*c i*) ∧ ´*c i*=(*Suc 0*) |}
 COEND
 {| ´*x*=*n* |}
apply *oghoare*
apply (*simp-all cong del*: *sum.cong-simp*)
apply (*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
apply (*simp-all cong del*: *sum.cong-simp*)
   apply(*erule* (*1*) *Example2-lemma2*)
  apply(*erule* (*1*) *Example2-lemma2*)
 apply(*erule* (*1*) *Example2-lemma2*)
apply(*simp*)
done

end

Chapter 2

Case Study: Single and Multi-Mutator Garbage Collection Algorithms

2.1 Formalization of the Memory

theory *Graph* **imports** *Main* **begin**

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

type-synonym *nodes* = *node list*
type-synonym *edge* = *nat* × *nat*
type-synonym *edges* = *edge list*

consts *Roots* :: *nat set*

definition *Proper-Roots* :: *nodes* ⇒ *bool* **where**
Proper-Roots M ≡ *Roots*≠{} ∧ *Roots* ⊆ {*i*. *i*<*length M*}

definition *Proper-Edges* :: (*nodes* × *edges*) ⇒ *bool* **where**
Proper-Edges ≡ (λ(*M*,*E*). ∀ *i*<*length E*. *fst*(*E*!*i*)<*length M* ∧ *snd*(*E*!*i*)<*length M*)

definition *BtoW* :: (*edge* × *nodes*) ⇒ *bool* **where**
BtoW ≡ (λ(*e*,*M*). (*M*!*fst e*)=*Black* ∧ (*M*!*snd e*)≠*Black*)

definition *Blacks* :: *nodes* ⇒ *nat set* **where**
Blacks M ≡ {*i*. *i*<*length M* ∧ *M*!*i*=*Black*}

definition *Reach* :: *edges* ⇒ *nat set* **where**
Reach E ≡ {*x*. (∃ *path*. *1*<*length path* ∧ *path*!(*length path* − *1*)∈*Roots* ∧ *x*=*path*!*0*
∧ (∀ *i*<*length path* − *1*. (∃ *j*<*length E*. *E*!*j*=(*path*!(*i*+*1*), *path*!*i*))))
∨ *x*∈*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*]:
  $\llbracket$ *Roots*$\subseteq$*Blacks M*; $\forall$ *i*<*length E*. $\neg$*BtoW*(*E*!*i*,*M*)$\rrbracket$
  $\Longrightarrow$ *1*< *length path* $\longrightarrow$ (*path*!(*length path* $-$ *1*))$\in$*Roots* $\longrightarrow$
  ($\forall$ *i*<*length path* $-$ *1*. ($\exists$ *j*. *j* < *length E* $\wedge$ *E*!*j*=(*path*!(*Suc i*), *path*!*i*)))
  $\longrightarrow$ *M*!(*path*!*0*) = *Black*
apply(*induct-tac path*)
 apply *force*
apply *clarify*
apply *simp*
apply(*case-tac list*)
 apply *force*
apply *simp*
apply(*rename-tac lista*)
apply(*rotate-tac* $-$*2*)
apply(*erule-tac x* = *0* **in** *all-dupE*)
apply *simp*
apply *clarify*
apply(*erule allE* , *erule* (*1*) *notE impE*)
apply *simp*
apply(*erule mp*)
apply(*case-tac lista*)
 apply *force*
apply *simp*
apply(*erule mp*)
apply *clarify*
apply(*erule-tac x* = *Suc i* **in** *allE*)
apply *force*
done

lemma *Graph1*:
  $\llbracket$*Roots*$\subseteq$*Blacks M*; *Proper-Edges*(*M*, *E*); $\forall$ *i*<*length E*. $\neg$*BtoW*(*E*!*i*,*M*) $\rrbracket$
  $\Longrightarrow$ *Reach E*$\subseteq$*Blacks M*
apply (*unfold Reach-def*)
apply *simp*
apply *clarify*
apply(*erule disjE*)
 apply *clarify*

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

Graph 2

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

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

lemma *Ex-last-occurrence*:
$[\![P\ (n::nat);\ n\leq l]\!] \Longrightarrow (\exists\, m.\ P\ (l - m) \wedge (\forall\, i.\ i<m \longrightarrow \neg P\ (l - i)))$
apply(*drule Compl-lemma*)
apply *clarify*
apply(*erule Ex-first-occurrence*)
done

lemma *Graph2*:
$[\![T \in Reach\ E;\ R<length\ E]\!] \Longrightarrow T \in Reach\ (E[R:=(fst(E!R),\ T)])$
apply (*unfold Reach-def*)
apply *clarify*
apply *simp*
apply(*case-tac* $\forall\, z<length\ path.\ fst(E!R)\neq path!z$)
apply(*rule-tac x = 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 = Suc i* **in** *allE*)
apply *simp*
apply (*force simp add:nth-list-update*)
apply *simp*
apply(*erule exE*)

apply(*subgoal-tac z* ≤ *length path* − *Suc 0*)
 prefer *2* **apply** *arith*
apply(*drule-tac P* = λ*m*. *m*<*length path* ∧ *fst*(*E*!*R*)=*path*!*m* **in** *Ex-last-occurrence*)
 apply *assumption*
apply *clarify*
apply *simp*
apply(*rule-tac x* = (*path*!*0*)#(*drop* (*length path* − *Suc m*) *path*) **in** *exI*)
apply *simp*
apply(*case-tac length path* − (*length path* − *Suc m*))
 apply *arith*
apply *simp*
apply(*subgoal-tac* (*length path* − *Suc m*) + *nat* ≤ *length path*)
 prefer *2* **apply** *arith*
apply(*subgoal-tac length path* − *Suc m* + *nat* = *length path* − *Suc 0*)
 prefer *2* **apply** *arith*
apply *clarify*
apply(*case-tac i*)
 apply(*force simp add*: *nth-list-update*)
apply *simp*
apply(*subgoal-tac* (*length path* − *Suc m*) + *nata* ≤ *length path*)
 prefer *2* **apply** *arith*
apply(*subgoal-tac* (*length path* − *Suc m*) + (*Suc nata*) ≤ *length path*)
 prefer *2* **apply** *arith*
apply *simp*
apply(*erule-tac x* = *length path* − *Suc m* + *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* = *path* ! (*length path* − *Suc m*) **in** *sym*)
apply *simp*
apply(*case-tac length path* − *Suc 0* < *m*)
 apply(*subgoal-tac* (*length path* − *Suc m*)=*0*)
  prefer *2* **apply** *arith*
 apply(*simp del*: *diff-is-0-eq*)
 apply(*subgoal-tac Suc nata*≤*nat*)
 prefer *2* **apply** *arith*
 apply(*drule-tac n* = *Suc nata* **in** *Compl-lemma*)
 apply *clarify*
 subgoal using [[*linarith-split-limit* = *0*]] **by** *force*
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.absorb1* [*simp*] *min.absorb2* [*simp*]

lemma *Graph3*:
$[\![\ T{\in}Reach\ E;\ R{<}length\ E\ ]\!] \Longrightarrow Reach(E[R{:=}(fst(E!R),T)]) \subseteq Reach\ E$
apply (*unfold Reach-def*)
apply *clarify*
apply *simp*
apply(*case-tac* $\exists\, i{<}length\ path - 1.\ (fst(E!R),T){=}(path!(Suc\ i),path!i)$)
— the changed edge is part of the path
apply(*erule exE*)
apply(*drule-tac* $P = \lambda i.\ i{<}length\ path - 1 \wedge (fst(E!R),T){=}(path!Suc\ i,path!i)$
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* $\neg(length\ path{\le}m)$)
prefer *2* **apply** *arith*
apply(*simp*)
apply(*rule conjI*)
apply(*subgoal-tac* $\neg(m + length\ patha - 1 < m)$)
prefer *2* **apply** *arith*
apply(*simp add*: *nth-append*)
apply(*rule conjI*)
apply(*case-tac m*)
apply *force*
apply(*case-tac path*)
apply *force*
apply *force*
apply *clarify*
apply(*case-tac* $Suc\ i{\le}m$)
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*:
⟦*T* ∈ *Reach E*; *Roots*⊆*Blacks M*; *I*≤*length E*; *T*<*length M*; *R*<*length E*;
∀ *i*<*I*. ¬*BtoW*(*E*!*i*,*M*); *R*<*I*; *M*!*fst*(*E*!*R*)=*Black*; *M*!*T*≠*Black*⟧ ⟹
(∃ *r*. *I*≤*r* ∧ *r*<*length E* ∧ *BtoW*(*E*[*R*:=(*fst*(*E*!*R*),*T*)]!*r*,*M*))
apply (*unfold Reach-def*)
apply *simp*
apply(*erule disjE*)
prefer *2* **apply** *force*

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

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

Graph 5

lemma *Graph5*:
⟦ *T* ∈ *Reach E* ; *Roots* ⊆ *Blacks M*; ∀ *i*<*R*. ¬*BtoW*(*E*!*i*,*M*); *T*<*length M*;
R<*length E*; *M*!*fst*(*E*!*R*)=*Black*; *M*!*snd*(*E*!*R*)=*Black*; *M*!*T* ≠ *Black*⟧
⟹ (∃ *r*. *R*<*r* ∧ *r*<*length E* ∧ *BtoW*(*E*[*R*:=(*fst*(*E*!*R*),*T*)]!*r*,*M*))
apply (*unfold Reach-def*)
apply *simp*
apply(*erule disjE*)
prefer *2* **apply** *force*
apply *clarify*
— there exist a black node in the path to T
apply(*case-tac* $\exists$ *m*<*length path. M*!(*path*!*m*)=*Black*)
apply(*erule exE*)
apply(*drule-tac P* = λ*m. m*<*length path* $\wedge$ *M*!(*path*!*m*)=*Black* **in** *Ex-first-occurrence*)
apply *clarify*
apply(*case-tac ma*)

apply *force*
apply *simp*
apply(*case-tac length path*)
apply *force*
apply *simp*
apply(*erule-tac* $P = \lambda i.\ i < nata \longrightarrow P\ i$ **and** $x = nat$ **for** P **in** *allE*)
apply *simp*
apply *clarify*
apply(*erule-tac* $P = \lambda i.\ i < Suc\ nat \longrightarrow P\ i$ **and** $x = nat$ **for** P **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* $x = length\ path - 1$ **in** *allE*)
apply(*case-tac length path*)
apply *force*
apply *force*
done

Other lemmas about graphs

lemma *Graph6*:
$[\![Proper\text{-}Edges(M,E);\ R<length\ E\ ;\ T<length\ M]\!] \Longrightarrow Proper\text{-}Edges(M,E[R:=(fst(E!R),T)])$
apply (*unfold Proper-Edges-def*)
apply(*force simp add*: *nth-list-update*)
done

lemma *Graph7*:
$[\![Proper\text{-}Edges(M,E)]\!] \Longrightarrow Proper\text{-}Edges(M[T:=a],E)$
apply (*unfold Proper-Edges-def*)
apply *force*
done

lemma *Graph8*:
$[\![Proper\text{-}Roots(M)]\!] \Longrightarrow Proper\text{-}Roots(M[T:=a])$
apply (*unfold Proper-Roots-def*)
apply *force*
done

Some specific lemmata for the verification of garbage collection algorithms.

lemma *Graph9*: $j<length\ M \Longrightarrow Blacks\ M \subseteq Blacks\ (M[j := Black])$
apply (*unfold Blacks-def*)

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

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

lemma *Graph11* [*rule-format* (*no-asm*)]:
⟦ *M*!*j*≠*Black*;*j*<*length M*⟧ ⟹ *Blacks M* ⊂ *Blacks* (*M*[*j* := *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*: ⟦*a*⊆*Blacks M*;*j*<*length M*⟧ ⟹ *a*⊆*Blacks* (*M*[*j* := *Black*])
apply (*unfold Blacks-def*)
apply(*force simp add*: *nth-list-update*)
done

lemma *Graph13*: ⟦*a*⊂ *Blacks M*;*j*<*length M*⟧ ⟹ *a* ⊂ *Blacks* (*M*[*j* := *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* ⇒ *bool* **where**
Mut-init ≡ « *T* ∈ *Reach* ´*E* ∧ *R* < *length* ´*E* ∧ *T* < *length* ´*M* »

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 ≡ {|´*Mut-init* ∧ ´*z*|} ⟨´*E*:=´*E*[*R*:=(*fst*(´*E*!*R*), *T*)],, ´*z*:= (¬´*z*)⟩

definition *Color-Target* :: *gar-coll-state ann-com* **where**
Color-Target ≡ {|´*Mut-init* ∧ ¬´*z*|} ⟨´*M*:=´*M*[*T*:=*Black*],, ´*z*:= (¬´*z*)⟩

definition *Mutator* :: *gar-coll-state ann-com* **where**
Mutator ≡
{|´*Mut-init* ∧ ´*z*|}
WHILE True INV {|´*Mut-init* ∧ ´*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*:
⊢ *Redirect-Edge pre*(*Color-Target*)
apply (*unfold mutator-defs*)
apply *annhoare*
apply(*simp-all*)
apply(*force elim*:*Graph2*)
done

lemma *Color-Target*:
⊢ *Color-Target* {|´*Mut-init* ∧ ´*z*|}
apply (*unfold mutator-defs*)
apply *annhoare*

apply(*simp-all*)
done

lemma *Mutator*:
⊢ *Mutator* {|*False*|}
apply(*unfold Mutator-def*)
apply *annhoare*
apply(*simp-all add:Redirect-Edge Color-Target*)
apply(*simp add:mutator-defs*)
done

2.2.2 The Collector

A constant *M-init* is used to give *´Ma* a suitable first value, defined as a list of nodes where only the *Roots* are black.

consts *M-init* :: *nodes*

definition *Proper-M-init* :: *gar-coll-state* ⇒ *bool* **where**
Proper-M-init ≡ « *Blacks M-init=Roots* ∧ *length M-init=length ´M* »

definition *Proper* :: *gar-coll-state* ⇒ *bool* **where**
Proper ≡ « *Proper-Roots ´M* ∧ *Proper-Edges*(*´M*, *´E*) ∧ *´Proper-M-init* »

definition *Safe* :: *gar-coll-state* ⇒ *bool* **where**
Safe ≡ « *Reach ´E* ⊆ *Blacks ´M* »

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

Blackening the roots

definition *Blacken-Roots* :: *gar-coll-state ann-com* **where**
Blacken-Roots ≡
{|*´Proper*|}
´ind:=*0*;;
{|*´Proper* ∧ *´ind=0*|}
WHILE ´ind<length ´M
INV {|*´Proper* ∧ (∀ *i<´ind*. *i* ∈ *Roots* ⟶ *´M*!*i=Black*) ∧ *´ind≤length ´M*|}
DO {|*´Proper* ∧ (∀ *i<´ind*. *i* ∈ *Roots* ⟶ *´M*!*i=Black*) ∧ *´ind<length ´M*|}
IF ´ind∈*Roots THEN*
{|*´Proper* ∧ (∀ *i<´ind*. *i* ∈ *Roots* ⟶ *´M*!*i=Black*) ∧ *´ind<length ´M* ∧ *´ind*∈*Roots*|}
´M:=*´M*[*´ind*:=*Black*] *FI*;;
{|*´Proper* ∧ (∀ *i<´ind+1*. *i* ∈ *Roots* ⟶ *´M*!*i=Black*) ∧ *´ind<length ´M*|}
´ind:=*´ind+1*
OD

lemma *Blacken-Roots*:
⊢ *Blacken-Roots* {|*´Proper* ∧ *Roots*⊆*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* ⇒ *nat* ⇒ *bool* **where**
 PBInv ≡ « λ*ind*. ´*obc* < *Blacks* ´*M* ∨ (∀ *i* <*ind*. ¬*BtoW* (´*E*!*i*, ´*M*) ∨
 (¬´*z* ∧ *i*=*R* ∧ (*snd*(´*E*!*R*)) = *T* ∧ (∃ *r*. *ind* ≤ *r* ∧ *r* < *length* ´*E* ∧ *BtoW*(´*E*!*r*,´*M*))))»

definition *Propagate-Black-aux* :: *gar-coll-state ann-com* **where**
 Propagate-Black-aux ≡
 {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*|}
 ´*ind*:=*0*;;
 {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M* ∧ ´*ind*=*0*|}
 WHILE ´*ind*<*length* ´*E*
  INV {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
   ∧ ´*PBInv* ´*ind* ∧ ´*ind*≤*length* ´*E*|}
 DO {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
   ∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E*|}
  IF ´*M*!(*fst* (´*E*!´*ind*)) = *Black THEN*
   {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
    ∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E* ∧ ´*M*!*fst*(´*E*!´*ind*)=*Black*|}
   ´*M*:=´*M*[*snd*(´*E*!´*ind*):=*Black*];;
   {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
    ∧ ´*PBInv* (´*ind* + *1*) ∧ ´*ind*<*length* ´*E*|}
   ´*ind*:=´*ind*+*1*
  FI
 OD

lemma *Propagate-Black-aux*:
 ⊢ *Propagate-Black-aux*
 {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
  ∧ (´*obc* < *Blacks* ´*M* ∨ ´*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* ⇒ *bool* **where**
  Auxk ≡ «´*k*<*length* ´*M* ∧ (´*M*!´*k*≠*Black* ∨ ¬*BtoW*(´*E*!´*ind*, ´*M*) ∨
        ´*obc*<*Blacks* ´*M* ∨ (¬´*z* ∧ ´*ind*=*R* ∧ *snd*(´*E*!*R*)=*T*
        ∧ (∃ *r*. ´*ind*<*r* ∧ *r*<*length* ´*E* ∧ *BtoW*(´*E*!*r*, ´*M*))))»

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

{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*|}
´*ind*:=*0*;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M* ∧ ´*ind*=*0*|}
WHILE ´*ind*<*length* ´*E*
INV {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ ´*PBInv* ´*ind* ∧ ´*ind*≤*length* ´*E*|}
DO {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E*|}
IF (´*M*!(*fst* (´*E*!´*ind*)))=*Black* *THEN*
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E* ∧ (´*M*!*fst*(´*E*!´*ind*))=*Black*|}
´*k*:=(*snd*(´*E*!´*ind*));;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E* ∧ (´*M*!*fst*(´*E*!´*ind*))=*Black*
∧ ´*Auxk*|}
⟨´*M*:=´*M*[´*k*:=*Black*],, ´*ind*:=´*ind*+*1*⟩
ELSE {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ ´*PBInv* ´*ind* ∧ ´*ind*<*length* ´*E*|}
⟨*IF* (´*M*!(*fst* (´*E*!´*ind*)))≠*Black* *THEN* ´*ind*:=´*ind*+*1* *FI*⟩
FI
OD

lemma *Propagate-Black*:
⊢ *Propagate-Black*
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ (´*obc* < *Blacks* ´*M* ∨ ´*Safe*)|}
apply (*unfold Propagate-Black-def PBInv-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* = *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* ⇒ *nat* ⇒ *bool* **where**
CountInv ≡ « λ*ind*. {*i*. *i*<*ind* ∧ ´*Ma*!*i*=*Black*}⊆´*bc* »

definition *Count* :: *gar-coll-state ann-com* **where**
Count ≡
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*bc*={}|}
´*ind*:=*0*;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*bc*={}
∧ ´*ind*=*0*|}
WHILE ´*ind*<*length* ´*M*
INV {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ ´*CountInv* ´*ind*
∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*ind*≤*length* ´*M*|}
DO {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ ´*CountInv* ´*ind*
∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*ind*<*length* ´*M*|}
IF ´*M*!´*ind*=*Black*
THEN {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ ´*CountInv* ´*ind*
∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*ind*<*length* ´*M* ∧ ´*M*!´*ind*=*Black*|}
´*bc*:=*insert* ´*ind* ´*bc*
FI;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ ´*CountInv* (´*ind*+*1*)
∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*) ∧ ´*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

axiomatization *Append-to-free* :: *nat* × *edges* ⇒ *edges*
where
Append-to-free0: *length* (*Append-to-free* (*i*, *e*)) = *length e* **and**
Append-to-free1: *Proper-Edges* (*m*, *e*)
⟹ *Proper-Edges* (*m*, *Append-to-free*(*i*, *e*)) **and**
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 ≡ «λ*ind*. ∀ *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* ∧ ´*AppendInv* ´*ind* ∧ ´*ind*<*length* ´*M* ∧ ´*ind*∉*Reach* ´*E*⦄
´*E*:=*Append-to-free*(´*ind*,´*E*)
FI;;
⦃´*Proper* ∧ ´*AppendInv* (´*ind*+*1*) ∧ ´*ind*<*length* ´*M*⦄
´*ind*:=´*ind*+*1*
OD

lemma *Append*:

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

Correctness of the Collector

definition *Collector* :: *gar-coll-state ann-com* **where**
Collector ≡
{|´*Proper*|}
WHILE True INV {|´*Proper*|}
DO
Blacken-Roots;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*|}
´*obc*:={};;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*={}|}
´*bc*:=*Roots*;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*={} ∧ ´*bc*=*Roots*|}
´*Ma*:=*M-init*;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*={} ∧ ´*bc*=*Roots* ∧ ´*Ma*=*M-init*|}
WHILE ´*obc*≠´*bc*
INV {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M*
∧ ´*obc*⊆*Blacks* ´*Ma* ∧ *Blacks* ´*Ma*⊆´*bc* ∧ ´*bc*⊆*Blacks* ´*M*
∧ *length* ´*Ma*=*length* ´*M* ∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*)|}
DO {|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*|}
´*obc*:=´*bc*;;
Propagate-Black;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
∧ (´*obc* < *Blacks* ´*M* ∨ ´*Safe*)|}
´*Ma*:=´*M*;;
{|´*Proper* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*Ma*
∧ *Blacks* ´*Ma*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M* ∧ *length* ´*Ma*=*length* ´*M*
∧ (´*obc* < *Blacks* ´*Ma* ∨ ´*Safe*)|}

´*bc*:={};;
Count
OD;;
Append
OD

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

2.2.3 Interference Freedom

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Interference freedom Collector-Mutator

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

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

Interference freedom Mutator-Collector

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

The Garbage Collection algorithm

In total there are 289 verification conditions.

lemma *Gar-Coll*:
‖− {|´*Proper* ∧ ´*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-mut-init* :: *mul-gar-coll-state* ⇒ *nat* ⇒ *bool* **where**
  Mul-mut-init ≡ « λ*n*. *n*=*length* ´*Muts* ∧ (∀ *i*<*n*. *R* (´*Muts*!*i*)<*length* ´*E*
    ∧ *T* (´*Muts*!*i*)<*length* ´*M*) »

definition *Mul-Redirect-Edge* :: *nat* ⇒ *nat* ⇒ *mul-gar-coll-state ann-com* **where**
  Mul-Redirect-Edge j n ≡
  {|´*Mul-mut-init n* ∧ *Z* (´*Muts*!*j*)|}
  ⟨*IF T*(´*Muts*!*j*) ∈ *Reach* ´*E THEN*
  ´*E*:= ´*E*[*R* (´*Muts*!*j*):= (*fst* (´*E*!*R*(´*Muts*!*j*)), *T* (´*Muts*!*j*))] *FI*,,
  ´*Muts*:= ´*Muts*[*j*:= (´*Muts*!*j*) (|*Z*:=*False*|)]⟩

definition *Mul-Color-Target* :: *nat* ⇒ *nat* ⇒ *mul-gar-coll-state ann-com* **where**
  Mul-Color-Target j n ≡
  {|´*Mul-mut-init n* ∧ ¬ *Z* (´*Muts*!*j*)|}
  ⟨´*M*:=´*M*[*T* (´*Muts*!*j*):=*Black*],, ´*Muts*:=´*Muts*[*j*:= (´*Muts*!*j*) (|*Z*:=*True*|)]⟩

definition *Mul-Mutator* :: *nat* ⇒ *nat* ⇒ *mul-gar-coll-state ann-com* **where**
  Mul-Mutator j n ≡
  {|´*Mul-mut-init n* ∧ *Z* (´*Muts*!*j*)|}
  WHILE True
    INV {|´*Mul-mut-init n* ∧ *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≤j* ∧ *j<n* ⟹
 ⊢ *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≤j* ∧ *j<n* ⟹
 ⊢ *Mul-Color-Target j n*
  ⦃´*Mul-mut-init n* ∧ *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≤j* ∧ *j<n* ⟹
⊢ *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*:
 ⟦*0≤i*; *i<n*; *0≤j*; *j<n*; *i≠j*⟧ ⟹
 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*:
 ⟦*0≤i*; *i<n*; *0≤j*; *j<n*; *i≠j*⟧ ⟹
 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*:
 ⟦*0*≤*i*; *i*<*n*; *0*≤*j*; *j*<*n*; *i*≠*j*⟧ ⟹
 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*:
 ⟦*0*≤*i*; *i*<*n*; *0*≤*j*; *j*<*n*; *i*≠*j*⟧ ⟹
 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*: ⟦*i* < *n*; *j* < *n*; *i* ≠ *j*⟧ ⟹
 interfree-aux (*Some* (*Mul-Mutator i n*), {}, *Some* (*Mul-Mutator j n*))
apply(*unfold Mul-Mutator-def*)
apply(*interfree-aux*)
apply(*simp-all add*:*mul-mutator-interfree*)
apply(*simp-all add*: *mul-mutator-defs*)
apply(*tactic* ‹*TRYALL* (*interfree-aux-tac* ***context***)›)
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
apply (*simp-all add*:*nth-list-update*)
done

Modular Parameterized Mutators

lemma *Mul-Parameterized-Mutators*: *0*<*n* ⟹
 ∥− {|´*Mul-mut-init n* ∧ (∀ *i*<*n*. *Z* (´*Muts*!*i*))|}
 COBEGIN
 SCHEME [*0*≤ *j*< *n*]
 Mul-Mutator j n
 {|*False*|}
 COEND
 {|*False*|}

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

2.3.2 The Collector

definition *Queue* :: *mul-gar-coll-state* ⇒ *nat* **where**
Queue ≡ « *length* (*filter* (λ*i*. ¬ *Z i* ∧ ´*M*!(*T i*) ≠ *Black*) ´*Muts*) »

consts *M-init* :: *nodes*

definition *Proper-M-init* :: *mul-gar-coll-state* ⇒ *bool* **where**
Proper-M-init ≡ « *Blacks M-init*=*Roots* ∧ *length M-init*=*length* ´*M* »

definition *Mul-Proper* :: *mul-gar-coll-state* ⇒ *nat* ⇒ *bool* **where**
Mul-Proper ≡ « λ*n*. *Proper-Roots* ´*M* ∧ *Proper-Edges* (´*M*, ´*E*) ∧ ´*Proper-M-init* ∧ *n*=*length* ´*Muts* »

definition *Safe* :: *mul-gar-coll-state* ⇒ *bool* **where**
Safe ≡ « *Reach* ´*E* ⊆ *Blacks* ´*M* »

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

Blackening Roots

definition *Mul-Blacken-Roots* :: *nat* ⇒ *mul-gar-coll-state ann-com* **where**
Mul-Blacken-Roots n ≡
{|´*Mul-Proper n*|}
´*ind*:=*0*;;
{|´*Mul-Proper n* ∧ ´*ind*=*0*|}
WHILE ´*ind*<*length* ´*M*
INV {|´*Mul-Proper n* ∧ (∀ *i*<´*ind*. *i*∈*Roots* ⟶ ´*M*!*i*=*Black*) ∧ ´*ind*≤*length* ´*M*|}
DO {|´*Mul-Proper n* ∧ (∀ *i*<´*ind*. *i*∈*Roots* ⟶ ´*M*!*i*=*Black*) ∧ ´*ind*<*length* ´*M*|}
IF ´*ind*∈*Roots THEN*
{|´*Mul-Proper n* ∧ (∀ *i*<´*ind*. *i*∈*Roots* ⟶ ´*M*!*i*=*Black*) ∧ ´*ind*<*length* ´*M* ∧ ´*ind*∈*Roots*|}
´*M*:=´*M*[´*ind*:=*Black*] *FI*;;
{|´*Mul-Proper n* ∧ (∀ *i*<´*ind*+*1*. *i*∈*Roots* ⟶ ´*M*!*i*=*Black*) ∧ ´*ind*<*length* ´*M*|}
´*ind*:=´*ind*+*1*
OD

lemma *Mul-Blacken-Roots*:
⊢ *Mul-Blacken-Roots n*
{|´*Mul-Proper 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-Proper n* ∧ *Roots*⊆*Blacks* ´*M* ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
  ∧ (´*Safe* ∨ ´*l*≤´*Queue* ∨ ´*obc*⊂*Blacks* ´*M*)|}
 ´*ind*:=*0*;;
 {|´*Mul-Proper 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-Proper n* ∧ *Roots*⊆*Blacks* ´*M*
     ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
     ∧ ´*Mul-PBInv* ∧ ´*ind*≤*length* ´*E*|}
 DO {|´*Mul-Proper n* ∧ *Roots*⊆*Blacks* ´*M*
    ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
    ∧ ´*Mul-PBInv* ∧ ´*ind*<*length* ´*E*|}
   IF ´*M*!(*fst* (´*E*!´*ind*))=*Black THEN*
   {|´*Mul-Proper 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-Proper 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-Proper n* ∧ *Roots*⊆*Blacks* ´*M*
     ∧ ´*obc*⊆*Blacks* ´*M* ∧ ´*bc*⊆*Blacks* ´*M*
     ∧ ´*Mul-PBInv* ∧ ´*ind*<*length* ´*E*|}

⟨*IF ´M*!(*fst* (*´E*!*´ind*))≠*Black THEN ´ind*:=*´ind*+*1 FI*⟩ *FI*
OD

lemma *Mul-Propagate-Black*:
⊢ *Mul-Propagate-Black n*
{|*´Mul-Proper n* ∧ *Roots*⊆*Blacks ´M* ∧ *´obc*⊆*Blacks ´M* ∧ *´bc*⊆*Blacks ´M*
∧ (*´Safe* ∨ *´obc*⊂*Blacks ´M* ∨ *´l*<*´Queue* ∧ (*´l*≤*´Queue* ∨ *´obc*⊂*Blacks*
´M))|}
apply(*unfold Mul-Propagate-Black-def*)
apply *annhoare*
apply(*simp-all add*:*Mul-PBInv-def mul-collector-defs Mul-Auxk-def Graph6 Graph7*
Graph8 Graph12 mul-collector-defs Queue-def)
— 8 subgoals left
apply *force*
apply *force*
apply *force*
apply(*force simp add*:*BtoW-def Graph-defs*)
— 4 subgoals left
apply *clarify*
apply(*simp add*: *mul-collector-defs Graph12 Graph6 Graph7 Graph8*)
apply(*disjE-tac*)
 apply(*simp-all add*:*Graph12 Graph13*)
 apply(*case-tac M x*! *k x*=*Black*)
  apply(*simp add*: *Graph10*)
 apply(*rule disjI2*, *rule disjI1*, *erule subset-psubset-trans*, *erule Graph11*, *force*)
apply(*case-tac M x*! *k x*=*Black*)
 apply(*simp add*: *Graph10 BtoW-def*)
 apply(*rule disjI2*, *clarify*, *erule less-SucE*, *force*)
 apply(*case-tac M x*!*snd*(*E x*! *ind x*)=*Black*)
  apply(*force*)
 apply(*force*)
apply(*rule disjI2*, *rule disjI1*, *erule subset-psubset-trans*, *erule Graph11*, *force*)
— 2 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
 apply (*simp-all*)
apply *clarify*
apply(*erule less-SucE*)
 apply *force*
apply (*simp add*:*BtoW-def*)
— 1 subgoal left
apply *clarify*
apply *simp*
apply(*disjE-tac*)
apply (*simp-all*)
apply(*rule disjI1* , *rule Graph1*)
 apply *simp-all*
done

Counting Black Nodes

```
definition Mul-CountInv :: mul-gar-coll-state ⇒ nat ⇒ bool where
  Mul-CountInv ≡ « λind. {i. i<ind ∧ ´Ma!i=Black}⊆´bc »

definition Mul-Count :: nat ⇒  mul-gar-coll-state ann-com where
  Mul-Count n ≡
  {|´Mul-Proper n ∧ Roots⊆Blacks ´M
    ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
    ∧ length ´Ma=length ´M
    ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks ´M) )
    ∧ ´q<n+1 ∧ ´bc={}|}
  ´ind:=0;;
  {|´Mul-Proper n ∧ Roots⊆Blacks ´M
    ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
    ∧ length ´Ma=length ´M
    ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks ´M) )
    ∧ ´q<n+1 ∧ ´bc={} ∧ ´ind=0|}
  WHILE ´ind<length ´M
     INV {|´Mul-Proper n ∧ Roots⊆Blacks ´M
          ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
          ∧ length ´Ma=length ´M ∧ ´Mul-CountInv ´ind
        ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks ´M))
          ∧ ´q<n+1 ∧ ´ind≤length ´M|}
  DO {|´Mul-Proper n ∧ Roots⊆Blacks ´M
       ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
       ∧ length ´Ma=length ´M ∧ ´Mul-CountInv ´ind
       ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks ´M))
       ∧ ´q<n+1 ∧ ´ind<length ´M|}
     IF ´M!´ind=Black
     THEN {|´Mul-Proper n ∧ Roots⊆Blacks ´M
           ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
           ∧ length ´Ma=length ´M ∧ ´Mul-CountInv ´ind
             ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks
´M))
           ∧ ´q<n+1 ∧ ´ind<length ´M ∧ ´M!´ind=Black|}
          ´bc:=insert ´ind ´bc
     FI;;
  {|´Mul-Proper n ∧ Roots⊆Blacks ´M
    ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆Blacks ´M
    ∧ length ´Ma=length ´M ∧ ´Mul-CountInv (´ind+1)
    ∧ (´Safe ∨ ´obc⊂Blacks ´Ma ∨ ´l<´q ∧ (´q≤´Queue ∨ ´obc⊂Blacks ´M))
    ∧ ´q<n+1 ∧ ´ind<length ´M|}
  ´ind:=´ind+1
  OD

lemma Mul-Count:
  ⊢ Mul-Count n
  {|´Mul-Proper n ∧ Roots⊆Blacks ´M
    ∧ ´obc⊆Blacks ´Ma ∧ Blacks ´Ma⊆Blacks ´M ∧ ´bc⊆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*⦄
apply (*unfold Mul-Count-def*)
apply *annhoare*
apply(*simp-all add*:*Mul-CountInv-def mul-collector-defs Mul-Auxk-def Graph6 Graph7 Graph8 Graph12 mul-collector-defs Queue-def*)
— 7 subgoals left
apply *force*
apply *force*
apply *force*
— 4 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
apply *simp-all*
apply(*simp add*:*Blacks-def*)
apply *clarify*
apply(*erule less-SucE*)
back
apply *force*
apply *force*
— 3 subgoals left
apply *clarify*
apply(*conjI-tac*)
apply(*disjE-tac*)
apply *simp-all*
apply *clarify*
apply(*erule less-SucE*)
back
apply *force*
apply *simp*
apply(*rotate-tac −1*)
apply (*force simp add*:*Blacks-def*)
— 2 subgoals left
apply *force*
— 1 subgoal left
apply *clarify*
apply(*drule-tac x = ind x* **in** *le-imp-less-or-eq*)
apply (*simp-all add*:*Blacks-def*)
done

Appending garbage nodes to the free list

axiomatization *Append-to-free* :: *nat* $\times$ *edges* $\Rightarrow$ *edges*
where
Append-to-free0: *length* (*Append-to-free* (*i*, *e*)) = *length e* **and**
Append-to-free1: *Proper-Edges* (*m*, *e*)
$\Longrightarrow$ *Proper-Edges* (*m*, *Append-to-free*(*i*, *e*)) **and**

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

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* ⇒ *mul-gar-coll-state ann-com* **where**
Mul-Collector n ≡
{|´*Mul-Proper n*|}

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

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

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

2.3.3 Interference Freedom

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

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

lemma *Mul-interfree-Blacken-Roots-Redirect-Edge*: ⟦*0*≤*j*; *j*<*n*⟧ ⟹
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: ⟦$0 \le j$; $j<n$⟧⟹
  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*: ⟦$0 \le j$; $j<n$⟧⟹
  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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
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*: $[\![0 \leq j;\ j<n]\!] \Longrightarrow$
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*: $[\![0 \leq j;\ j<n]\!] \Longrightarrow$
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
  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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
  interfree-aux (*Some*(*Mul-Append n*),{}, *Some*(*Mul-Redirect-Edge j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
  interfree-aux (*Some*(*Mul-Redirect-Edge j n*),{},*Some*(*Mul-Append n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
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*: ⟦*0*≤*j*; *j*<*n*⟧⟹
  interfree-aux (*Some*(*Mul-Append n*),{}, *Some*(*Mul-Color-Target j n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
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*: ⟦*0≤j*; *j<n*⟧⟹
interfree-aux (*Some*(*Mul-Color-Target j n*),{}, *Some*(*Mul-Append n*))
apply (*unfold mul-modules*)
apply *interfree-aux*
apply(*tactic* ‹*ALLGOALS* (*clarify-tac* ***context***)›)
apply(*simp-all add*: *mul-mutator-defs nth-list-update*)
apply(*simp add*:*Mul-AppendInv-def Append-to-free0*)
done

Interference freedom Collector-Mutator

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

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

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

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

Interference freedom Mutator-Collector

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

The Multi-Mutator Garbage Collection Algorithm

The total number of verification conditions is 328

lemma *Mul-Gar-Coll*:
‖− {|´*Mul-Proper n* ∧ ´*Mul-mut-init n* ∧ (∀ *i*<*n*. *Z* (´*Muts*!*i*))|}
COBEGIN
Mul-Collector n
{|*False*|}
‖
SCHEME [*0*≤ *j*< *n*]
Mul-Mutator j n
{|*False*|}
COEND
{|*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*.

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

datatype *'a com* =
  Basic 'a ⇒'a
  | *Seq 'a com 'a com*
  | *Cond 'a bexp 'a com 'a com*
  | *While 'a bexp 'a com*
  | *Await 'a bexp 'a com*

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

end

3.2 Operational Semantics

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

3.2.1 Semantics of Component Programs

Environment transitions

type-synonym *'a conf* = ((*'a com*) *option*) × *'a*

inductive-set
  etran :: (*'a conf* × *'a conf*) *set*
  and *etran'* :: *'a conf* ⇒ *'a conf* ⇒ *bool* (‹- −*e*→ -› [*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'* :: $'a\ conf \Rightarrow 'a\ conf \Rightarrow bool$ (‹- −c→ -› [81,81] 80)
  and *ctrans* :: $'a\ conf \Rightarrow 'a\ conf \Rightarrow bool$ (‹- −c∗→ -› [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*: $[\![s{\in}b;\ (Some\ P,\ s)\ -c{*}\rightarrow\ (None,\ t)]\!] \Longrightarrow (Some(Await\ b\ P),\ s)\ -c\rightarrow\ (None,\ t)$

monos *rtrancl-mono*

3.2.2 Semantics of Parallel Programs

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

inductive-set
  par-etran :: $('a\ par\text{-}conf \times 'a\ par\text{-}conf)\ set$
  and *par-etran'* :: $['a\ par\text{-}conf,'a\ par\text{-}conf] \Rightarrow bool$ (‹- −pe→ -› [81,81] 80)
where
  $P\ -pe\rightarrow\ Q \equiv (P,Q) \in par\text{-}etran$
| *ParEnv*: $(Ps,\ s)\ -pe\rightarrow\ (Ps,\ t)$

inductive-set
  par-ctran :: $('a\ par\text{-}conf \times 'a\ par\text{-}conf)\ set$
  and *par-ctran'* :: $['a\ par\text{-}conf,'a\ par\text{-}conf] \Rightarrow bool$ (‹- −pc→ -› [81,81] 80)

where
P −pc→ Q ≡ (*P*,*Q*) ∈ *par-ctran*
| *ParComp*: ⟦*i*<*length Ps*; (*Ps*!*i*, *s*) −*c*→ (*r*, *t*)⟧ ⟹ (*Ps*, *s*) −*pc*→ (*Ps*[*i*:=*r*], *t*)

lemma *par-ctranE*: *c* −*pc*→ *c*′ ⟹
(⋀*i Ps s r t*. *c* = (*Ps*, *s*) ⟹ *c*′ = (*Ps*[*i* := *r*], *t*) ⟹ *i* < *length Ps* ⟹
(*Ps* ! *i*, *s*) −*c*→ (*r*, *t*) ⟹ *P*) ⟹ *P*
by (*induct c*, *induct c*′, *erule par-ctran.cases*, *blast*)

3.2.3 Computations

Sequential computations

type-synonym ′*a confs* = ′*a conf list*

inductive-set *cptn* :: ′*a confs set*
where
CptnOne: [(*P*,*s*)] ∈ *cptn*
| *CptnEnv*: (*P*, *t*)#*xs* ∈ *cptn* ⟹ (*P*,*s*)#(*P*,*t*)#*xs* ∈ *cptn*
| *CptnComp*: ⟦(*P*,*s*) −*c*→ (*Q*,*t*); (*Q*, *t*)#*xs* ∈ *cptn* ⟧ ⟹ (*P*,*s*)#(*Q*,*t*)#*xs* ∈ *cptn*

definition *cp* :: (′*a com*) *option* ⇒ ′*a* ⇒ (′*a confs*) *set* **where**
cp P s ≡ {*l*. *l*!*0*=(*P*,*s*) ∧ *l* ∈ *cptn*}

Parallel computations

type-synonym ′*a par-confs* = ′*a par-conf list*

inductive-set *par-cptn* :: ′*a par-confs set*
where
ParCptnOne: [(*P*,*s*)] ∈ *par-cptn*
| *ParCptnEnv*: (*P*,*t*)#*xs* ∈ *par-cptn* ⟹ (*P*,*s*)#(*P*,*t*)#*xs* ∈ *par-cptn*
| *ParCptnComp*: ⟦ (*P*,*s*) −*pc*→ (*Q*,*t*); (*Q*,*t*)#*xs* ∈ *par-cptn* ⟧ ⟹ (*P*,*s*)#(*Q*,*t*)#*xs* ∈ *par-cptn*

definition *par-cp* :: ′*a par-com* ⇒ ′*a* ⇒ (′*a par-confs*) *set* **where**
par-cp P s ≡ {*l*. *l*!*0*=(*P*,*s*) ∧ *l* ∈ *par-cptn*}

3.2.4 Modular Definition of Computation

definition *lift* :: ′*a com* ⇒ ′*a conf* ⇒ ′*a conf* **where**
lift Q ≡ λ(*P*, *s*). (*if P*=*None then* (*Some Q*,*s*) *else* (*Some*(*Seq* (*the P*) *Q*), *s*))

inductive-set *cptn-mod* :: (′*a confs*) *set*
where
CptnModOne: [(*P*, *s*)] ∈ *cptn-mod*
| *CptnModEnv*: (*P*, *t*)#*xs* ∈ *cptn-mod* ⟹ (*P*, *s*)#(*P*, *t*)#*xs* ∈ *cptn-mod*
| *CptnModNone*: ⟦(*Some P*, *s*) −*c*→ (*None*, *t*); (*None*, *t*)#*xs* ∈ *cptn-mod* ⟧ ⟹ (*Some P*,*s*)#(*None*, *t*)#*xs* ∈*cptn-mod*

| *CptnModCondT*: ⟦(*Some P0*, *s*)#*ys* ∈ *cptn-mod*; *s* ∈ *b* ⟧ ⟹ (*Some*(*Cond b P0 P1*), *s*)#(*Some P0*, *s*)#*ys* ∈ *cptn-mod*
| *CptnModCondF*: ⟦(*Some P1*, *s*)#*ys* ∈ *cptn-mod*; *s* ∉ *b* ⟧ ⟹ (*Some*(*Cond b P0 P1*), *s*)#(*Some P1*, *s*)#*ys* ∈ *cptn-mod*
| *CptnModSeq1*: ⟦(*Some P0*, *s*)#*xs* ∈ *cptn-mod*; *zs*=*map* (*lift P1*) *xs* ⟧
⟹ (*Some*(*Seq P0 P1*), *s*)#*zs* ∈ *cptn-mod*
| *CptnModSeq2*:
⟦(*Some P0*, *s*)#*xs* ∈ *cptn-mod*; *fst*(*last* ((*Some P0*, *s*)#*xs*)) = *None*;
(*Some P1*, *snd*(*last* ((*Some P0*, *s*)#*xs*)))#*ys* ∈ *cptn-mod*;
zs=(*map* (*lift P1*) *xs*)@*ys* ⟧ ⟹ (*Some*(*Seq P0 P1*), *s*)#*zs* ∈ *cptn-mod*

| *CptnModWhile1*:
⟦ (*Some P*, *s*)#*xs* ∈ *cptn-mod*; *s* ∈ *b*; *zs*=*map* (*lift* (*While b P*)) *xs* ⟧
⟹ (*Some*(*While b P*), *s*)#(*Some*(*Seq P* (*While b P*)), *s*)#*zs* ∈ *cptn-mod*
| *CptnModWhile2*:
⟦ (*Some P*, *s*)#*xs* ∈ *cptn-mod*; *fst*(*last* ((*Some P*, *s*)#*xs*))=*None*; *s* ∈ *b*;
zs=(*map* (*lift* (*While b P*)) *xs*)@*ys*;
(*Some*(*While b P*), *snd*(*last* ((*Some P*, *s*)#*xs*)))#*ys* ∈ *cptn-mod*⟧
⟹ (*Some*(*While b P*), *s*)#(*Some*(*Seq P* (*While b P*)), *s*)#*zs* ∈ *cptn-mod*

3.2.5 Equivalence of Both Definitions.

lemma *last-length*: ((*a*#*xs*)!(*length xs*))=*last* (*a*#*xs*)
by (*induct xs*) *auto*

lemma *div-seq* [*rule-format*]: *list* ∈ *cptn-mod* ⟹
(∀ *s P Q zs*. *list*=(*Some* (*Seq P Q*), *s*)#*zs* ⟶
(∃ *xs*. (*Some P*, *s*)#*xs* ∈ *cptn-mod* ∧ (*zs*=(*map* (*lift Q*) *xs*) ∨
(*fst*(((*Some P*, *s*)#*xs*)!*length xs*)=*None* ∧
(∃ *ys*. (*Some Q*, *snd*(((*Some P*, *s*)#*xs*)!*length xs*))#*ys* ∈ *cptn-mod*
∧ *zs*=(*map* (*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*]:
∀ *s Q t xs*.((*Some a*, *s*), *Q*, *t*) ∈ *ctran* ⟶ (*Q*, *t*) # *xs* ∈ *cptn-mod*
⟶ (*Some a*, *s*) # (*Q*, *t*) # *xs* ∈ *cptn-mod*
supply [[*simproc del*: *defined-all*]]
apply(*induct a*)
apply *simp-all*
— basic
apply *clarify*
apply(*erule ctran.cases*,*simp-all*)
apply(*rule CptnModNone*,*rule Basic*,*simp*)
apply *clarify*
apply(*erule ctran.cases*,*simp-all*)
— Seq1
apply(*rule-tac xs*=[(*None*,*ta*)] **in** *CptnModSeq2*)
apply(*erule CptnModNone*)
apply(*rule CptnModOne*)
apply *simp*
apply *simp*
apply(*simp add*:*lift-def*)
— Seq2
apply(*erule-tac x*=*sa* **in** *allE*)
apply(*erule-tac x*=*Some P2* **in** *allE*)
apply(*erule allE*,*erule impE*, *assumption*)
apply(*drule div-seq*,*simp*)
apply *clarify*
apply(*erule disjE*)
apply *clarify*
apply(*erule allE*,*erule impE*, *assumption*)
apply(*erule-tac CptnModSeq1*)
apply(*simp add*:*lift-def*)

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

lemma *cptn-onlyif-cptn-mod* [*rule-format*]: *c* ∈ *cptn* ⟹ *c* ∈ *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*∈*cptn* ⟹ *map* (*lift P*) *c* ∈ *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*: ⟦*xs*≠[]; *fst*(*xs*!(*length xs* − (*Suc 0*)))=*None*⟧
⟹ *fst*((*map* (*lift P*) *xs*)!(*length* (*map* (*lift P*) *xs*)− (*Suc 0*)))=(*Some P*)
by (*cases* (*xs* ! (*length xs* − (*Suc 0*)))) (*simp add*:*lift-def*)

lemma *last-fst* [*rule-format*]: *P*((*a*#*x*)!*length x*) ⟶ ¬*P a* ⟶ *P* (*x*!(*length x* − (*Suc 0*)))
by (*induct x*) *simp-all*

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

lemma *last-snd*: *xs*≠[] ⟹
snd(((*map* (*lift P*) *xs*))!(*length* (*map* (*lift P*) *xs*) − (*Suc 0*)))=*snd*(*xs*!(*length xs* − (*Suc 0*)))
by (*cases* (*xs* ! (*length xs* − (*Suc 0*)))) (*simp-all add*:*lift-def*)

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

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

lemma *lift-nth*: *i*<*length xs* ⟹ *map* (*lift Q*) *xs* ! *i* = *lift Q* (*xs*! *i*)
by (*simp add*:*lift-def*)

lemma *snd-lift*: *i*< *length xs* ⟹ *snd*(*lift Q* (*xs* ! *i*))= *snd* (*xs* ! *i*)
by (*cases xs*!*i*) (*simp add*:*lift-def*)

lemma *cptn-if-cptn-mod*: *c* ∈ *cptn-mod* ⟹ *c* ∈ *cptn*
apply(*erule cptn-mod.induct*)
apply(*rule CptnOne*)
apply(*erule CptnEnv*)
apply(*erule CptnComp*,*simp*)
apply(*rule CptnComp*)
apply(*erule CondT*,*simp*)
apply(*rule CptnComp*)
apply(*erule CondF*,*simp*)
— Seq1

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

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

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

3.3 Validity of Correctness Formulas

3.3.1 Validity for Component Programs.

type-synonym *'a rgformula* =
'a com × *'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 ≡ ∀ *c*∈*set xs*. *c*=*None*

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

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

definition *par-com-validity* :: *'a par-com* ⇒ *'a set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*
⇒ *'a set* ⇒ *bool* (‹⊨ - *SAT* [-, -, -, -]› [*60*,*0*,*0*,*0*,*0*] *45*) **where**
⊨ *Ps SAT* [*pre*, *rely*, *guar*, *post*] ≡
∀ *s*. *par-cp Ps s* ∩ *par-assum*(*pre*, *rely*) ⊆ *par-comm*(*guar*, *post*)

3.3.3 Compositionality of the Semantics

Definition of the conjoin operator

definition *same-length* :: *'a par-confs* ⇒ (*'a confs*) *list* ⇒ *bool* **where**
same-length c clist ≡ (∀ *i*<*length clist*. *length*(*clist*!*i*)=*length c*)

definition *same-state* :: *'a par-confs* ⇒ (*'a confs*) *list* ⇒ *bool* **where**
same-state c clist ≡ (∀ *i* <*length clist*. ∀ *j*<*length c*. *snd*(*c*!*j*) = *snd*((*clist*!*i*)!*j*))

definition *same-program* :: *'a par-confs* ⇒ (*'a confs*) *list* ⇒ *bool* **where**
same-program c clist ≡ (∀ *j*<*length c*. *fst*(*c*!*j*) = *map* (λ*x*. *fst*(*nth x j*)) *clist*)

definition *compat-label* :: *'a par-confs* ⇒ (*'a confs*) *list* ⇒ *bool* **where**
compat-label c clist ≡ (∀ *j*. *Suc j*<*length c* ⟶
(*c*!*j* −*pc*→ *c*!*Suc j* ∧ (∃ *i*<*length clist*. (*clist*!*i*)!*j* −*c*→ (*clist*!*i*)! *Suc j* ∧
(∀ *l*<*length clist*. *l*≠*i* ⟶ (*clist*!*l*)!*j* −*e*→ (*clist*!*l*)! *Suc j*))) ∨
(*c*!*j* −*pe*→ *c*!*Suc j* ∧ (∀ *i*<*length clist*. (*clist*!*i*)!*j* −*e*→ (*clist*!*i*)! *Suc j*)))

definition *conjoin* :: *'a par-confs* ⇒ (*'a confs*) *list* ⇒ *bool* (‹- ∝ -› [*65*,*65*] *64*)
where
c ∝ *clist* ≡ (*same-length c clist*) ∧ (*same-state c clist*) ∧ (*same-program c clist*)
∧ (*compat-label c clist*)

Some previous lemmas

lemma *list-eq-if* [*rule-format*]:
∀ *ys*. *xs*=*ys* ⟶ (*length xs* = *length ys*) ⟶ (∀ *i*<*length xs*. *xs*!*i*=*ys*!*i*)
by (*induct xs*) *auto*

lemma *list-eq*: (*length xs* = *length ys* ∧ (∀ *i*<*length xs*. *xs*!*i*=*ys*!*i*)) = (*xs*=*ys*)
apply(*rule iffI*)
apply *clarify*
apply(*erule nth-equalityI*)
apply *simp+*
done

lemma *nth-tl*: ⟦ *ys*!*0*=*a*; *ys*≠[] ⟧ ⟹ *ys*=(*a*#(*tl ys*))
by (*cases ys*) *simp-all*

lemma *nth-tl-if* [*rule-format*]: *ys*≠[] ⟶ *ys*!*0*=*a* ⟶ *P ys* ⟶ *P* (*a*#(*tl ys*))
by (*induct ys*) *simp-all*

lemma *nth-tl-onlyif* [*rule-format*]: *ys*≠[] ⟶ *ys*!*0*=*a* ⟶ *P* (*a*#(*tl ys*)) ⟶ *P ys*
by (*induct ys*) *simp-all*

lemma *seq-not-eq1*: *Seq c1 c2*≠*c1*
by (*induct c1*) *auto*

lemma *seq-not-eq2*: *Seq c1 c2*≠*c2*

by (*induct c2*) *auto*

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

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

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

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

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

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

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

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

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

3.3.4 The Semantics is Compositional

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

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

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

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

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

apply *clarify*
apply *simp*
apply(*erule-tac x=Suc*(*Suc nat*) **and** *P*=λ*j*. *H j* ⟶ (*fst* (*s j*))=(*t j*) **for** *H s t* **in** *allE*,*simp*)
apply(*rule nth-equalityI*,*simp*,*simp*)
apply(*force simp add*:*length-Suc-conv*)
apply(*rule allI*,*rule impI*)
apply(*erule-tac x=Suc j* **and** *P*=λ*j*. *H j* ⟶ (*I j* ∨ *J j*) **for** *H I 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*=λ*i*. *H i* ⟶ *J i* ∈*etran* **for** *H J* **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*=λ*j*. *H j* ⟶ (*snd* (*d j*))=(*snd* (*e j*)) **for** *H d e* **in** *allE*,*simp*)
apply(*rule nth-tl-if*)
apply(*erule-tac x=i* **and** *P*=λ*j*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
apply *simp*
apply(*erule tl-zero*,*force*)
apply(*erule-tac x=i* **and** *P*=λ*j*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
apply *clarify*
apply(*erule-tac x=l* **and** *P*=λ*i*. *H i* ⟶ *J i* ∈*etran* **for** *H J* **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*=λ*j*. *H j* ⟶ (*snd* (*d j*))=(*snd* (*e j*)) **for** *H d e* **in** *allE*,*simp*)
apply(*rule nth-tl-if*)
apply(*erule-tac x=l* **and** *P*=λ*j*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
apply *simp*
apply(*rule tl-zero*,*force*)
apply *force*
apply(*erule-tac x=l* **and** *P*=λ*j*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s 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*=λ*i*. *H i* ⟶ *J i* ∈*etran* **for** *H J* **in** *allE*, *erule impE*, *assumption*)
apply(*erule-tac x=i* **and** *P*=λ*i*. *H i* ⟶ *J i* ∈*etran* **for** *H J* **in** *allE*, *erule impE*, *assumption*)
apply(*force elim*:*etranE intro*:*Env*)
apply *force*
apply(*erule-tac x=i* **and** *P*=λ*j*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
apply *simp*

apply(*rule tl-zero*)
apply(*rule tl-zero*,*force*)
apply *force*
apply(*erule-tac x*=*i* **and** *P*=*λj*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
apply *force*
apply(*erule-tac x*=*i* **and** *P*=*λj*. *H j* ⟶ (*length* (*s j*) = *t*) **for** *H s t* **in** *allE*,*force*)
done

lemma *aux-onlyif* [*rule-format*]: ∀ *xs s*. (*xs*, *s*)#*ys* ∈ *par-cptn* ⟶
(∃ *clist*. (*length clist* = *length xs*) ∧
(*xs*, *s*)#*ys* ∝ *map* (*λi*. (*fst i*,*s*)#(*snd i*)) (*zip xs clist*) ∧
(∀ *i*<*length xs*. (*xs*!*i*,*s*)#(*clist*!*i*) ∈ *cptn*))
supply [[*simproc del*: *defined-all*]]
apply(*induct ys*)
apply(*clarify*)
apply(*rule-tac x*=*map* (*λi*. []) [*0*..<*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* (λ*j*. (*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*=λ*j*. *H j* ⟶ (*I j* ∈ *cptn*) **for** *H I* **in** *allE*,*simp*)
apply *simp*
apply(*erule-tac x=ia* **in** *allE*)
apply(*rule CptnEnv*,*simp*)
apply(*simp add*:*conjoin-def*)
apply (*rule conjI*)
apply(*simp add*:*same-length-def*)
apply *clarify*
apply(*case-tac i*=*ia*,*simp*,*simp*)
apply(*rule conjI*)
apply(*simp add*:*same-state-def*)
apply *clarify*
apply(*case-tac j*, *simp*, *simp* (*no-asm-simp*))
apply(*case-tac i*=*ia*,*simp*,*simp*)
apply(*rule conjI*)
apply(*simp add*:*same-program-def*)
apply *clarify*
apply(*case-tac j*,*simp*)
apply(*rule nth-equalityI*,*simp*,*simp*)
apply *simp*
apply(*rule nth-equalityI*,*simp*,*simp*)
apply(*erule-tac x=nat* **and** *P*=λ*j*. *H j* ⟶ (*fst* (*a j*))=((*b j*)) **for** *H a b* **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=λj. H j ⟶ (P j ∨ Q j) for H P Q in allE,simp)
apply(erule disjE)
 apply clarify
 apply(rule-tac x=ia in exI,simp)
 apply(rule conjI)
  apply(case-tac i=ia,simp,simp)
 apply clarify
 apply(case-tac i=l,simp)
  apply(case-tac l=ia,simp,simp)
  apply(erule-tac x=l in allE,erule impE,assumption,erule impE, assumption,simp)
 apply simp
 apply(erule-tac x=l in allE,erule impE,assumption,erule impE, assumption,simp)
apply clarify
apply(erule-tac x=ia and P=λj. H j ⟶ (P j)∈etran for H P in allE, erule
impE, assumption)
apply(case-tac i=ia,simp,simp)
done

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

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

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

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

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

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

end

3.4 The Proof System

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

3.4.1 Proof System for Component Programs

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

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

inductive
rghoare :: [′*a com*, ′*a set*, (′*a* × ′*a*) *set*, (′*a* × ′*a*) *set*, ′*a set*] ⇒ *bool*
(‹⊢ - *sat* [-, -, -, -]› [*60*,*0*,*0*,*0*,*0*] *45*)
where

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

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

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

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

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

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

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

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

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

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

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

3.4.2 Proof System for Parallel Programs

type-synonym *'a par-rgformula* =
(*'a rgformula*) *list* $\times$ *'a set* $\times$ (*'a* $\times$ *'a*) *set* $\times$ (*'a* $\times$ *'a*) *set* $\times$ *'a set*

inductive
par-rghoare :: (*'a rgformula*) *list* $\Rightarrow$ *'a set* $\Rightarrow$ (*'a* $\times$ *'a*) *set* $\Rightarrow$ (*'a* $\times$ *'a*) *set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
(‹$\vdash$ - *SAT* [-, -, -, -]› *[60,0,0,0,0] 45*)
where

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

3.5 Soundness

Some previous lemmas

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

lemma *etran-in-comm*:
$(P,\ t)\ \#\ xs \in comm(guar,\ post) \Longrightarrow (P,\ s)\ \#\ (P,\ t)\ \#\ xs \in comm(guar,\ post)$
apply(*simp add*:*comm-def*)
apply *clarify*
apply(*case-tac i*,*simp+*)
done

lemma *ctran-in-comm*:
$[\![(s,\ s) \in guar;\ (Q,\ s)\ \#\ xs \in comm(guar,\ post)]\!]$
$\Longrightarrow (P,\ s)\ \#\ (Q,\ s)\ \#\ xs \in comm(guar,\ post)$
apply(*simp add*:*comm-def*)
apply *clarify*
apply(*case-tac i*,*simp+*)
done

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

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

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

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

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

lemma *etran-or-ctran2-disjI2*:

⟦ x∈cptn; Suc i<length x; x!i −e→ x!Suc i⟧ ⟹ ¬ x!i −c→ x!Suc i
by(drule etran-or-ctran2,simp-all)

lemma not-ctran-None2 [rule-format]:
⟦ (None, s) # xs ∈cptn; i<length xs⟧ ⟹ ¬ ((None, s) # xs) ! i −c→ 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::nat) ⟶ (∃ m. P m ∧ (∀ i<m. ¬ P i))
apply(rule nat-less-induct)
apply clarify
apply(case-tac ∀ m. m<n ⟶ ¬ P m)
apply auto
done

lemma stability [rule-format]:
∀ j k. x ∈ cptn ⟶ stable p rely ⟶ j≤k ⟶ k<length x ⟶ snd(x!j)∈p ⟶
(∀ i. (Suc i)<length x ⟶
(x!i −e→ x!(Suc i)) ⟶ (snd(x!i), snd(x!(Suc i))) ∈ rely) ⟶
(∀ i. j≤i ∧ i<k ⟶ x!i −e→ x!Suc i) ⟶ snd(x!k)∈p ∧ fst(x!j)=fst(x!k)
supply [[simproc del: defined-all]]
apply(induct x)
apply clarify
apply(force elim:cptn.cases)
apply clarify
apply(erule cptn.cases,simp)
apply simp
apply(case-tac k,simp,simp)
apply(case-tac j,simp)
apply(erule-tac x=0 **in** allE)
apply(erule-tac x=nat **and** P=λj. (0≤j) ⟶ (J j) **for** J **in** allE,simp)
apply(subgoal-tac t∈p)
apply(subgoal-tac (∀ i. i < length xs ⟶ ((P, t) # xs) ! i −e→ xs ! i ⟶ (snd (((P, t) # xs) ! i), snd (xs ! i)) ∈ rely))
apply clarify
apply(erule-tac x=Suc i **and** P=λj. (H j) ⟶ (J j)∈etran **for** H J **in** allE,simp)
apply clarify
apply(erule-tac x=Suc i **and** P=λj. (H j) ⟶ (J j) ⟶ (T j)∈rely **for** H J T **in** allE,simp)
apply(erule-tac x=0 **and** P=λj. (H j) ⟶ (J j)∈etran ⟶ T j **for** H J T **in** allE,simp)
apply(simp(no-asm-use) only:stable-def)
apply(erule-tac x=s **in** allE)

```
  apply(erule-tac x=t in allE)
  apply simp
  apply(erule mp)
  apply(erule mp)
  apply(rule Env)
 apply simp
 apply(erule-tac x=nata in allE)
 apply(erule-tac x=nat and P=λj. (s≤j) ⟶ (J j) for s J in allE,simp)
 apply(subgoal-tac (∀ i. i < length xs ⟶ ((P, t) # xs) ! i −e→ xs ! i ⟶ (snd
(((P, t) # xs) ! i), snd (xs ! i)) ∈ rely))
  apply clarify
  apply(erule-tac x=Suc i and P=λj. (H j) ⟶ (J j)∈etran for H J in allE,simp)
 apply clarify
 apply(erule-tac x=Suc i and P=λj. (H j) ⟶ (J j) ⟶ (T j)∈rely for H J T
in allE,simp)
apply(case-tac k,simp,simp)
apply(case-tac j)
 apply(erule-tac x=0 and P=λj. (H j) ⟶ (J j)∈etran for H J in allE,simp)
 apply(erule etran.cases,simp)
apply(erule-tac x=nata in allE)
apply(erule-tac x=nat and P=λj. (s≤j) ⟶ (J j) for s J in allE,simp)
apply(subgoal-tac (∀ i. i < length xs ⟶ ((Q, t) # xs) ! i −e→ xs ! i ⟶ (snd
(((Q, t) # xs) ! i), snd (xs ! i)) ∈ rely))
 apply clarify
 apply(erule-tac x=Suc i and P=λj. (H j) ⟶ (J j)∈etran for H J in allE,simp)
apply clarify
apply(erule-tac x=Suc i and P=λj. (H j) ⟶ (J j) ⟶ (T j)∈rely for H J T in
allE,simp)
done
```

3.5.1 Soundness of the System for Component Programs

Soundness of the Basic rule

```
lemma unique-ctran-Basic [rule-format]:
  ∀ s i. x ∈ cptn ⟶ x ! 0 = (Some (Basic f), s) ⟶
  Suc i<length x ⟶ x!i −c→ x!Suc i ⟶
  (∀ j. Suc j<length x ⟶ i≠j ⟶ x!j −e→ x!Suc j)
apply(induct x,simp)
apply simp
apply clarify
apply(erule cptn.cases,simp)
 apply(case-tac i,simp+)
 apply clarify
 apply(case-tac j,simp)
  apply(rule Env)
 apply simp
apply clarify
apply simp
apply(case-tac i)
```

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

lemma *exists-ctran-Basic-None* [*rule-format*]:
∀ *s i*. *x* ∈ *cptn* ⟶ *x* ! *0* = (*Some* (*Basic f*), *s*)
⟶ *i*<*length x* ⟶ *fst*(*x*!*i*)=*None* ⟶ (∃ *j*<*i*. *x*!*j* −*c*→ *x*!*Suc j*)
apply(*induct x*,*simp*)
apply *simp*
apply *clarify*
apply(*erule cptn.cases*,*simp*)
apply(*case-tac i*,*simp*,*simp*)
apply(*erule-tac x*=*nat* **in** *allE*,*simp*)
apply *clarify*
apply(*rule-tac x*=*Suc j* **in** *exI*,*simp*,*simp*)
apply *clarify*
apply(*case-tac i*,*simp*,*simp*)
apply(*rule-tac x*=*0* **in** *exI*,*simp*)
done

lemma *Basic-sound*:
⟦*pre* ⊆ {*s*. *f s* ∈ *post*}; {(*s*, *t*). *s* ∈ *pre* ∧ *t* = *f s*} ⊆ *guar*;
stable pre rely; *stable post rely*⟧
⟹ ⊨ *Basic f sat* [*pre*, *rely*, *guar*, *post*]
supply [[*simproc del*: *defined-all*]]
apply(*unfold com-validity-def*)
apply *clarify*
apply(*simp add*:*comm-def*)
apply(*rule conjI*)
apply *clarify*
apply(*simp add*:*cp-def assum-def*)
apply *clarify*
apply(*frule-tac j*=*0* **and** *k*=*i* **and** *p*=*pre* **in** *stability*)
apply *simp-all*
apply(*erule-tac x*=*ia* **in** *allE*,*simp*)
apply(*erule-tac i*=*i* **and** *f*=*f* **in** *unique-ctran-Basic*,*simp-all*)
apply(*erule subsetD*,*simp*)
apply(*case-tac x*!*i*)
apply *clarify*
apply(*drule-tac s*=*Some* (*Basic f*) **in** *sym*,*simp*)
apply(*thin-tac* ∀ *j*. *H j* **for** *H*)
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*]:
$\forall s\ i.\ x \in cptn \longrightarrow x\ !\ 0 = (Some\ (Await\ b\ c),\ s) \longrightarrow$
$Suc\ i{<}length\ x \longrightarrow x!i\ {-c\rightarrow}\ x!Suc\ i \longrightarrow$
$(\forall j.\ Suc\ j{<}length\ x \longrightarrow i{\neq}j \longrightarrow x!j\ {-e\rightarrow}\ x!Suc\ j)$
apply(*induct x*,*simp+*)
apply *clarify*
apply(*erule cptn.cases*,*simp*)
apply(*case-tac i*,*simp+*)
apply *clarify*
apply(*case-tac j*,*simp*)
apply(*rule Env*)
apply *simp*
apply *clarify*
apply *simp*
apply(*case-tac i*)
apply(*case-tac j*,*simp*,*simp*)
apply(*erule ctran.cases*,*simp-all*)
apply(*force elim*: *not-ctran-None*)

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

lemma exists-ctran-Await-None [rule-format]:
  ∀ s i.  x ∈ cptn ⟶ x ! 0 = (Some (Await b c), s)
  ⟶ i<length x ⟶ fst(x!i)=None ⟶ (∃ j<i. x!j −c→ x!Suc j)
apply(induct x,simp+)
apply clarify
apply(erule cptn.cases,simp)
 apply(case-tac i,simp+)
 apply(erule-tac x=nat in allE,simp)
 apply clarify
 apply(rule-tac x=Suc j in exI,simp,simp)
apply clarify
apply(case-tac i,simp,simp)
apply(rule-tac x=0 in exI,simp)
done

lemma Star-imp-cptn:
  (P, s) −c∗→ (R, t) ⟹ ∃ l ∈ cp P s. (last l)=(R, t)
  ∧ (∀ i. Suc i<length l ⟶ l!i −c→ l!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:
  ⟦stable pre rely; stable post rely;
  ∀ V. ⊢ P sat [pre ∩ b ∩ {s. s = V}, {(s, t). s = t},
                UNIV, {s. (V, s) ∈ guar} ∩ post] ∧
```

$\models$ *P sat* [*pre* $\cap$ *b* $\cap$ {*s*. *s* = *V*}, {(*s*, *t*). *s* = *t*},
UNIV, {*s*. (*V*, *s*) $\in$ *guar*} $\cap$ *post*] $\rrbracket$
$\Longrightarrow$ $\models$ *Await b P sat* [*pre*, *rely*, *guar*, *post*]
apply(*unfold com-validity-def*)
apply *clarify*
apply(*simp add*:*comm-def*)
apply(*rule conjI*)
apply *clarify*
apply(*simp add*:*cp-def assum-def*)
apply *clarify*
apply(*frule-tac j=0* **and** *k=i* **and** *p=pre* **in** *stability*,*simp-all*)
apply(*erule-tac x=ia* **in** *allE*,*simp*)
apply(*subgoal-tac x*$\in$ *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*=λ*i*. *H i* $\longrightarrow$ (*J i*, *I i*)$\in$*ctran* **for** *H J I* **in** *allE*,*simp*)
apply(*erule etranE*,*simp*)
apply *simp*
apply *clarify*
apply(*simp add*:*cp-def*)
apply *clarify*
apply(*frule-tac i=length x* $-$ *1* **in** *exists-ctran-Await-None*,*force*)
apply (*case-tac x*,*simp+*)
apply(*rule last-fst-esp*,*simp add*:*last-length*)
apply(*case-tac x*, *simp+*)
apply *clarify*
apply(*simp add*:*assum-def*)
apply *clarify*
apply(*frule-tac j=0* **and** *k=j* **and** *p=pre* **in** *stability*,*simp-all*)
apply(*erule-tac x=i* **in** *allE*,*simp*)
apply(*erule-tac i=j* **in** *unique-ctran-Await*,*force*,*simp-all*)
apply(*case-tac x*!*j*)
apply *clarify*
apply *simp*
apply(*drule-tac s=Some* (*Await b P*) **in** *sym*,*simp*)
apply(*case-tac x*!*Suc j*,*simp*)
apply(*rule ctran.cases*,*simp*)
apply(*simp-all*)
apply(*drule Star-imp-cptn*)

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

Soundness of the Conditional rule

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

apply *clarify*
apply *simp*
apply *clarify*
apply(*case-tac* $i \leq m$)
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* $x=i - (Suc\ m)$ **and** $P=\lambda j.\ H\ j \longrightarrow J\ j \longrightarrow (I\ j) \in guar$ **for** $H\ J\ I$ **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*
apply(*case-tac length* (*drop* (*Suc m*) *x*),*simp*)
apply(*erule-tac* $x=sa$ **in** *allE*)
back
apply(*drule-tac* $c=drop\ (Suc\ m)\ x$ **in** *subsetD*,*simp*)
apply *clarify*
apply *simp*
apply *clarify*
apply(*case-tac* $i \leq m$)
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** $P=\lambda j.\ H\ j \longrightarrow J\ j \longrightarrow (I\ j) \in guar$ **for** $H\ J\ I$ **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

lemma *Seq-sound1* [*rule-format*]:
  x∈ *cptn-mod* ⟹ ∀ *s P*. *x* !*0*=(*Some* (*Seq P Q*), *s*) ⟶
  (∀ *i*<*length x*. *fst*(*x*!*i*)≠*Some Q*) ⟶
  (∃ *xs*∈ *cp* (*Some P*) *s*. *x*=*map* (*lift Q*) *xs*)
  supply [[*simproc del*: *defined-all*]]
apply(*erule cptn-mod.induct*)
apply(*unfold cp-def*)
apply *safe*
apply *simp-all*
    apply(*simp add*:*lift-def*)
    apply(*rule-tac x*=[(*Some Pa*, *sa*)] **in** *exI*,*simp add*:*CptnOne*)
   apply(*subgoal-tac* (∀ *i* < *Suc* (*length xs*). *fst* (((*Some* (*Seq Pa Q*), *t*) # *xs*) ! *i*) ≠ *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*) ∈ *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

lemma *Seq-sound2* [*rule-format*]:
  x ∈ *cptn* ⟹ ∀ *s P i*. *x*!*0*=(*Some* (*Seq P Q*), *s*) ⟶ *i*<*length x*
  ⟶ *fst*(*x*!*i*)=*Some Q* ⟶
  (∀ *j*<*i*. *fst*(*x*!*j*)≠(*Some Q*)) ⟶
  (∃ *xs ys*. *xs* ∈ *cp* (*Some P*) *s* ∧ *length xs*=*Suc i*
   ∧ *ys* ∈ *cp* (*Some Q*) (*snd*(*xs* !*i*)) ∧ *x*=(*map* (*lift Q*) *xs*)@*tl ys*)
  supply [[*simproc del*: *defined-all*]]
apply(*erule cptn.induct*)
apply(*unfold cp-def*)
apply *safe*
apply *simp-all*
 apply(*case-tac i*,*simp*+)
 apply(*erule allE*,*erule impE*,*assumption*,*simp*)
 apply *clarify*
 apply(*subgoal-tac* (∀ *j* < *nat*. *fst* (((*Some* (*Seq Pa Q*), *t*) # *xs*) ! *j*) ≠ *Some Q*),*clarify*)

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


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

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

lemma Seq-sound:
  ⟦⊨ P sat [pre, rely, guar, mid]; ⊨ Q sat [mid, rely, guar, post]⟧
  ⟹ ⊨ Seq P Q sat [pre, rely, guar, post]
apply(unfold com-validity-def)
apply clarify
apply(case-tac ∃ i<length x. fst(x!i)=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=λj. H j ⟶ J j ⟶ I j for H J I in allE,erule impE,
assumption)
  apply(simp add:snd-lift)
  apply(erule mp)
  apply(force elim:etranE intro:Env simp add:lift-def)
 apply(simp add:comm-def)
 apply(rule conjI)
  apply clarify
   apply(erule-tac P=λj. H j ⟶ J j ⟶ I j for H J I in allE,erule impE,
assumption)
  apply(simp add:snd-lift)
  apply(erule mp)
  apply(case-tac (xs!i))
  apply(case-tac (xs! Suc i))
  apply(case-tac fst(xs!i))
   apply(erule-tac x=i in allE, simp add:lift-def)
  apply(case-tac fst(xs!Suc i))
   apply(force simp add:lift-def)
  apply(force simp add:lift-def)
 apply clarify
 apply(case-tac xs,simp add:cp-def)
 apply clarify
 apply (simp del:list.map)
 apply (rename-tac list)
 apply(subgoal-tac (map (lift Q) ((a, b) # list))≠[])
  apply(drule last-conv-nth)
  apply (simp del:list.map)
```

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

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

Soundness of the While rule

lemma *last-append*[*rule-format*]:
  ∀ *xs*. *ys*≠[] ⟶ ((*xs*@*ys*)!(*length* (*xs*@*ys*) − (*Suc 0*)))=(*ys*!(*length ys* − (*Suc 0*)))
apply(*induct ys*)
 apply *simp*
apply *clarify*
apply (*simp add*:*nth-append*)
done

lemma *assum-after-body*:
  ⟦ ⊨ *P sat* [*pre* ∩ *b*, *rely*, *guar*, *pre*];
  (*Some P*, *s*) # *xs* ∈ *cptn-mod*; *fst* (*last* ((*Some P*, *s*) # *xs*)) = *None*; *s* ∈ *b*;
  (*Some* (*While b P*), *s*) # (*Some* (*Seq P* (*While b P*)), *s*) #
   map (*lift* (*While b P*)) *xs* @ *ys* ∈ *assum* (*pre*, *rely*)⟧
  ⟹ (*Some* (*While b P*), *snd* (*last* ((*Some P*, *s*) # *xs*))) # *ys* ∈ *assum* (*pre*, *rely*)
apply(*simp add*:*assum-def com-validity-def cp-def cptn-iff-cptn-mod*)
apply *clarify*
apply(*erule-tac x*=*s* **in** *allE*)
apply(*drule-tac c*=(*Some P*, *s*) # *xs* **in** *subsetD*,*simp*)
 apply *clarify*
 apply(*erule-tac x*=*Suc i* **in** *allE*)
 apply *simp*
 apply(*simp add*:*Cons-lift-append nth-append snd-lift del*:*list*.*map*)
 apply(*erule mp*)
 apply(*erule etranE*,*simp*)
 apply(*case-tac fst*(((*Some P*, *s*) # *xs*) ! *i*))
  apply(*force intro*:*Env simp add*:*lift-def*)
 apply(*force intro*:*Env simp add*:*lift-def*)
apply(*rule conjI*)
 apply *clarify*
 apply(*simp add*:*comm-def last-length*)
apply *clarify*
apply(*rule conjI*)
 apply(*simp add*:*comm-def*)
apply *clarify*
apply(*erule-tac x*=*Suc*(*length xs* + *i*) **in** *allE*,*simp*)
apply(*case-tac i*, *simp add*:*nth-append Cons-lift-append snd-lift last-conv-nth lift-def split-def*)
apply(*simp add*:*Cons-lift-append nth-append snd-lift*)
done

lemma *While-sound-aux* [*rule-format*]:
  ⟦ *pre* ∩ − *b* ⊆ *post*; ⊨ *P sat* [*pre* ∩ *b*, *rely*, *guar*, *pre*]; ∀ *s*. (*s*, *s*) ∈ *guar*;
   stable pre rely; *stable post rely*; *x* ∈ *cptn-mod* ⟧
  ⟹ ∀ *s xs*. *x*=(*Some*(*While b P*),*s*)#*xs* ⟶ *x*∈*assum*(*pre*, *rely*) ⟶ *x* ∈ *comm* (*guar*, *post*)
  supply [[*simproc del*: *defined-all*]]
apply(*erule cptn-mod*.*induct*)
apply *safe*

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

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

apply(*simp add*:*snd-lift del*:*list.map last.simps*)
apply(*thin-tac* ∀ *i*. *i* < *length ys* ⟶ *P i* **for** *P*)
apply(*simp only*:*com-validity-def cp-def cptn-iff-cptn-mod*)
apply(*erule-tac x*=*sa* **in** *allE*)
apply(*drule-tac c*=(*Some P*, *sa*) # *xs* **in** *subsetD*)
apply (*simp add*:*assum-def del*:*list.map last.simps*)
apply *clarify*
apply(*erule-tac x*=*Suc ia* **in** *allE*,*simp add*:*nth-append snd-lift del*:*list.map last.simps*, *erule mp*)
apply(*case-tac fst*(((*Some P*, *sa*) # *xs*) ! *ia*))
apply(*erule etranE*,*simp add*:*lift-def*)
apply(*rule Env*)
apply(*erule etranE*,*simp add*:*lift-def*)
apply(*rule Env*)
apply (*simp add*:*comm-def del*:*list.map*)
apply *clarify*
apply(*erule allE*,*erule impE*,*assumption*)
apply(*erule mp*)
apply(*case-tac* ((*Some P*, *sa*) # *xs*) ! *i*)
apply(*case-tac xs*!*i*)
apply(*simp add*:*lift-def*)
apply(*case-tac fst*(*xs*!*i*))
apply *force*
apply *force*
— *i* ≥ *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*:*list.map last.simps*)
apply(*simp add*:*last-length del*:*last.simps*)
apply(*erule mp*)
apply(*case-tac last*((*Some P*, *sa*) # *xs*))
apply(*simp add*:*lift-def del*:*last.simps*)
— *i*>*length xs*
apply(*case-tac i*−*length xs*)
apply *arith*
apply(*simp add*:*nth-append del*:*list.map 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*:*list.map last.simps*)
apply(*subgoal-tac* (*map* (*lift* (*While b P*)) ((*Some P*, *sa*) # *xs*))≠[])

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

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

Soundness of the Rule of Consequence

lemma *Conseq-sound*:
⟦*pre* ⊆ *pre′*; *rely* ⊆ *rely′*; *guar′* ⊆ *guar*; *post′* ⊆ *post*;
⊨ *P sat* [*pre′*, *rely′*, *guar′*, *post′*]⟧
⟹ ⊨ *P sat* [*pre*, *rely*, *guar*, *post*]
apply(*simp add*:*com-validity-def assum-def comm-def*)
apply *clarify*
apply(*erule-tac x*=*s* **in** *allE*)
apply(*drule-tac c*=*x* **in** *subsetD*)
apply *force*

apply *force*
done

Soundness of the system for sequential component programs

theorem *rgsound*:
$\vdash P\ sat\ [pre,\ rely,\ guar,\ post] \Longrightarrow \models P\ sat\ [pre,\ rely,\ guar,\ post]$
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*:
$[\![\ \forall i{<}length\ xs.\ rely \cup (\bigcup j{\in}\{j.\ j < length\ xs \wedge j \neq i\}.\ Guar\ (xs\ !\ j))$
$\subseteq Rely\ (xs\ !\ i);$
$pre \subseteq (\bigcap i{\in}\{i.\ i < length\ xs\}.\ Pre\ (xs\ !\ i));$
$\forall i{<}length\ xs.$
$\models Com\ (xs\ !\ i)\ sat\ [Pre\ (xs\ !\ i),\ Rely\ (xs\ !\ i),\ Guar\ (xs\ !\ i),\ Post\ (xs\ !\ i)];$
$length\ xs{=}length\ clist;\ x \in par\text{-}cp\ (ParallelCom\ xs)\ s;\ x{\in}par\text{-}assum(pre,\ rely);$
$\forall i{<}length\ clist.\ clist!i{\in}cp\ (Some(Com(xs!i)))\ s;\ x \propto clist\ ]\!]$
$\Longrightarrow \forall j\ i.\ i{<}length\ clist \wedge Suc\ j{<}length\ x \longrightarrow (clist!i!j)\ {-c\rightarrow}\ (clist!i!Suc\ j)$
$\longrightarrow (snd(clist!i!j),\ snd(clist!i!Suc\ j)) \in Guar(xs!i)$
apply(*unfold par-cp-def*)
apply (*rule ccontr*)
— By contradiction:
apply *simp*
apply(*erule exE*)
— the first c-tran that does not satisfy the guarantee-condition is from σ-i at step m.
apply(*drule-tac n=j* **and** $P{=}\lambda j.\ \exists i.\ H\ i\ j$ **for** *H* **in** *Ex-first-occurrence*)
apply(*erule exE*)
apply *clarify*
— σ-$i \in A(pre,\ rely\text{-}1)$
apply(*subgoal-tac take* (*Suc* (*Suc m*)) (*clist*!*i*) $\in$ *assum*(*Pre*(*xs*!*i*), *Rely*(*xs*!*i*)))
— but this contradicts $\models \sigma$-i *sat* [*pre-i*,*rely-i*,*guar-i*,*post-i*]
apply(*erule-tac x=i* **and** $P{=}\lambda i.\ H\ i \longrightarrow \models (J\ i)\ sat\ [I\ i,K\ i,M\ i,N\ i]$ **for** *H J I K M N* **in** *allE*,*erule impE*,*assumption*)
apply(*simp add*:*com-validity-def*)
apply(*erule-tac x=s* **in** *allE*)
apply(*simp add*:*cp-def comm-def*)
apply(*drule-tac c=take* (*Suc* (*Suc m*)) (*clist* ! *i*) **in** *subsetD*)

apply *simp*
apply (*blast intro*: *takecptn-is-cptn*)
apply *simp*
apply *clarify*
apply(*erule-tac x=m* **and** *P*=λj. *I j* $\wedge$ *J j* $\longrightarrow$ *H j* **for** *I J H* **in** *allE*)
apply (*simp add*:*conjoin-def same-length-def*)
apply(*simp add*:*assum-def*)
apply(*rule conjI*)
apply(*erule-tac x=i* **and** *P*=λj. *H j* $\longrightarrow$ *I j* $\in$*cp* (*K j*) (*J j*) **for** *H I K J* **in** *allE*)
apply(*simp add*:*cp-def par-assum-def*)
apply(*drule-tac c=s* **in** *subsetD*,*simp*)
apply *simp*
apply *clarify*
apply(*erule-tac x=i* **and** *P*=λj. *H j* $\longrightarrow$ *M* $\cup$ $\bigcup$((*T j*) ‘ (*S j*)) $\subseteq$ (*L j*) **for** *H M S T L* **in** *allE*)
apply *simp*
apply(*erule subsetD*)
apply *simp*
apply(*simp add*:*conjoin-def compat-label-def*)
apply *clarify*
apply(*erule-tac x=ia* **and** *P*=λj. *H j* $\longrightarrow$ (*P j*) $\vee$ *Q j* **for** *H P Q* **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*=λi. *H i* $\longrightarrow$ (*I i*) $\vee$ (*J i*) **for** *H I J* **in** *allE*)
apply (*erule etranE*)
apply(*case-tac ia=m*,*simp*)
apply *simp*
apply(*erule-tac x=ia* **and** *P*=λj. *H j* $\longrightarrow$ ($\forall$ *i*. *P i j*) **for** *H P* **in** *allE*)
apply(*subgoal-tac ia*<*m*,*simp*)
prefer *2*
apply *arith*
apply(*erule-tac x=ib* **and** *P*=λj. (*I j*, *H j*) $\in$ *ctran* $\longrightarrow$ *P i j* **for** *I H P* **in** *allE*,*simp*)
apply(*simp add*:*same-state-def*)
apply(*erule-tac x=i* **and** *P*=λj. (*T j*) $\longrightarrow$ ($\forall$ *i*. (*H j i*) $\longrightarrow$ (*snd* (*d j i*))=(*snd* (*e j i*))) **for** *T H d e* **in** *all-dupE*)
apply(*erule-tac x=ib* **and** *P*=λj. (*T j*) $\longrightarrow$ ($\forall$ *i*. (*H j i*) $\longrightarrow$ (*snd* (*d j i*))=(*snd* (*e j i*))) **for** *T H d e* **in** *allE*,*simp*)
— or an e-tran in σ, therefore it satisfies *rely* $\vee$ *guar*-{*ib*}
apply (*force simp add*:*par-assum-def same-state-def*)
done

lemma *three* [*rule-format*]:
⟦ *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*);
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*)];
length xs=*length clist*; *x* $\in$ *par-cp* (*ParallelCom xs*) *s*; *x* $\in$ *par-assum*(*pre*, *rely*);
$\forall$ *i*<*length clist*. *clist*!*i*$\in$*cp* (*Some*(*Com*(*xs*!*i*))) *s*; *x* $\propto$ *clist* $]\!]$
$\Longrightarrow$ $\forall$ *j i*. *i*<*length clist* $\wedge$ *Suc j*<*length x* $\longrightarrow$ (*clist*!*i*!*j*) $-e\rightarrow$ (*clist*!*i*!*Suc j*)
$\longrightarrow$ (*snd*(*clist*!*i*!*j*), *snd*(*clist*!*i*!*Suc j*)) $\in$ *rely* $\cup$ ($\bigcup$ *j*$\in${*j*. *j* < *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*=λ*j*. *H j* $\longrightarrow$ (*J j* $\wedge$ ($\exists$ *i*. *P i j*)) $\vee$ *I j* **for** *H J P I* **in** *allE*,*simp*)
apply(*erule disjE*)
prefer *2*
apply(*force simp add*:*same-state-def par-assum-def*)
apply *clarify*
apply(*case-tac i*=*ia*,*simp*)
apply(*erule etranE*,*simp*)
apply(*erule-tac x*=*ia* **and** *P*=λ*i*. *H i* $\longrightarrow$ (*I i*) $\vee$ (*J i*) **for** *H I J* **in** *allE*,*simp*)
apply(*erule-tac x*=*j* **and** *P*=λ*j*. $\forall$ *i*. *S j i* $\longrightarrow$ (*I j i*, *H j i*) $\in$ *ctran* $\longrightarrow$ *P i j* **for** *S I H P* **in** *allE*)
apply(*erule-tac x*=*ia* **and** *P*=λ*j*. *S j* $\longrightarrow$ (*I j*, *H j*) $\in$ *ctran* $\longrightarrow$ *P j* **for** *S I H P* **in** *allE*)
apply(*simp add*:*same-state-def*)
apply(*erule-tac x*=*i* **and** *P*=λ*j*. *T j* $\longrightarrow$ ($\forall$ *i*. *H j i* $\longrightarrow$ (*snd* (*d j i*))=(*snd* (*e j i*))) **for** *T H d e* **in** *all-dupE*)
apply(*erule-tac x*=*ia* **and** *P*=λ*j*. *T j* $\longrightarrow$ ($\forall$ *i*. *H j i* $\longrightarrow$ (*snd* (*d j i*))=(*snd* (*e j i*))) **for** *T H d e* **in** *allE*,*simp*)
done

lemma *four*:
$[\![$*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-cp* (*ParallelCom xs*) *s*; *x* $\in$ *par-assum* (*pre*, *rely*); *Suc i* < *length x*;
x ! *i* $-pc\rightarrow$ *x* ! *Suc i*$]\!]$
$\Longrightarrow$ (*snd* (*x* ! *i*), *snd* (*x* ! *Suc i*)) $\in$ *guar*
apply(*simp add*: *ParallelCom-def*)
apply(*subgoal-tac* (*map* (*Some* $\circ$ *fst*) *xs*)$\neq$[])
prefer *2*
apply *simp*
apply(*frule rev-subsetD*)

apply(*erule one* [*THEN equalityD1*])
apply(*erule subsetD*)
apply *simp*
apply *clarify*
apply(*drule-tac pre=pre* **and** *rely=rely* **and** *x=x* **and** *s=s* **and** *xs=xs* **and** *clist=clist* **in** *two*)
apply(*assumption+*)
apply(*erule sym*)
apply(*simp add:ParallelCom-def*)
apply *assumption*
apply(*simp add:Com-def*)
apply *assumption*
apply(*simp add:conjoin-def same-program-def*)
apply *clarify*
apply(*erule-tac x=i* **and** *P=λj. H j ⟶ fst(I j)=(J j)* **for** *H I J* **in** *all-dupE*)
apply(*erule-tac x=Suc i* **and** *P=λj. H j ⟶ fst(I j)=(J j)* **for** *H I J* **in** *allE*)
apply(*erule par-ctranE,simp*)
apply(*erule-tac x=i* **and** *P=λj. ∀ i. S j i ⟶ (I j i, H j i) ∈ ctran ⟶ P i j* **for** *S I H P* **in** *allE*)
apply(*erule-tac x=ia* **and** *P=λj. S j ⟶ (I j, H j) ∈ ctran ⟶ P j* **for** *S I H P* **in** *allE*)
apply(*rule-tac x=ia* **in** *exI*)
apply(*simp add:same-state-def*)
apply(*erule-tac x=ia* **and** *P=λj. T j ⟶ (∀ i. H j i ⟶ (snd (d j i))=(snd (e j i)))* **for** *T H d e* **in** *all-dupE,simp*)
apply(*erule-tac x=ia* **and** *P=λj. T j ⟶ (∀ i. H j i ⟶ (snd (d j i))=(snd (e j i)))* **for** *T H d e* **in** *allE,simp*)
apply(*erule-tac x=i* **and** *P=λj. H j ⟶ (snd (d j))=(snd (e j))* **for** *H d e* **in** *all-dupE*)
apply(*erule-tac x=i* **and** *P=λj. H j ⟶ (snd (d j))=(snd (e j))* **for** *H d e* **in** *all-dupE,simp*)
apply(*erule-tac x=Suc i* **and** *P=λj. H j ⟶ (snd (d j))=(snd (e j))* **for** *H d e* **in** *allE,simp*)
apply(*erule mp*)
apply(*subgoal-tac r=fst(clist ! ia ! Suc i),simp*)
apply(*drule-tac i=ia* **in** *list-eq-if*)
back
apply *simp-all*
done

lemma *parcptn-not-empty* [*simp*]:[] ∉ *par-cptn*
apply(*force elim:par-cptn.cases*)
done

lemma *five*:
⟦*xs≠*[]; ∀ *i<length xs. rely* ∪ (⋃*j*∈{*j. j < length xs ∧ j ≠ i*}*. Guar* (*xs ! j*))
⊆ *Rely* (*xs ! i*);
pre ⊆ (⋂*i*∈{*i. i < length xs*}*. Pre* (*xs ! i*));
(⋂*i*∈{*i. i < length xs*}*. Post* (*xs ! i*)) ⊆ *post*;

$\forall i <$ *length xs.*
$\models$ *Com* (*xs* ! *i*) *sat* [*Pre* (*xs* ! *i*), *Rely* (*xs* ! *i*), *Guar* (*xs* ! *i*), *Post* (*xs* ! *i*)];
x $\in$ *par-cp* (*ParallelCom xs*) *s*; *x* $\in$ *par-assum* (*pre*, *rely*);
All-None (*fst* (*last x*)) $\rrbracket \Longrightarrow$ *snd* (*last x*) $\in$ *post*
apply(*simp add*: *ParallelCom-def*)
apply(*subgoal-tac* (*map* (*Some* $\circ$ *fst*) *xs*)$\neq$[])
prefer *2*
apply *simp*
apply(*frule rev-subsetD*)
apply(*erule one* [*THEN equalityD1*])
apply(*erule subsetD*)
apply *simp*
apply *clarify*
apply(*subgoal-tac* $\forall i<$*length clist*. *clist*!*i*$\in$*assum*(*Pre*(*xs*!*i*), *Rely*(*xs*!*i*)))
apply(*erule-tac x*=*xa* **and** *P*=λi. *H i* $\longrightarrow \models$ (*J i*) *sat* [*I i*,*K i*,*M i*,*N i*] **for** *H J I K M N* **in** *allE*,*erule impE*,*assumption*)
apply(*simp add*:*com-validity-def*)
apply(*erule-tac x*=*s* **in** *allE*)
apply(*erule-tac x*=*xa* **and** *P*=λj. *H j* $\longrightarrow$ (*I j*) $\in$ *cp* (*J j*) *s* **for** *H I J* **in** *allE*,*simp*)
apply(*drule-tac c*=*clist*!*xa* **in** *subsetD*)
apply (*force simp add*:*Com-def*)
apply(*simp add*:*comm-def conjoin-def same-program-def del*:*last.simps*)
apply *clarify*
apply(*erule-tac x*=*length x* $-$ *1* **and** *P*=λj. *H j* $\longrightarrow$ *fst*(*I j*)=(*J j*) **for** *H I J* **in** *allE*)
apply (*simp add*:*All-None-def same-length-def*)
apply(*erule-tac x*=*xa* **and** *P*=λj. *H j* $\longrightarrow$ *length*(*J j*)=(*K j*) **for** *H J K* **in** *allE*)
apply(*subgoal-tac length x* $-$ *1* $<$ *length x*,*simp*)
apply(*case-tac x*$\neq$[])
apply(*simp add*: *last-conv-nth*)
apply(*erule-tac x*=*clist*!*xa* **in** *ballE*)
apply(*simp add*:*same-state-def*)
apply(*subgoal-tac clist*!*xa*$\neq$[])
apply(*simp add*: *last-conv-nth*)
apply(*case-tac x*)
apply (*force simp add*:*par-cp-def*)
apply (*force simp add*:*par-cp-def*)
apply *force*
apply (*force simp add*:*par-cp-def*)
apply(*case-tac x*)
apply (*force simp add*:*par-cp-def*)
apply (*force simp add*:*par-cp-def*)
apply *clarify*
apply(*simp add*:*assum-def*)
apply(*rule conjI*)
apply(*simp add*:*conjoin-def same-state-def par-cp-def*)
apply *clarify*
apply(*erule-tac x*=*i* **and** *P*=λj. *T j* $\longrightarrow$ ($\forall i$. *H j i* $\longrightarrow$ (*snd* (*d j i*))=(*snd* (*e j i*))) **for** *T H d e* **in** *allE*,*simp*)

apply(*erule-tac x=0* **and** *P=λj. H j ⟶ (snd (d j))=(snd (e j))* **for** *H d e* **in** *allE*)
apply(*case-tac x*,*simp+*)
apply (*simp add*:*par-assum-def*)
apply *clarify*
apply(*drule-tac c=snd (clist ! i ! 0)* **in** *subsetD*)
apply *assumption*
apply *simp*
apply *clarify*
apply(*erule-tac x=i* **in** *all-dupE*)
apply(*rule subsetD*, *erule mp*, *assumption*)
apply(*erule-tac pre=pre* **and** *rely=rely* **and** *x=x* **and** *s=s* **in** *three*)
apply(*erule-tac x=ib* **in** *allE*,*erule mp*)
apply *simp-all*
apply(*simp add*:*ParallelCom-def*)
apply(*force simp add*:*Com-def*)
apply(*simp add*:*conjoin-def same-length-def*)
done

lemma *ParallelEmpty* [*rule-format*]:
∀ *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*:
⊢ *c SAT* [*pre*, *rely*, *guar*, *post*] ⟹
⊨ (*ParallelCom c*) *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*≠[])
prefer *2*
apply *simp*
apply(*rename-tac a list*)
apply(*thin-tac xs* = *a* # *list*)
apply(*simp add*:*par-com-validity-def par-comm-def*)
apply *clarify*
apply(*rule conjI*)
apply *clarify*
apply(*erule-tac pre*=*pre* **and** *rely*=*rely* **and** *guar*=*guar* **and** *x*=*x* **and** *s*=*s* **and** *xs*=*xs* **in** *four*)
apply(*assumption*+)
apply *clarify*
apply (*erule allE*, *erule impE*, *assumption*,*erule rgsound*)
apply(*assumption*+)
apply *clarify*
apply(*erule-tac pre*=*pre* **and** *rely*=*rely* **and** *post*=*post* **and** *x*=*x* **and** *s*=*s* **and** *xs*=*xs* **in** *five*)
apply(*assumption*+)
apply *clarify*
apply (*erule allE*, *erule impE*, *assumption*,*erule rgsound*)
apply(*assumption*+)
done

end

3.6 Concrete Syntax

theory *RG-Syntax*
imports *RG-Hoare Quote-Antiquote*
begin

abbreviation *Skip* :: *'a com* (‹*SKIP*›)
where *SKIP* ≡ *Basic id*

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

syntax
-Assign :: *idt* ⇒ *'b* ⇒ *'a com* (‹(´- :=/ -)› [*70*, *65*] *61*)
-Cond :: *'a bexp* ⇒ *'a com* ⇒ *'a com* ⇒ *'a com* (‹(*0IF* -/ *THEN* -/ *ELSE* -/*FI*)› [*0*, *0*, *0*] *61*)
-Cond2 :: *'a bexp* ⇒ *'a com* ⇒ *'a com* (‹(*0IF* - *THEN* - *FI*)› [*0*,*0*] *56*)
-While :: *'a bexp* ⇒ *'a com* ⇒ *'a com* (‹(*0WHILE* - /*DO* - /*OD*)› [*0*, *0*] *61*)
-Await :: *'a bexp* ⇒ *'a com* ⇒ *'a com* (‹(*0AWAIT* - /*THEN* /-

/END)› [0,0] 61)
-Atom :: 'a com ⇒ 'a com (‹(⟨-⟩)› 61)
-Wait :: 'a bexp ⇒ 'a com (‹(0WAIT - END)› 61)

translations
´x := a ⇀ CONST Basic «´(-update-name x (λ-. a))»
IF b THEN c1 ELSE c2 FI ⇀ CONST Cond {|b|} c1 c2
IF b THEN c FI ⇌ IF b THEN c ELSE SKIP FI
WHILE b DO c OD ⇀ CONST While {|b|} c
AWAIT b THEN c END ⇌ CONST Await {|b|} c
⟨c⟩ ⇌ AWAIT CONST True THEN c END
WAIT b END ⇌ AWAIT b THEN SKIP END

nonterminal *prgs*

syntax
-PAR :: prgs ⇒ 'a (‹COBEGIN//-//COEND› 60)
-prg :: 'a ⇒ prgs (‹-› 57)
-prgs :: ['a, prgs] ⇒ prgs (‹-//‖//-› [60,57] 57)

translations
-prg a ⇀ [a]
-prgs a ps ⇀ a # ps
-PAR ps ⇀ ps

syntax
-prg-scheme :: ['a, 'a, 'a, 'a] ⇒ prgs (‹SCHEME [- ≤ - < -] -› [0,0,0,60] 57)

translations
-prg-scheme j i k c ⇌ (CONST map (λi. c) [j..<k])

Translations for variables before and after a transition:

syntax
-before :: id ⇒ 'a (‹º-›)
-after :: id ⇒ 'a (‹ª-›)

translations
ºx ⇌ x ´CONST fst
ªx ⇌ x ´CONST snd

print-translation ‹
let
fun quote-tr' f (t :: ts) =
Term.list-comb (f $ Syntax-Trans.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 assign-tr′ (*Abs* (*x*, -, *f* $ *k* $ *Bound 0*) :: *ts*) =
quote-tr′ (*Syntax.const* ***syntax-const*** ‹*-Assign*› $ *Syntax-Trans.update-name-tr′ f*)
(*Abs* (*x*, *dummyT*, *Syntax-Trans.const-abs-tr′ k*) :: *ts*)
| *assign-tr′* - = *raise Match*;
in
[(***const-syntax*** ‹*Collect*›, *K assert-tr′*),
(***const-syntax*** ‹*Basic*›, *K assign-tr′*),
(***const-syntax*** ‹*Cond*›, *K* (*bexp-tr′* ***syntax-const*** ‹*-Cond*›)),
(***const-syntax*** ‹*While*›, *K* (*bexp-tr′* ***syntax-const*** ‹*-While*›))]
end
›

end

3.7 Examples

theory *RG-Examples*
imports *RG-Syntax*
begin

lemmas *definitions* [*simp*]= *stable-def Pre-def Rely-def Guar-def Post-def Com-def*

3.7.1 Set Elements of an Array to Zero

lemma *le-less-trans2*: $[\![(j::nat)<k;\ i\leq j]\!] \Longrightarrow i<k$
by *simp*

lemma *add-le-less-mono*: $[\![\ (a::nat) < c;\ b\leq d\ ]\!] \Longrightarrow a + b < c + d$
by *simp*

record *Example1* =
A :: *nat list*

lemma *Example1*:
⊢ *COBEGIN*
SCHEME [*0* ≤ *i* < *n*]
(´*A* := ´*A* [*i* := *0*],
⦃ *n* < *length* ´*A* ⦄,
⦃ *length* ºA = *length* ªA ∧ ºA ! *i* = ªA ! *i* ⦄,
⦃ *length* ºA = *length* ªA ∧ (∀ *j*<*n*. *i* ≠ *j* ⟶ ºA ! *j* = ªA ! *j*) ⦄,
⦃ ´*A* ! *i* = *0* ⦄)
COEND
SAT [⦃ *n* < *length* ´*A* ⦄, ⦃ ºA = ªA ⦄, ⦃ *True* ⦄, ⦃ ∀ *i* < *n*. ´*A* ! *i* = *0* ⦄]
apply(*rule Parallel*)

apply (*auto intro*!: *Basic*)
done

lemma *Example1-parameterized*:
$k < t \Longrightarrow$
$\vdash$ *COBEGIN*
SCHEME $[k*n \le i < (Suc\ k)*n]$ $(\acute{}A:=\acute{}A[i:=0],$
$\{\!| t*n < length\ \acute{}A |\!\}$,
$\{\!| t*n < length\ {}^{o}A \wedge length\ {}^{o}A = length\ {}^{a}A \wedge {}^{o}A!i = {}^{a}A!i |\!\}$,
$\{\!| t*n < length\ {}^{o}A \wedge length\ {}^{o}A = length\ {}^{a}A \wedge (\forall j < length\ {}^{o}A\ .\ i \neq j \longrightarrow {}^{o}A!j =$
${}^{a}A!j) |\!\}$,
$\{\!| \acute{}A!i = 0 |\!\})$
COEND
SAT $[\{\!| t*n < length\ \acute{}A |\!\}$,
$\{\!| t*n < length\ {}^{o}A \wedge length\ {}^{o}A = length\ {}^{a}A \wedge (\forall i < n.\ {}^{o}A!(k*n+i) = {}^{a}A!(k*n+i)) |\!\}$,
$\{\!| t*n < length\ {}^{o}A \wedge length\ {}^{o}A = length\ {}^{a}A \wedge$
$(\forall i < length\ {}^{o}A\ .\ (i < k*n \longrightarrow {}^{o}A!i = {}^{a}A!i) \wedge ((Suc\ k)*n \le i \longrightarrow {}^{o}A!i = {}^{a}A!i)) |\!\}$,
$\{\!| \forall i < n.\ \acute{}A!(k*n+i) = 0 |\!\}]$
apply(*rule Parallel*)
apply *auto*
apply(*erule-tac* $x=k*n+i$ **in** *allE*)
apply(*subgoal-tac* $k*n+i < length\ (A\ b)$)
apply *force*
apply(*erule le-less-trans2*)
apply(*case-tac t*,*simp*+)
apply (*simp add*:*add.commute*)
apply(*simp add*: *add-le-mono*)
apply(*rule Basic*)
apply *simp*
apply *clarify*
apply (*subgoal-tac* $k*n+i < length\ (A\ x)$)
apply *simp*
apply(*erule le-less-trans2*)
apply(*case-tac t*,*simp*+)
apply (*simp add*:*add.commute*)
apply(*rule add-le-mono*, *auto*)
done

3.7.2 Increment a Variable in Parallel

Two components

record *Example2* =
x :: *nat*
c-0 :: *nat*
c-1 :: *nat*

lemma *Example2*:
$\vdash$ *COBEGIN*
$(\langle\ \acute{}x:=\acute{}x+1;;\ \acute{}c\text{-}0:=\acute{}c\text{-}0 + 1\ \rangle,$

```
    {|´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 simp
   apply(rule conjI)
    apply clarify
    apply simp
   apply clarify
   apply simp
   apply(subgoal-tac x=0)
    apply simp
   apply arith
  apply clarify
  apply(case-tac xa, 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*: $j<n \Longrightarrow$
 $(\sum i=0..<n.\ (b\ i::nat)) =$
 $(\sum i=0..<j.\ b\ i) + b\ j + (\sum i=0..<n-(Suc\ j)\ .\ b\ (Suc\ j + i))$
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*:
 $j\leq s \Longrightarrow (\sum i::nat=0..<j.\ (b\ (s:=t))\ i) = (\sum i=0..<j.\ b\ i)$
 by (*induct j*) *simp-all*

lemma *Example2-lemma2*:
 $[\![j<n;\ b\ j=0]\!] \Longrightarrow Suc\ (\sum i::nat=0..<n.\ b\ i)=(\sum i=0..<n.\ (b\ (j := Suc\ 0))\ i)$
apply(*frule-tac* $b=(b\ (j:=(Suc\ 0)))$ **in** *Example2-lemma2-aux*)
apply(*erule-tac* $t=sum\ (b(j := (Suc\ 0)))\ \{0..<n\}$ **in** *ssubst*)
apply(*frule-tac* $b=b$ **in** *Example2-lemma2-aux*)
apply(*erule-tac* $t=sum\ b\ \{0..<n\}$ **in** *ssubst*)
apply(*subgoal-tac* $Suc\ (sum\ b\ \{0..<j\} + b\ j + (\sum i=0..<n - Suc\ j.\ b\ (Suc\ j +$

$i)))=(sum\ b\ \{0..<j\} + Suc\ (b\ j) + (\sum i=0..<n - Suc\ j.\ b\ (Suc\ j + i))))$
apply(*rotate-tac* −*1*)
apply(*erule ssubst*)
apply(*subgoal-tac* $j \le j$)
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*: $[\![j<n;\ b\ j=0]\!] \Longrightarrow$
$Suc\ (\sum i::nat=0..<\ n.\ b\ i)=(\sum i=0..<\ n.\ (b\ (j:=Suc\ 0))\ i)$
by(*simp add*:*Example2-lemma2*)

record *Example2-parameterized* =
  $C :: nat \Rightarrow nat$
  $y :: nat$

lemma *Example2-parameterized*: $0<n \Longrightarrow$
 $\vdash$ *COBEGIN SCHEME* $[0 \le i < n]$
   $(\langle\ \acute{}y:=\acute{}y+1;;\ \acute{}C:=\acute{}C\ (i:=1)\ \rangle,$
   $\{\!|\acute{}y=(\sum i=0..<n.\ \acute{}C\ i) \wedge \acute{}C\ i=0|\!\},$
   $\{\!|{}^{o}C\ i = {}^{a}C\ i \wedge$
    $({}^{o}y=(\sum i=0..<n.\ {}^{o}C\ i) \longrightarrow {}^{a}y =(\sum i=0..<n.\ {}^{a}C\ i))|\!\},$
   $\{\!|(\forall j<n.\ i\neq j \longrightarrow {}^{o}C\ j = {}^{a}C\ j) \wedge$
    $({}^{o}y=(\sum i=0..<n.\ {}^{o}C\ i) \longrightarrow {}^{a}y =(\sum i=0..<n.\ {}^{a}C\ i))|\!\},$
   $\{\!|\acute{}y=(\sum i=0..<n.\ \acute{}C\ i) \wedge \acute{}C\ i=1|\!\})$
  COEND
 SAT $[\{\!|\acute{}y=0 \wedge (\sum i=0..<n.\ \acute{}C\ i)=0\ |\!\},\ \{\!|{}^{o}C={}^{a}C \wedge {}^{o}y={}^{a}y|\!\},\ \{\!|True|\!\},\ \{\!|\acute{}y=n|\!\}]$
apply(*rule Parallel*)
apply *force*
apply *force*
apply(*force*)
apply *clarify*
apply *simp*
apply *simp*
apply *clarify*
apply *simp*
apply(*rule Await*)
apply *simp-all*
apply *clarify*
apply(*rule Seq*)
prefer *2*
apply(*rule Basic*)
apply(*rule subset-refl*)
apply *simp+*
apply(*rule Basic*)
apply *simp*
apply *clarify*

apply *simp*
apply(*simp add*:*Example2-lemma2-Suc0 cong*:*if-cong*)
apply *simp-all*
done

3.7.3 Find Least Element

A previous lemma:

lemma *mod-aux* :$[\![i < (n::nat);\ a\ mod\ n = i;\ \ j < a + n;\ j\ mod\ n = i;\ a < j]\!] \Longrightarrow$ *False*
apply(*subgoal-tac a=a div n*n + a mod n*)
 prefer *2* **apply** (*simp* (*no-asm-use*))
apply(*subgoal-tac j=j div n*n + j mod n*)
 prefer *2* **apply** (*simp* (*no-asm-use*))
apply *simp*
apply(*subgoal-tac a div n*n < j div n*n*)
prefer *2* **apply** *arith*
apply(*subgoal-tac j div n*n < (a div n + 1)*n*)
prefer *2* **apply** *simp*
apply (*simp only*:*mult-less-cancel2*)
apply *arith*
done

record *Example3* =
 X :: *nat* ⇒ *nat*
 Y :: *nat* ⇒ *nat*

lemma *Example3*: *m mod n=0* ⟹
⊢ *COBEGIN*
 SCHEME [*0*≤*i*<*n*]
 (*WHILE* (∀ *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*) *mod n=i* ∧ (∀ *j*<´*X i*. *j mod n=i* ⟶ ¬*P*(*B*!*j*)) ∧ (´*Y i*<*m* ⟶ *P*(*B*!(´*Y i*)) ∧ ´*Y i*≤ *m*+*i*)⦄,
 ⦃(∀ *j*<*n*. *i*≠*j* ⟶ ª*Y j* ≤ º*Y j*) ∧ º*X i* = ª*X i* ∧
  º*Y i* = ª*Y i*⦄,
 ⦃(∀ *j*<*n*. *i*≠*j* ⟶ º*X j* = ª*X j* ∧ º*Y j* = ª*Y j*) ∧
  ª*Y i* ≤ º*Y i*⦄,
 ⦃(´*X i*) *mod n=i* ∧ (∀ *j*<´*X i*. *j mod n=i* ⟶ ¬*P*(*B*!*j*)) ∧ (´*Y i*<*m* ⟶ *P*(*B*!(´*Y i*)) ∧ ´*Y i*≤ *m*+*i*) ∧ (∃ *j*<*n*. ´*Y j* ≤ ´*X i*) ⦄)
 COEND
 SAT [⦃ ∀ *i*<*n*. ´*X i*=*i* ∧ ´*Y i*=*m*+*i* ⦄,⦃º*X*=ª*X* ∧ º*Y*=ª*Y*⦄,⦃*True*⦄,
 ⦃∀ *i*<*n*. (´*X i*) *mod n=i* ∧ (∀ *j*<´*X i*. *j mod n=i* ⟶ ¬*P*(*B*!*j*)) ∧
  (´*Y i*<*m* ⟶ *P*(*B*!(´*Y i*)) ∧ ´*Y i*≤ *m*+*i*) ∧ (∃ *j*<*n*. ´*Y j* ≤ ´*X i*)⦄]
apply(*rule Parallel*)
— 5 subgoals left
apply *force*+

apply *clarify*
apply *simp*
apply(*rule While*)
apply *force*
apply *force*
apply *force*
apply (*erule dvdE*)
apply(*rule-tac pre'*=$\{\!|$ ´*X i mod n* = *i* ∧ (∀ *j*. *j*<´*X i* ⟶ *j mod n* = *i* ⟶ ¬*P*(*B*!*j*)) ∧ (´*Y i* < *n* ∗ *k* ⟶ *P* (*B*!(´*Y i*))) ∧ ´*X i*<´*Y i*$|\!\}$ **in** *Conseq*)
apply *force*
apply(*rule subset-refl*)+
apply(*rule Cond*)
apply *force*
apply(*rule Basic*)
apply *force*
apply *fastforce*
apply *force*
apply *force*
apply(*rule Basic*)
apply *simp*
apply *clarify*
apply *simp*
apply (*case-tac X x* (*j mod n*) ≤ *j*)
apply (*drule le-imp-less-or-eq*)
apply (*erule disjE*)
apply (*drule-tac j*=*j* **and** *n*=*n* **and** *i*=*j mod n* **and** *a*=*X x* (*j mod n*) **in** *mod-aux*)
apply *auto*
done

Same but with a list as auxiliary variable:

record *Example3-list* =
X :: *nat list*
Y :: *nat list*

lemma *Example3-list*: *m mod n*=*0* ⟹ ⊢ (*COBEGIN SCHEME* [*0*≤*i*<*n*]
(*WHILE* (∀ *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*<*length* ´*X* ∧ *n*<*length* ´*Y* ∧ (´*X*!*i*) *mod n*=*i* ∧ (∀ *j*<´*X*!*i*. *j mod n*=*i* ⟶ ¬*P*(*B*!*j*)) ∧ (´*Y*!*i*<*m* ⟶ *P*(*B*!(´*Y*!*i*)) ∧ ´*Y*!*i*≤ *m*+*i*)$|\!\}$,
$\{\!|$(∀ *j*<*n*. *i*≠*j* ⟶ ª*Y*!*j* ≤ º*Y*!*j*) ∧ º*X*!*i* = ª*X*!*i* ∧
º*Y*!*i* = ª*Y*!*i* ∧ *length* º*X* = *length* ª*X* ∧ *length* º*Y* = *length* ª*Y*$|\!\}$,
$\{\!|$(∀ *j*<*n*. *i*≠*j* ⟶ º*X*!*j* = ª*X*!*j* ∧ º*Y*!*j* = ª*Y*!*j*) ∧
ª*Y*!*i* ≤ º*Y*!*i* ∧ *length* º*X* = *length* ª*X* ∧ *length* º*Y* = *length* ª*Y*$|\!\}$,
$\{\!|$(´*X*!*i*) *mod n*=*i* ∧ (∀ *j*<´*X*!*i*. *j mod n*=*i* ⟶ ¬*P*(*B*!*j*)) ∧ (´*Y*!*i*<*m* ⟶ *P*(*B*!(´*Y*!*i*)) ∧ ´*Y*!*i*≤ *m*+*i*) ∧ (∃ *j*<*n*. ´*Y*!*j* ≤ ´*X*!*i*) $|\!\}$) *COEND*)
SAT [$\{\!|$*n*<*length* ´*X* ∧ *n*<*length* ´*Y* ∧ (∀ *i*<*n*. ´*X*!*i*=*i* ∧ ´*Y*!*i*=*m*+*i*) $|\!\}$,
$\{\!|$º*X*=ª*X* ∧ º*Y*=ª*Y*$|\!\}$,

```
      {|True|},
      {|∀ i<n. (´X!i) mod n=i ∧ (∀ j<´X!i. j mod n=i ⟶ ¬P(B!j)) ∧
        (´Y!i<m ⟶ P(B!(´Y!i)) ∧ ´Y!i≤ m+i) ∧ (∃ j<n. ´Y!j ≤ ´X!i)|}]
  apply (rule Parallel)
apply (auto cong del: image-cong-simp)
apply force
apply (rule While)
    apply force
   apply force
  apply force
  apply (erule dvdE)
 apply(rule-tac pre'={|n<length ´X ∧ n<length ´Y ∧ ´X ! i mod n = i ∧ (∀ j. j
< ´X ! i ⟶ j mod n = i ⟶ ¬ P (B ! j)) ∧ (´Y ! i < n * k ⟶ P (B ! (´Y !
i))) ∧ ´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.