Complx: a Verification Framework for Concurrent Imperative Programs

Sidney Amani, June Andronick, Maksym Bortin,
Corey Lewis, Christine Rizkallah, Joseph Tuong

March 17, 2025

Abstract

We propose a concurrency reasoning framework for imperative programs, based on the Owicki-Gries (OG) foundational shared-variable concurrency method. Our framework combines the approaches of Hoare-Parallel, a formalisation of OG in Isabelle/HOL for a simple while-language, and Simpl, a generic imperative language embedded in Isabelle/HOL, allowing formal reasoning on C programs.

We define the Complx language, extending the syntax and semantics of Simpl with support for parallel composition and synchronisation. We additionally define an OG logic, which we prove sound w.r.t. the semantics, and a verification condition generator, both supporting involved low-level imperative constructs such as function calls and abrupt termination. We illustrate our framework on an example that features exceptions, guards and function calls. We aim to then target concurrent operating systems, such as the interruptible eChronos embedded operating system for which we already have a model-level OG proof using Hoare-Parallel.

Contents

1 The COMPLX Syntax

theory Language
imports Main
begin

1.1 The Core Language

We use a shallow embedding of boolean expressions as well as assertions as sets of states.

```
type_synonym 's bexp = "'s set"
type_synonym 's assn = "'s set"

datatype ('s, 'p, 'f) com =
    Skip
  | Basic "'s ⇒ 's"
  | Spec "('s × 's) set"
  | Seq "('s ,'p, 'f) com" "('s,'p, 'f) com"
  | Cond "'s bexp" "('s,'p,'f) com"  "('s,'p,'f) com"
  | While "'s bexp" "('s,'p,'f) com"
  | Call "'p"
  | DynCom "'s ⇒ ('s,'p,'f) com"
  | Guard "'f" "'s bexp" "('s,'p,'f) com"
  | Throw
  | Catch "('s,'p,'f) com" "('s,'p,'f) com"
  | Parallel "('s, 'p, 'f) com list"
  | Await "'s bexp" "('s,'p,'f) com"
```

1.2 Derived Language Constructs

We inherit the remainder of derived language constructs from SIMPL

definition
```
  raise:: "('s ⇒ 's) ⇒ ('s,'p,'f) com" where
  "raise f = Seq (Basic f) Throw"
```

definition
```
  condCatch:: "('s,'p,'f) com ⇒ 's bexp ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f)
com" where
  "condCatch c1 b c2 = Catch c1 (Cond b c2 Throw)"
```

definition
```
  bind:: "('s ⇒ 'v) ⇒ ('v ⇒ ('s,'p,'f) com) ⇒ ('s,'p,'f) com" where
  "bind e c = DynCom (λs. c (e s))"
```

definition
```
  bseq:: "('s,'p,'f) com ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com" where
  "bseq = Seq"
```

definition
```
  block:: "['s⇒'s ,('s,'p,'f) com,'s⇒'s⇒'s,'s⇒'s⇒('s,'p,'f) com]⇒('s,'p,'f)
com"
where
  "block init bdy restore return =
    DynCom (λs. (Seq (Catch (Seq (Basic init) bdy) (Seq (Basic (restore
s)) Throw))
                             (DynCom (λt. Seq (Basic (restore s)) (return
s t))))
                        )"
```

definition
```
  call:: "('s⇒'s) ⇒ 'p ⇒ ('s ⇒ 's ⇒ 's)⇒('s⇒'s⇒('s,'p,'f) com)⇒('s,'p,'f)com"
where
  "call init p restore return = block init (Call p) restore return"
```

definition
```
  dynCall:: "('s⇒'s) ⇒ ('s ⇒ 'p) ⇒ ('s ⇒ 's ⇒ 's) ⇒
             ('s ⇒ 's ⇒ ('s,'p,'f) com) ⇒ ('s,'p,'f) com" where
  "dynCall init p restore return = DynCom (λs. call init (p s) restore
return)"
```

definition
```
  fcall:: "('s⇒'s) ⇒ 'p ⇒ ('s ⇒ 's ⇒ 's)⇒('s ⇒ 'v) ⇒ ('v⇒('s,'p,'f)
com)
            ⇒('s,'p,'f) com" where
  "fcall init p restore result return = call init p restore (λs t. return
(result t))"
```

definition
```
  lem:: "'x ⇒ ('s,'p,'f)com ⇒('s,'p,'f)com" where
  "lem x c = c"

primrec switch:: "('s ⇒ 'v) ⇒ ('v set × ('s,'p,'f) com) list ⇒ ('s,'p,'f)
com"
where
"switch v [] = Skip" |
"switch v (Vc#vs) = Cond {s. v s ∈ fst Vc} (snd Vc) (switch v vs)"

definition guaranteeStrip:: "'f ⇒ 's set ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f)
com"
  where "guaranteeStrip f g c = Guard f g c"

definition guaranteeStripPair:: "'f ⇒ 's set ⇒ ('f × 's set)"
  where "guaranteeStripPair f g = (f,g)"

primrec guards:: "('f × 's set ) list ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f)
com"
where
"guards [] c = c" |
"guards (g#gs) c = Guard (fst g) (snd g) (guards gs c)"

definition
  while::  "('f × 's set) list ⇒ 's bexp ⇒ ('s,'p,'f) com ⇒ ('s, 'p,
'f) com"
where
  "while gs b c = guards gs (While b (Seq c (guards gs Skip)))"

definition
  whileAnno::
  "'s bexp ⇒ 's assn ⇒ ('s × 's) assn ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f)
com" where
  "whileAnno b I V c = While b c"

definition
  whileAnnoG::
  "('f × 's set) list ⇒ 's bexp ⇒ 's assn ⇒ ('s × 's) assn ⇒
     ('s,'p,'f) com ⇒ ('s,'p,'f) com" where
  "whileAnnoG gs b I V c = while gs b c"

definition
  specAnno::  "('a ⇒ 's assn) ⇒ ('a ⇒ ('s,'p,'f) com) ⇒
                          ('a ⇒ 's assn) ⇒ ('a ⇒ 's assn) ⇒ ('s,'p,'f)
com"
  where "specAnno P c Q A = (c undefined)"

definition
  whileAnnoFix::
```

```
  "'s bexp ⇒ ('a ⇒ 's assn) ⇒ ('a ⇒ ('s × 's) assn) ⇒ ('a ⇒ ('s,'p,'f)
com) ⇒
     ('s,'p,'f) com" where
  "whileAnnoFix b I V c = While b (c undefined)"

definition
  whileAnnoGFix::
  "('f × 's set) list ⇒ 's bexp ⇒ ('a ⇒ 's assn) ⇒ ('a ⇒ ('s × 's)
assn) ⇒
     ('a ⇒ ('s,'p,'f) com) ⇒ ('s,'p,'f) com" where
  "whileAnnoGFix gs b I V c = while gs b (c undefined)"

definition if_rel::"('s ⇒ bool) ⇒ ('s ⇒ 's) ⇒ ('s ⇒ 's) ⇒ ('s ⇒ 's)
⇒ ('s × 's) set"
  where "if_rel b f g h = {(s,t). if b s then t = f s else t = g s ∨
t = h s}"

lemma fst_guaranteeStripPair: "fst (guaranteeStripPair f g) = f"
  by (simp add: guaranteeStripPair_def)

lemma snd_guaranteeStripPair: "snd (guaranteeStripPair f g) = g"
  by (simp add: guaranteeStripPair_def)

end
```

2 COMPLX small-step semantics

```
theory SmallStep
imports Language
begin
```

The procedure environment

```
type_synonym ('s,'p,'f) body = "'p ⇒ ('s,'p,'f) com option"
```

State types

```
datatype ('s,'f) xstate = Normal 's | Fault 'f | Stuck

lemma rtrancl_mono_proof[mono]:
   "(⋀a b. x a b ⟶ y a b) ⟹ rtranclp x a b ⟶ rtranclp y a b"
   apply (rule impI, rotate_tac, induct rule: rtranclp.induct)
    apply simp_all
   apply (metis rtranclp.intros)
   done


primrec redex:: "('s,'p,'f)com ⇒ ('s,'p,'f)com"
where
"redex Skip = Skip" |
```

```
"redex (Basic f) = (Basic f)" |
"redex (Spec r) = (Spec r)" |
"redex (Seq c₁ c₂) = redex c₁" |
"redex (Cond b c₁ c₂) = (Cond b c₁ c₂)" |
"redex (While b c) = (While b c)" |
"redex (Call p) = (Call p)" |
"redex (DynCom d) = (DynCom d)" |
"redex (Guard f b c) = (Guard f b c)" |
"redex (Throw) = Throw" |
"redex (Catch c₁ c₂) = redex c₁" |
"redex (Await b c) = Await b c" |
"redex (Parallel cs) = Parallel cs"
```

2.1 Small-Step Computation: Γ ⊢(c, s) → (c', s')

The final configuration is either of the form `(Skip,_)` for normal termination, or `(Throw, Normal s)` in case the program was started in a `Normal` state and terminated abruptly. Explicit abrupt states are removed from the language definition and thus do not need to be propogated.

type_synonym (’s,’p,’f) config = "(’s,’p,’f)com × (’s,’f) xstate"

definition final:: "(’s,’p,’f) config ⇒ bool" **where**
"final cfg = (fst cfg=Skip ∨ (fst cfg=Throw ∧ (∃s. snd cfg=Normal s)))"

primrec atom_com :: "~~(’s, ’p, ’f) com~~ ⇒ bool" **where**
```
  "atom_com Skip = True" |
  "atom_com (Basic f) = True" |
  "atom_com (Spec r) = 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 c) = atom_com c" |

  "atom_com (Call p) = False" |
  "atom_com (DynCom f) = (∀s::’s. atom_com (f s))" |
  "atom_com (Guard f g c) = atom_com c" |
  "atom_com Throw = True" |
  "atom_com (Catch c₁ c₂) = (atom_com c₁ ∧ atom_com c₂)" |
  "atom_com (Parallel cs) = False" |
  "atom_com (Await b c) = False"
```

inductive
```
      "step"::"[(’s,’p,’f) body, (’s,’p,’f) config,(’s,’p,’f) config]
⇒ bool"
                                (<_⊢ (_ →/ _)> [81,81,81] 100)
  and "step_rtrancl" :: "[(’s,’p,’f) body, (’s,’p,’f) config,(’s,’p,’f)
config] ⇒ bool"
                                (<_ ⊢ (_ →*/ _)> [81,81,81] 100)
```

for Γ::"('s,'p,'f) body"
where
"Γ ⊢ a →* b ≡ (step Γ)** a b"
| Basic: "Γ ⊢(Basic f,Normal s) → (Skip,Normal (f s))"

| Spec: "(s,t) ∈ r ⟹ Γ ⊢(Spec r,Normal s) → (Skip,Normal t)"
| SpecStuck: "∀t. (s,t) ∉ r ⟹ Γ ⊢(Spec r,Normal s) → (Skip,Stuck)"

| Guard: "s∈g ⟹ Γ ⊢(Guard f g c,Normal s) → (c,Normal s)"

| GuardFault: "s∉g ⟹ Γ ⊢(Guard f g c,Normal s) → (Skip,Fault f)"

| Seq: "Γ ⊢(c$_1$,s) → (c$_1$',s')
⟹
Γ ⊢(Seq c$_1$ c$_2$,s) → (Seq c$_1$' c$_2$, s')"
| SeqSkip: "Γ ⊢(Seq Skip c$_2$,s) → (c$_2$, s)"
| SeqThrow: "Γ ⊢(Seq Throw c$_2$,Normal s) → (Throw, Normal s)"

| CondTrue: "s∈b ⟹ Γ ⊢(Cond b c$_1$ c$_2$,Normal s) → (c$_1$,Normal s)"
| CondFalse: "s∉b ⟹ Γ ⊢(Cond b c$_1$ c$_2$,Normal s) → (c$_2$,Normal s)"

| WhileTrue: "⟦s∈b⟧
⟹
Γ ⊢(While b c,Normal s) → (Seq c (While b c),Normal s)"

| WhileFalse: "⟦s∉b⟧
⟹
Γ ⊢(While b c,Normal s) → (Skip,Normal s)"

| Call: "Γ p=Some b ⟹
Γ ⊢(Call p,Normal s) → (b,Normal s)"

| CallUndefined: "Γ p=None ⟹
Γ ⊢(Call p,Normal s) → (Skip,Stuck)"

| DynCom: "Γ ⊢(DynCom c,Normal s) → (c s,Normal s)"

| Catch: "⟦Γ ⊢(c$_1$,s) → (c$_1$',s')⟧
⟹
Γ ⊢(Catch c$_1$ c$_2$,s) → (Catch c$_1$' c$_2$,s')"

| CatchSkip: "Γ ⊢(Catch Skip c$_2$,s) → (Skip,s)"
| CatchThrow: "Γ ⊢(Catch Throw c$_2$,Normal s) → (c$_2$,Normal s)"

| FaultProp: "⟦c≠Skip; redex c = c⟧ ⟹ Γ ⊢(c,Fault f) → (Skip,Fault f)"
| StuckProp: "⟦c≠Skip; redex c = c⟧ ⟹ Γ ⊢(c,Stuck) → (Skip,Stuck)"

```
| Parallel: "⟦ i < length cs; Γ ⊢ (cs!i, s) → (c', s') ⟧
           ⟹ Γ ⊢ (Parallel cs, s) → (Parallel (cs[i := c']), s')"

| ParSkip: "⟦ ∀c ∈ set cs. c = Skip ⟧ ⟹ Γ ⊢ (Parallel cs, s) → (Skip,
s)"

| ParThrow: "⟦ Throw ∈ set cs ⟧ ⟹ Γ ⊢ (Parallel cs, s) → (Throw, s)"


| Await: "⟦ s ∈ b; Γ ⊢ (c, Normal s) →* (c', Normal s');
          atom_com c; c' = Skip ∨ c' = Throw ⟧
         ⟹ Γ ⊢ (Await b c, Normal s) → (c', Normal s')"
| AwaitStuck:
        "⟦ s ∈ b; Γ ⊢ (c, Normal s) →* (c', Stuck) ;
          atom_com c⟧
         ⟹ Γ ⊢ (Await b c, Normal s) → (Skip, Stuck)"
| AwaitFault:
        "⟦ s ∈ b; Γ ⊢ (c, Normal s) →* (c', Fault f) ;
          atom_com c⟧
         ⟹ Γ ⊢ (Await b c, Normal s) → (Skip, Fault f)"
| AwaitNonAtom:
        " ¬ atom_com c
         ⟹ Γ ⊢ (Await b c, Normal s) → (Skip, Stuck)"
```

lemmas step_induct = step.induct [of _ "(c,s)" "(c',s')", split_format (complete), case_names
Basic Spec SpecStuck Guard GuardFault Seq SeqSkip SeqThrow CondTrue CondFalse WhileTrue WhileFalse Call CallUndefined DynCom Catch CatchThrow CatchSkip FaultProp StuckProp Parallel ParSkip Await, induct set]

The execution of a command is blocked if it cannot make progress, but is not in a final state. It is the intention that ∃cfg. Γ ⊢ (c, s) → cfg) ∨ final (c, s) ∨ blocked Γ c s, but we do not prove this.

function(sequential) blocked :: "('s,'p,'f) body ⇒ ('s,'p,'f) com ⇒ ('s,'f)xstate ⇒ bool" **where**
  "blocked Γ (Seq (Await b c_1) c_2) (Normal s) = (s ∉ b)"
| "blocked Γ (Catch (Await b c_1) c_2) (Normal s) = (s ∉ b)"
| "blocked Γ (Await b c) (Normal s) = (s ∉ b ∨ (∀c' s'. Γ ⊢ (c, Normal s) →* (c', Normal s') ⟶ ¬ final (c', Normal s')))"
| "blocked Γ (Parallel cs) (Normal s) = (∀t ∈ set cs. blocked Γ t (Normal s) ∨ final (t, Normal s))"
| "blocked Γ _ _ = False"
by (pat_completeness, auto)

inductive_cases step_elim_cases [cases set]:
 "Γ ⊢(Skip,s) → u"
 "Γ ⊢(Guard f g c,s) → u"

```
 "Γ ⊢(Basic f,s) → u"
 "Γ ⊢(Spec r,s) → u"
 "Γ ⊢(Seq c1 c2,s) → u"
 "Γ ⊢(Cond b c1 c2,s) → u"
 "Γ ⊢(While b c,s) → u"
 "Γ ⊢(Call p,s) → u"
 "Γ ⊢(DynCom c,s) → u"
 "Γ ⊢(Throw,s) → u"
 "Γ ⊢(Catch c1 c2,s) → u"
 "Γ ⊢(Parallel cs,s) → u"
 "Γ ⊢(Await b e,s) → u"
```

inductive_cases step_Normal_elim_cases [cases set]:

```
 "Γ ⊢(Skip,Normal s) → u"
 "Γ ⊢(Guard f g c,Normal s) → u"
 "Γ ⊢(Basic f,Normal s) → u"
 "Γ ⊢(Spec r,Normal s) → u"
 "Γ ⊢(Seq c1 c2,Normal s) → u"
 "Γ ⊢(Cond b c1 c2,Normal s) → u"
 "Γ ⊢(While b c,Normal s) → u"
 "Γ ⊢(Call p,Normal s) → u"
 "Γ ⊢(DynCom c,Normal s) → u"
 "Γ ⊢(Throw,Normal s) → u"
 "Γ ⊢(Catch c1 c2,Normal s) → u"
 "Γ ⊢(Parallel cs,Normal s) → u"
 "Γ ⊢(Await b e,Normal s) → u"
```

abbreviation

```
 "step_trancl" :: "[('s,'p,'f) body, ('s,'p,'f) config,('s,'p,'f) config]
⇒ bool"
                                (‹_⊢ (_ →⁺/ _)› [81,81,81] 100)
```

where

```
  "Γ ⊢cf0 →⁺ cf1 ≡ (CONST step Γ)⁺⁺ cf0 cf1"
```

abbreviation

```
 "step_n_trancl" :: "[('s,'p,'f) body, ('s,'p,'f) config,nat,('s,'p,'f)
config] ⇒ bool"
                                (‹_⊢ (_ →ⁿ_/ _)› [81,81,81,81] 100)
```

where

```
  "Γ ⊢cf0 →ⁿn cf1 ≡ (CONST step Γ ^^ n) cf0 cf1"
```

lemma no_step_final:

assumes step: "Γ ⊢(c,s) → (c',s')"

shows "final (c,s) ⟹ P"

using step

by induct (auto simp add: final_def)

```
lemma no_step_final':
  assumes step: "Γ ⊢cfg → cfg'"
  shows "final cfg ⟹ P"
using step
  by (cases cfg, cases cfg') (auto intro: no_step_final)

lemma no_steps_final:
"Γ ⊢ v →* w ⟹ final v ⟹ w = v"
  apply (cases v)
  apply simp
  apply  (erule converse_rtranclpE)
   apply simp
   apply (erule (1) no_step_final')
 done

lemma step_Fault:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "⋀f. s=Fault f ⟹ s'=Fault f"
using step

by (induct) auto

lemma step_Stuck:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "⋀f. s=Stuck ⟹ s'=Stuck"
using step
by (induct) auto

lemma SeqSteps:
  assumes steps: "Γ ⊢cfg₁→* cfg₂"
  shows "⋀ c₁ s c₁' s'. ⟦cfg₁ = (c₁,s);cfg₂=(c₁',s')⟧
         ⟹ Γ ⊢(Seq c₁ c₂,s) →* (Seq c₁' c₂, s')"
using steps
proof (induct rule: converse_rtranclp_induct [case_names Refl Trans])
  case Refl
  thus ?case
    by simp
next
  case (Trans cfg₁ cfg'')
  have step: "Γ ⊢ cfg₁ → cfg''" by fact
  have steps: "Γ ⊢ cfg'' →* cfg₂" by fact
  have cfg₁: "cfg₁ = (c₁, s)" and cfg₂: "cfg₂ = (c₁', s')"  by fact+
  obtain c₁'' s'' where cfg'': "cfg''=(c₁'',s'')"
    by (cases cfg'') auto
  from step cfg₁ cfg''
  have "Γ ⊢ (c₁,s) → (c₁'',s'')"
    by simp
  hence "Γ ⊢ (Seq c₁ c₂,s) → (Seq c₁'' c₂,s'')"
    by (rule step.Seq)
```

```
  also from Trans.hyps (3) [OF cfg'' cfg₂]
  have "Γ ⊢ (Seq c₁'' c₂, s'') →* (Seq c₁' c₂, s')" .
  finally show ?case .
qed


lemma CatchSteps:
  assumes steps: "Γ ⊢cfg₁→* cfg₂"
  shows "⋀ c₁ s c₁' s'. ⟦cfg₁ = (c₁,s); cfg₂=(c₁',s')⟧
         ⟹ Γ ⊢(Catch c₁ c₂,s) →* (Catch c₁' c₂, s')"
using steps
proof (induct rule: converse_rtranclp_induct [case_names Refl Trans])
  case Refl
  thus ?case
    by simp
next
  case (Trans cfg₁ cfg'')
  have step: "Γ ⊢ cfg₁ → cfg''" by fact
  have steps: "Γ ⊢ cfg'' →* cfg₂" by fact
  have cfg₁: "cfg₁ = (c₁, s)" and cfg₂: "cfg₂ = (c₁', s')"  by fact+
  obtain c₁'' s'' where cfg'': "cfg''=(c₁'',s'')"
    by (cases cfg'') auto
  from step cfg₁ cfg''
  have s: "Γ ⊢ (c₁,s) → (c₁'',s'')"
    by simp
  hence "Γ ⊢ (Catch c₁ c₂,s) → (Catch c₁'' c₂,s'')"
    by (rule step.Catch)
  also from Trans.hyps (3) [OF cfg'' cfg₂]
  have "Γ ⊢ (Catch c₁'' c₂, s'') →* (Catch c₁' c₂, s')" .
  finally show ?case .
qed


lemma steps_Fault: "Γ ⊢ (c, Fault f) →* (Skip, Fault f)"
proof (induct c)
  case (Seq c₁ c₂)
  have steps_c₁: "Γ ⊢ (c₁, Fault f) →* (Skip, Fault f)" by fact
  have steps_c₂: "Γ ⊢ (c₂, Fault f) →* (Skip, Fault f)" by fact
  from SeqSteps [OF steps_c₁ refl refl]
  have "Γ ⊢ (Seq c₁ c₂, Fault f) →* (Seq Skip c₂, Fault f)".
  also
  have "Γ ⊢ (Seq Skip c₂, Fault f) → (c₂, Fault f)" by (rule SeqSkip)
  also note steps_c₂
  finally show ?case by simp
next
  case (Catch c₁ c₂)
  have steps_c₁: "Γ ⊢ (c₁, Fault f) →* (Skip, Fault f)" by fact
  from CatchSteps [OF steps_c₁ refl refl]
  have "Γ ⊢ (Catch c₁ c₂, Fault f) →* (Catch Skip c₂, Fault f)".
```

```
  also
  have "Γ ⊢ (Catch Skip c₂, Fault f) → (Skip, Fault f)" by (rule CatchSkip)

  finally show ?case by simp
qed (fastforce intro: step.intros)+

lemma steps_Stuck: "Γ ⊢ (c, Stuck) →* (Skip, Stuck)"
proof (induct c)
  case (Seq c₁ c₂)
  have steps_c₁: "Γ ⊢ (c₁, Stuck) →* (Skip, Stuck)" by fact
  have steps_c₂: "Γ ⊢ (c₂, Stuck) →* (Skip, Stuck)" by fact
  from SeqSteps [OF steps_c₁ refl refl]
  have "Γ ⊢ (Seq c₁ c₂, Stuck) →* (Seq Skip c₂, Stuck)".
  also
  have "Γ ⊢ (Seq Skip c₂, Stuck) → (c₂, Stuck)" by (rule SeqSkip)
  also note steps_c₂
  finally show ?case by simp
next
  case (Catch c₁ c₂)
  have steps_c₁: "Γ ⊢ (c₁, Stuck) →* (Skip, Stuck)" by fact
  from CatchSteps [OF steps_c₁ refl refl]
  have "Γ ⊢ (Catch c₁ c₂, Stuck) →* (Catch Skip c₂, Stuck)" .
  also
  have "Γ ⊢ (Catch Skip c₂, Stuck) → (Skip, Stuck)" by (rule CatchSkip)

  finally show ?case by simp
qed (fastforce intro: step.intros)+

lemma step_Fault_prop:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "⋀f. s=Fault f ⟹ s'=Fault f"
using step
by (induct) auto

lemma step_Stuck_prop:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "s=Stuck ⟹ s'=Stuck"
using step
by (induct) auto

lemma steps_Fault_prop:
  assumes step: "Γ ⊢ (c, s) →* (c', s')"
  shows "s=Fault f ⟹ s'=Fault f"
using step
proof (induct rule: converse_rtranclp_induct2 [case_names Refl Trans])
  case Refl thus ?case by simp
next
  case (Trans c s c'' s'')
  thus ?case
```

```
    by (auto intro: step_Fault_prop)
qed

lemma steps_Stuck_prop:
  assumes step: "Γ ⊢ (c, s) →* (c', s')"
  shows "s=Stuck ⟹ s'=Stuck"
using step
proof (induct rule: converse_rtranclp_induct2 [case_names Refl Trans])
  case Refl thus ?case by simp
next
  case (Trans c s c'' s'')
  thus ?case
    by (auto intro: step_Stuck_prop)
qed

end
```

3 Annotations, assertions and associated operations

```
theory OG_Annotations
imports SmallStep
begin

type_synonym 's assn = "'s set"

datatype ('s, dead 'p, dead 'f) ann =
    AnnExpr "'s assn"
  | AnnRec "'s assn" "('s, 'p, 'f) ann"
  | AnnWhile "'s assn" "'s assn" "('s, 'p, 'f) ann"
  | AnnComp "('s, 'p, 'f) ann" "('s, 'p, 'f) ann"
  | AnnBin "'s assn" "('s, 'p, 'f) ann" "('s, 'p, 'f) ann"
  | AnnPar "(('s, 'p, 'f) ann × 's assn × 's assn) list"
  | AnnCall "'s assn" nat

type_synonym ('s, 'p, 'f) ann_triple = "('s, 'p, 'f) ann × 's assn
× 's assn"
```

The list of `ann_triple` is useful if the code calls the same function multiple times and require different annotations for the function body each time.

```
type_synonym ('s,'p,'f) proc_assns = "'p ⇒ (('s, 'p, 'f) ann) list
option"

abbreviation  (input) pres:: "('s, 'p, 'f) ann_triple ⇒ ('s, 'p, 'f)
ann"
where "pres a ≡ fst a"

abbreviation (input) postcond :: "('s, 'p, 'f) ann_triple ⇒ 's assn"
```

```
where "postcond a ≡ fst (snd a)"

abbreviation (input) abrcond :: "('s, 'p, 'f) ann_triple ⇒ 's assn"
where "abrcond a ≡ snd (snd a)"

fun pre :: "('s, 'p, 'f) ann ⇒ 's assn" where
  "pre (AnnExpr r)      = r"
| "pre (AnnRec r e)     = r"
| "pre (AnnWhile r i e) = r"
| "pre (AnnComp e₁ e₂)   =  pre e₁"
| "pre (AnnBin r e₁ e₂)  = r"
| "pre (AnnPar as)      = ⋂ (pre ‘ set (map pres (as)))"
| "pre (AnnCall r n)    = r"

fun pre_par :: "('s, 'p, 'f) ann ⇒ bool" where
  "pre_par (AnnComp e₁ e₂) =  pre_par e₁"
| "pre_par (AnnPar as)  = True"
| "pre_par _    = False"

fun pre_set :: "('s, 'p, 'f) ann ⇒ ('s assn) set" where
  "pre_set (AnnExpr r)      = {r}"
| "pre_set (AnnRec r e)     = {r}"
| "pre_set (AnnWhile r i e) = {r}"
| "pre_set (AnnComp e₁ e₂)   =  pre_set e₁"
| "pre_set (AnnBin r e₁ e₂)  = {r}"
| "pre_set (AnnPar as)       = ⋃ (pre_set ‘ set (map pres (as)))"

| "pre_set (AnnCall r n)    = {r}"

lemma fst_BNFs[simp]:
  "a ∈ Basic_BNFs.fsts (a,b)"
   using fsts.intros by auto

lemma "¬pre_par c  ⟹ pre c ∈ pre_set c"
  by (induct c; simp)

lemma pre_set:
  "pre c = ⋂ (pre_set c)"
  by (induct c; fastforce)

lemma pre_imp_pre_set:
  "s ∈ pre c ⟹ a ∈ pre_set c ⟹ s ∈ a"
  by (simp add: pre_set)

abbreviation precond :: "('s, 'p, 'f) ann_triple ⇒ 's assn"
where "precond a ≡ pre (fst a)"

fun strengthen_pre :: "('s, 'p, 'f) ann ⇒ 's assn ⇒ ('s, 'p, 'f) ann"
where
```

```
  "strengthen_pre (AnnExpr r)      r' = AnnExpr (r ∩ r')"
| "strengthen_pre (AnnRec r e)     r' = AnnRec (r ∩ r') e"
| "strengthen_pre (AnnWhile r i e) r' = AnnWhile (r ∩ r') i e"
| "strengthen_pre (AnnComp e1 e2)   r' = AnnComp (strengthen_pre e1 r')
e2"
| "strengthen_pre (AnnBin r e1 e2) r' = AnnBin (r ∩ r') e1 e2"
| "strengthen_pre (AnnPar as)    r' = (AnnPar as)"
| "strengthen_pre (AnnCall r n)    r' = AnnCall (r ∩ r') n"

fun weaken_pre :: "('s, 'p, 'f) ann ⇒ 's assn ⇒ ('s, 'p, 'f) ann" where
  "weaken_pre (AnnExpr r)      r' = AnnExpr (r ∪ r')"
| "weaken_pre (AnnRec r e)     r' = AnnRec (r ∪ r') e"
| "weaken_pre (AnnWhile r i e) r' = AnnWhile (r ∪ r') i e"
| "weaken_pre (AnnComp e1 e2)   r' = AnnComp (weaken_pre e1 r') e2"
| "weaken_pre (AnnBin r e1 e2) r' = AnnBin (r ∪ r') e1 e2"
| "weaken_pre (AnnPar as)   r' = AnnPar as"
| "weaken_pre (AnnCall r n)    r' = AnnCall (r ∪ r') n"

lemma weaken_pre_empty[simp]:
  "weaken_pre r {} = r"
  by (induct r) auto
```

Annotations for call definition (see Language.thy)

```
definition
 ann_call :: "'s assn ⇒ 's assn ⇒ nat ⇒  's assn ⇒'s assn ⇒  's assn
⇒ 's assn ⇒ ('s,'p,'f) ann"
where
 "ann_call init r n restoreq return restorea A ≡
  AnnRec init (AnnComp (AnnComp (AnnComp (AnnExpr init) (AnnCall r n))
(AnnComp (AnnExpr restorea) (AnnExpr A)))
          (AnnRec restoreq (AnnComp (AnnExpr restoreq) (AnnExpr return))))"


inductive ann_matches :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒
('s, 'p, 'f) ann ⇒ ('s, 'p, 'f) com ⇒ bool" where
  ann_skip: "ann_matches Γ Θ (AnnExpr a) Skip"
| ann_basic: "ann_matches Γ Θ (AnnExpr a) (Basic f)"
| ann_spec: "ann_matches Γ Θ (AnnExpr a) (Spec r)"
| ann_throw: "ann_matches Γ Θ (AnnExpr a) (Throw)"
| ann_await: "ann_matches Γ Θ a e ⟹
               ann_matches Γ Θ (AnnRec r a) (Await b e)"
| ann_seq: "⟦ ann_matches Γ Θ a1 p1; ann_matches Γ Θ a2 p2 ⟧ ⟹
               ann_matches Γ Θ (AnnComp a1 a2) (Seq p1 p2)"
| ann_cond: "⟦ ann_matches Γ Θ a1 c1; ann_matches Γ Θ a2 c2 ⟧ ⟹
               ann_matches Γ Θ (AnnBin a a1 a2) (Cond b c1 c2)"
| ann_catch: "⟦ ann_matches Γ Θ a1 c1; ann_matches Γ Θ a2 c2 ⟧ ⟹
                ann_matches Γ Θ (AnnComp a1 a2) (Catch c1 c2)"
| ann_while: "ann_matches Γ Θ a' e ⟹
               ann_matches Γ Θ (AnnWhile a i a') (While b e)"
```

```
| ann_guard: "⟦ ann_matches Γ Θ a' e ⟧ ⟹
               ann_matches Γ Θ (AnnRec a a') (Guard f b e)"
| ann_call: "⟦ Θ f = Some as; Γ f = Some b; n < length as;
              ann_matches Γ Θ (as!n) b⟧ ⟹
   ann_matches Γ Θ (AnnCall a n) (Call f)"
| ann_dyncom: "∀s∈r. ann_matches Γ Θ a (c s) ⟹
               ann_matches Γ Θ (AnnRec r a) (DynCom c)"
| ann_parallel: "⟦ length as = length cs;
                  ∀i<length cs. ann_matches Γ Θ (pres (as!i)) (cs!i)
⟧ ⟹
   ann_matches Γ Θ (AnnPar as) (Parallel cs)"

primrec ann_guards:: "'s assn ⇒ ('f × 's bexp ) list ⇒
                  ('s,'p,'f) ann ⇒ ('s,'p,'f) ann"
where
  "ann_guards _ [] c = c" |
  "ann_guards r (g#gs) c = AnnRec r (ann_guards (r ∩ snd g) gs c)"

end
```

4 Owicki-Gries for Partial Correctness

```
theory OG_Hoare
imports OG_Annotations
begin
```

4.1 Validity of Hoare Tuples: $\Gamma\models_{/F}$ P c Q,A

```
definition
  valid :: "[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com,'s assn,'s
assn] => bool"
                (⟨_ ⊨⇩_/ _ _ _, _⟩  [61,60,1000, 20, 1000,1000] 60)
where
  "Γ ⊨⇩/F P c Q,A ≡ ∀s t c'. Γ⊢(c,s) →* (c', t) ⟶ final (c', t ) ⟶
s ∈ Normal ‘ P ⟶ t ∉ Fault ‘ F
                      ⟶  c' = Skip ∧ t ∈ Normal ‘ Q ∨ c' = Throw ∧
t ∈ Normal ‘ A"
```

4.2 Interference Freedom

```
inductive
 atomicsR :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ ('s, 'p, 'f)
ann ⇒ ('s,'p,'f) com ⇒ ('s assn × ('s, 'p, 'f) com) ⇒ bool"
  for Γ::"('s,'p,'f) body"
  and Θ :: " ('s,'p,'f) proc_assns"
where
  AtBasic: "atomicsR Γ Θ (AnnExpr p) (Basic f) (p, Basic f)"
| AtSpec: "atomicsR Γ Θ (AnnExpr p) (Spec r) (p, Spec r)"
| AtAwait: "atomicsR Γ Θ (AnnRec r ae) (Await b e) (r ∩ b, Await b e)"
```

```
| AtWhileExpr: "atomicsR Γ Θ p e a ⟹ atomicsR Γ Θ (AnnWhile r i p)
(While b e) a"
| AtGuardExpr: "atomicsR Γ Θ p e a ⟹ atomicsR Γ Θ (AnnRec r p) (Guard
f b e) a"
| AtDynCom: "x ∈ r ⟹ atomicsR Γ Θ ad (f x) a ⟹ atomicsR Γ Θ (AnnRec
r ad) (DynCom f) a"
| AtCallExpr: "Γ f = Some b ⟹ Θ f = Some as ⟹
                n < length as ⟹
                atomicsR Γ Θ  (as!n) b a ⟹
               atomicsR Γ Θ (AnnCall r n) (Call f) a"
| AtSeqExpr1: "atomicsR Γ Θ  a1 c1 a ⟹
               atomicsR Γ Θ (AnnComp a1 a2) (Seq c1 c2) a"
| AtSeqExpr2: "atomicsR Γ Θ  a2 c2 a ⟹
               atomicsR Γ Θ (AnnComp a1 a2) (Seq c1 c2) a"
| AtCondExpr1: "atomicsR Γ Θ  a1 c1 a ⟹
               atomicsR Γ Θ (AnnBin r a1 a2) (Cond b c1 c2) a"
| AtCondExpr2: "atomicsR Γ Θ  a2 c2 a ⟹
               atomicsR Γ Θ (AnnBin r a1 a2) (Cond b c1 c2) a"
| AtCatchExpr1: "atomicsR Γ Θ  a1 c1 a ⟹
               atomicsR Γ Θ (AnnComp a1 a2) (Catch c1 c2) a"
| AtCatchExpr2: "atomicsR Γ Θ  a2 c2 a ⟹
               atomicsR Γ Θ (AnnComp a1 a2) (Catch c1 c2) a"
| AtParallelExprs: "i < length cs ⟹ atomicsR Γ Θ (fst (as!i)) (cs!i)
a ⟹
    atomicsR Γ Θ (AnnPar as) (Parallel cs) a"

lemma atomicsR_Skip[simp]:
  "atomicsR Γ Θ a Skip r = False"
 by (auto elim: atomicsR.cases)

lemma atomicsR_Throw[simp]:
  "atomicsR Γ Θ a Throw r = False"
 by (auto elim: atomicsR.cases)

inductive
 assertionsR :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 's assn ⇒
's assn ⇒ ('s, 'p, 'f) ann ⇒ ('s,'p,'f) com ⇒ 's assn ⇒ bool"
  for Γ::"('s,'p,'f) body"
  and Θ :: " ('s,'p,'f) proc_assns"
where
  AsSkip: "assertionsR Γ Θ Q A (AnnExpr p) Skip p"
| AsThrow: "assertionsR Γ Θ Q A (AnnExpr p) Throw p"
| AsBasic: "assertionsR Γ Θ Q A (AnnExpr p) (Basic f) p"
| AsBasicSkip: "assertionsR Γ Θ Q A (AnnExpr p) (Basic f) Q"
| AsSpec: "assertionsR Γ Θ Q A (AnnExpr p) (Spec r) p"
| AsSpecSkip: "assertionsR Γ Θ Q A (AnnExpr p) (Spec r) Q"
| AsAwaitSkip: "assertionsR Γ Θ Q A (AnnRec r ae) (Await b e) Q"
| AsAwaitThrow: "assertionsR Γ Θ Q A (AnnRec r ae) (Await b e) A"
| AsAwait: "assertionsR Γ Θ Q A (AnnRec r ae) (Await b e) r"
```

```
| AsWhileExpr: "assertionsR Γ Θ i A p e a ⟹ assertionsR Γ Θ Q A (AnnWhile
r i p) (While b e) a"
| AsWhileAs: "assertionsR Γ Θ Q A (AnnWhile r i p) (While b e) r"
| AsWhileInv: "assertionsR Γ Θ Q A (AnnWhile r i p) (While b e) i"
| AsWhileSkip: "assertionsR Γ Θ Q A (AnnWhile r i p) (While b e) Q"
| AsGuardExpr: "assertionsR Γ Θ Q A p e a ⟹ assertionsR Γ Θ Q A (AnnRec
r p) (Guard f b e) a"
| AsGuardAs: "assertionsR Γ Θ Q A (AnnRec r p) (Guard f b e) r"
| AsDynComExpr: "x ∈ r ⟹ assertionsR Γ Θ Q A ad (f x) a ⟹ assertionsR
Γ Θ Q A (AnnRec r ad) (DynCom f) a"
| AsDynComAs: "assertionsR Γ Θ Q A (AnnRec r p) (DynCom f) r"
| AsCallAs: "assertionsR Γ Θ Q A (AnnCall r n) (Call f) r"
| AsCallExpr: "Γ f = Some b ⟹ Θ f = Some as ⟹
                n < length as ⟹
                assertionsR Γ Θ Q A (as!n) b a ⟹
               assertionsR Γ Θ Q A (AnnCall r n) (Call f) a"
| AsSeqExpr1: "assertionsR Γ Θ (pre a2) A a1 c1 a ⟹
               assertionsR Γ Θ Q A (AnnComp a1 a2) (Seq c1 c2) a"
| AsSeqExpr2: "assertionsR Γ Θ Q A a2 c2 a ⟹
               assertionsR Γ Θ Q A (AnnComp a1 a2) (Seq c1 c2) a"
| AsCondExpr1: "assertionsR Γ Θ Q A a1 c1 a ⟹
               assertionsR Γ Θ Q A (AnnBin r a1 a2) (Cond b c1 c2) a"

| AsCondExpr2: "assertionsR Γ Θ Q A a2 c2 a ⟹
               assertionsR Γ Θ Q A (AnnBin r a1 a2) (Cond b c1 c2) a"

| AsCondAs: "assertionsR Γ Θ Q A (AnnBin r a1 a2) (Cond b c1 c2) r"
| AsCatchExpr1: "assertionsR Γ Θ Q (pre a2) a1 c1 a ⟹
               assertionsR Γ Θ Q A (AnnComp a1 a2) (Catch c1 c2) a"
| AsCatchExpr2: "assertionsR Γ Θ Q A a2 c2 a ⟹
               assertionsR Γ Θ Q A (AnnComp a1 a2) (Catch c1 c2) a"

| AsParallelExprs: "i < length cs ⟹ assertionsR Γ Θ (postcond (as!i))
(abrcond (as!i)) (pres (as!i)) (cs!i) a ⟹
    assertionsR Γ Θ Q A (AnnPar as) (Parallel cs) a"
| AsParallelSkips: "Qs = ⋂ (set (map postcond as)) ⟹
  assertionsR Γ Θ Q A (AnnPar as) (Parallel cs) (Qs)"
```

definition

```
  interfree_aux :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
⇒ (('s,'p,'f) com × ('s, 'p, 'f) ann_triple × ('s,'p,'f) com × ('s,
'p, 'f) ann) ⇒ bool"
```

where

```
  "interfree_aux Γ Θ F ≡ λ(c₁, (P₁, Q₁, A₁), c₂, P₂).
                            (∀p c .atomicsR Γ Θ P₂ c₂ (p,c) ⟶
                             Γ⊨/F (Q₁ ∩ p) c Q₁,Q₁  ∧ Γ⊨/F (A₁ ∩ p)
c A₁,A₁  ∧
                              (∀a. assertionsR Γ Θ Q₁ A₁ P₁ c₁ a ⟶
Γ⊨/F (a ∩ p) c a,a))"
```

definition

```
  interfree :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set ⇒
(('s, 'p, 'f) ann_triple) list ⇒ ('s,'p,'f) com list  ⇒ bool"
```

where

```
  "interfree Γ Θ F Ps Ts ≡ ∀i j. i < length Ts ∧ j < length Ts ∧ i ≠
j ⟶
                          interfree_aux Γ Θ F (Ts!i, Ps!i, Ts!j, fst (Ps!j))"
```

4.3 The Owicki-Gries Logic for COMPLX

inductive

```
  oghoare :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
              ⇒ ('s, 'p, 'f) ann ⇒ ('s,'p,'f) com ⇒ 's assn ⇒ 's assn
⇒ bool"
    (⦇(4_, _/ ⊢⇩_/ (_/ (_)/ _, _))⦈ [60,60,60,1000,1000,1000,1000]60)
```

and

```
  oghoare_seq :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
              ⇒ 's assn ⇒ ('s, 'p, 'f) ann ⇒ ('s,'p,'f) com ⇒ 's assn
⇒ 's assn ⇒ bool"
    (⦇(4_, _/ ⊨⇩_/ (_/ _/ (_)/ _, _))⦈ [60,60,60,1000,1000,1000,1000]60)
```

where

```
 Skip: " Γ, Θ ⊢/F (AnnExpr Q) Skip Q,A"

| Throw: "Γ, Θ ⊢/F (AnnExpr A) Throw Q,A"

| Basic: "Γ, Θ ⊢/F (AnnExpr {s. f s ∈ Q}) (Basic f) Q,A"

| Spec: "Γ, Θ ⊢/F (AnnExpr {s. (∀t. (s,t) ∈ rel ⟶ t ∈ Q) ∧ (∃t. (s,t)
∈ rel)}) (Spec rel) Q,A"

| Seq: "⟦Γ, Θ ⊢/F P₁ c₁ (pre P₂),A;
        Γ, Θ ⊢/F P₂ c₂ Q,A ⟧
        ⟹ Γ, Θ ⊢/F (AnnComp P₁ P₂) (Seq c₁ c₂) Q,A"

| Catch: "⟦Γ, Θ ⊢/F P₁ c₁ Q,(pre P₂);
          Γ, Θ ⊢/F P₂ c₂ Q,A ⟧
     ⟹  Γ, Θ ⊢/F (AnnComp P₁ P₂) (Catch c₁ c₂) Q,A"

| Cond: "⟦ Γ, Θ ⊢/F P₁ c₁ Q,A;
          Γ, Θ ⊢/F P₂ c₂ Q,A;
         r ∩ b ⊆ pre P₁;
         r ∩ -b ⊆ pre P₂ ⟧
        ⟹
        Γ, Θ ⊢/F (AnnBin r P₁ P₂) (Cond b c₁ c₂) Q,A"

| While: "⟦ Γ, Θ ⊢/F P c i,A;
           i ∩ b ⊆ pre P;
```

```
          i ∩ -b ⊆ Q;
          r ⊆ i  ⟧
         ⟹
         Γ, Θ ⊢_/F (AnnWhile r i P) (While b c) Q,A"

| Guard: "⟦ Γ, Θ ⊢_/F P c Q,A;
          r ∩ g ⊆ pre P;
          r ∩ -g ≠ {} ⟶ f ∈ F⟧ ⟹
         Γ, Θ ⊢_/F (AnnRec r P) (Guard f g c) Q,A"

| Call: "⟦  Θ p = Some as;
         (as ! n) = P;
         r ⊆ pre P;
         Γ p = Some b;
         n < length as;
         Γ,Θ ⊢_/F P b Q,A
         ⟧
        ⟹
        Γ, Θ ⊢_/F (AnnCall r n) (Call p) Q,A"

| DynCom:
     "r ⊆ pre a ⟹ ∀s∈r. Γ, Θ ⊢_/F a (c s) Q,A
     ⟹
     Γ, Θ ⊢_/F (AnnRec r a) (DynCom c) Q,A"

| Await: "⟦Γ, Θ ⊩_/F(r ∩ b) P c Q,A; atom_com c ⟧ ⟹
  Γ, Θ ⊢_/F (AnnRec r P) (Await b c) Q,A"

| Parallel: "⟦ length as = length cs;
              ∀i < length cs. Γ,Θ⊢_/F(pres (as!i)) (cs!i) (postcond (as!i)),(abrcond
(as!i));
              interfree Γ Θ F as cs;
              ⋂ (set (map postcond as)) ⊆ Q;
              ⋃ (set (map abrcond as)) ⊆ A
             ⟧
       ⟹ Γ,Θ⊢_/F (AnnPar as) (Parallel cs) Q,A"

| Conseq: "∃P' Q' A'. Γ,Θ⊢_/F (weaken_pre P P') c Q',A' ∧ Q' ⊆ Q ∧ A'
⊆ A
          ⟹ Γ,Θ⊢_/F P c Q,A"

| SeqSkip:   "Γ, Θ ⊩_/F Q (AnnExpr x) Skip Q,A"

| SeqThrow:   "Γ, Θ ⊩_/F A (AnnExpr x) Throw Q,A"

| SeqBasic:   "Γ, Θ ⊩_/F {s. f s ∈ Q} (AnnExpr x) (Basic f) Q,A"

| SeqSpec:   "Γ, Θ ⊩_/F {s. (∀t. (s,t) ∈ r ⟶ t ∈ Q) ∧ (∃t. (s,t) ∈
r)} (AnnExpr x) (Spec r) Q,A"
```

```
| SeqSeq:    "⟦ Γ, Θ ⊨/F R P2 c2 Q,A ; Γ, Θ ⊨/F P P1 c1 R,A⟧
        ⟹ Γ, Θ ⊨/F P (AnnComp P1 P2) (Seq c1 c2) Q,A"

| SeqCatch:  "⟦Γ, Θ ⊨/F R P2 c2 Q,A ; Γ, Θ ⊨/F P P1 c1 Q,R ⟧ ⟹
   Γ, Θ ⊨/F P (AnnComp P1 P2) (Catch c1 c2) Q,A"

| SeqCond: "⟦ Γ, Θ ⊨/F (P ∩ b) P1 c1 Q,A;
         Γ, Θ ⊨/F (P ∩ -b) P2 c2 Q,A ⟧
        ⟹
        Γ, Θ ⊨/F P (AnnBin r P1 P2) (Cond b c1 c2) Q,A"

| SeqWhile: "⟦ Γ, Θ ⊨/F (P∩b) a c P,A ⟧
         ⟹
         Γ, Θ ⊨/F P (AnnWhile r i a) (While b c) (P ∩ -b),A"

| SeqGuard: "⟦ Γ, Θ ⊨/F (P∩g) a c Q,A;
          P ∩ -g ≠ {} ⟹ f ∈ F⟧ ⟹
         Γ, Θ ⊨/F P (AnnRec r a) (Guard f g c) Q,A"

| SeqCall: "⟦  Θ p = Some as;
         (as ! n) = P'';
         Γ p = Some b;
         n < length as;
         Γ,Θ ⊨/F P P'' b Q,A
         ⟧
        ⟹
        Γ, Θ ⊨/F P (AnnCall r n) (Call p) Q,A"

| SeqDynCom:
     "r ⊆ pre a ⟹
     ∀s∈r. Γ, Θ ⊨/FP a (c s) Q,A ⟹
     P⊆r
     ⟹
     Γ, Θ ⊨/F P (AnnRec r a) (DynCom c) Q,A"

| SeqConseq: "⟦ P ⊆ P'; Γ,Θ⊨/F P' a c Q',A'; Q' ⊆ Q; A' ⊆ A ⟧
          ⟹ Γ,Θ⊨/F P a c Q,A"

| SeqParallel: "P ⊆ pre (AnnPar as) ⟹ Γ,Θ⊨/F (AnnPar as) (Parallel
cs) Q,A
  ⟹ Γ,Θ⊨/F P (AnnPar as) (Parallel cs) Q,A"

lemmas oghoare_intros = "oghoare_oghoare_seq.intros"

lemmas oghoare_inducts = oghoare_oghoare_seq.inducts

lemmas oghoare_induct = oghoare_oghoare_seq.inducts(1)
```

```
lemmas oghoare_seq_induct = oghoare_oghoare_seq.inducts(2)

end
```

Helper lemmas for sequential reasoning about Seq and Catch

```
theory SeqCatch_decomp
imports SmallStep
begin

lemma redex_size[rule_format] :
"∀r. redex c = r ⟶ size r ≤ size c"
  by(induct_tac c, simp_all)

lemma Normal_pre[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀u. s' = Normal u ⟶ (∃v. s = Normal v)"
  by(erule step_induct, simp_all)


lemma Normal_pre_star[rule_format, OF _ refl] :
"Γ ⊢ cfg₁ →* cfg₂ ⟹ ∀p' t. cfg₂ = (p', Normal t) ⟶
 (∃p s. cfg₁ = (p, Normal s))"
  apply(erule converse_rtranclp_induct)
   apply simp
  apply clarsimp
  apply(drule Normal_pre)
  by force



lemma Seq_decomp_Skip[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₂. p = Seq Skip p₂ ⟶ s' = s ∧ p' = p₂"
  apply(erule step_induct, simp_all)
    apply clarsimp
    apply(erule step.cases, simp_all)
   apply clarsimp+
 done

lemma Seq_decomp_Throw[rule_format, OF _ refl, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₂ z. s = Normal z ⟶ p = Seq Throw p₂ ⟶ s' = s ∧ p' = Throw"
  apply(erule step_induct, simp_all)
  apply clarsimp
  apply(erule step.cases, simp_all)
 done

lemma Throw_star[rule_format, OF _ refl] :
"Γ ⊢ cfg₁ →* cfg₂ ⟹ ∀s. cfg₁ = (Throw, Normal s) ⟶
  cfg₂ = cfg₁"
```

```
  apply(erule converse_rtranclp_induct)
   apply simp
  apply clarsimp
  apply(erule step.cases, simp_all)
 done

lemma Seq_decomp[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₁ p₂. p = Seq p₁ p₂ ⟶ p₁ ≠ Skip ⟶ p₁ ≠ Throw ⟶
  (∃p₁'. Γ ⊢ (p₁, s) → (p₁', s') ∧ p' = Seq p₁' p₂)"
  apply(erule step_induct, simp_all)
   apply clarsimp
   apply(drule redex_size)
   apply simp
  apply clarsimp
  apply(drule redex_size)
  apply simp
 done

lemma Seq_decomp_relpow:
"Γ ⊢ (Seq p₁ p₂, Normal s) →^n n (p', Normal s') ⟹
 final (p', Normal s') ⟹
 (∃n1<n. Γ ⊢ (p₁, Normal s) →^n n1 (Throw, Normal s')) ∧ p'=Throw ∨
 (∃t n1 n2. Γ ⊢ (p₁, Normal s) →^n n1 (Skip, Normal t) ∧ n1 < n ∧ n2 <
n ∧ Γ ⊢ (p₂, Normal t) →^n n2 (p', Normal s'))"
  apply (induct n arbitrary: p₁ p₂ s p' s')
   apply (clarsimp simp: final_def)
  apply (simp only: relpowp.simps)
  apply (subst (asm) relpowp_commute[symmetric])
  apply clarsimp
  apply (rename_tac n p₁ p₂ s p' s' a b)
  apply(case_tac "p₁ = Skip")
   apply clarsimp
   apply(drule Seq_decomp_Skip)
   apply clarsimp
   apply(drule_tac x=s in spec)
   apply(drule_tac x=0 in spec)
   apply simp
  apply (rename_tac n p₁ p₂ s p' s' a b)
  apply(case_tac "p₁ = Throw")
   apply clarsimp
   apply(drule Seq_decomp_Throw)
   apply clarsimp
   apply(frule relpowp_imp_rtranclp)
   apply(drule Throw_star)
   apply clarsimp
   apply(rule_tac x=n in exI, simp)
  apply(drule Seq_decomp)
    apply assumption+
```

```
  apply (rename_tac n p1 p2 s p' s' a b)
  apply clarsimp
  apply(frule relpowp_imp_rtranclp)
  apply(drule Normal_pre_star)
  apply clarsimp
  apply(drule meta_spec, drule meta_spec, drule meta_spec, drule meta_spec,
drule meta_spec, drule meta_mp, assumption)
  apply(drule meta_mp, assumption)
  apply(erule disjE, clarsimp)
   apply (rename_tac n1)
   apply(rule_tac x="Suc n1" in exI, simp)
   apply (subst relpowp_commute[symmetric])
   apply(rule_tac relcomppI, assumption+)
  apply clarsimp
  apply (rename_tac t n1 n2)
  apply(drule_tac x=t in spec)
  apply(drule_tac x="Suc n1" in spec)
  apply simp
  apply(drule mp)
  apply (subst relpowp_commute[symmetric])
  apply(rule_tac relcomppI, assumption+)
  apply simp
 done

lemma Seq_decomp_star:
"Γ ⊢ (Seq p1 p2, Normal s) →* (p', Normal s') ⟹ final (p', Normal s')
⟹
 Γ ⊢ (p1, Normal s) →* (Throw, Normal s') ∧ p'=Throw ∨
 (∃t. Γ ⊢ (p1, Normal s) →* (Skip, Normal t) ∧ Γ ⊢ (p2, Normal t) →*
(p', Normal s'))"
  apply (drule rtranclp_imp_relpowp)
  apply clarsimp
  apply (drule (1) Seq_decomp_relpow)
  apply (erule disjE)
   apply clarsimp
   apply (drule (1) relpowp_imp_rtranclp)
  apply clarsimp
  apply (drule  relpowp_imp_rtranclp)+
  apply clarsimp
 done

lemma Seq_decomp_relpowp_Fault:
"Γ ⊢ (Seq p1 p2, Normal s) →^n n (Skip, Fault f) ⟹
 (∃n1. Γ ⊢ (p1, Normal s) →^n n1 (Skip, Fault f)) ∨
 (∃t n1 n2. Γ ⊢ (p1, Normal s) →^n n1 (Skip, Normal t) ∧ n1 < n ∧ n2 <
n  ∧ Γ ⊢ (p2, Normal t) →^n n2 (Skip, Fault f))"
  apply (induct n arbitrary: s p1)
   apply (clarsimp simp: final_def)
  apply (simp only: relpowp.simps)
```

```
  apply (subst (asm) relpowp_commute[symmetric])
  apply clarsimp
  apply (rename_tac n s p1 a b)
  apply (case_tac "p1 = Skip")
   apply simp
   apply (drule Seq_decomp_Skip)
   apply clarify
   apply (drule_tac x=s in spec)
   apply (drule spec[where x=0])
   apply simp
  apply (case_tac "p1 = Throw")
   apply simp
   apply (drule Seq_decomp_Throw)
   apply fastforce
  apply (frule Seq_decomp )
    apply simp+
  apply clarsimp
  apply (rename_tac s p1 t p1')
  apply (case_tac "∃v. Normal v = t")
   apply clarsimp
   apply (rename_tac v)
   apply (drule_tac x=v in meta_spec)
   apply (drule_tac x=p1' in meta_spec)
   apply clarsimp
   apply (erule disjE)
    apply clarsimp
    apply (rename_tac n1)
    apply (rule_tac x="n1+1" in exI)
    apply (drule (1) relpowp_Suc_I2, fastforce)
   apply clarsimp
   apply (rename_tac t n1 n2)
   apply (drule_tac x=t in spec)
   apply (drule_tac x="n1+1" in spec)
   apply simp
   apply (subst (asm) relpowp_commute[symmetric])
   apply (drule mp)
   apply (erule relcomppI, assumption)
   apply clarsimp
  apply (case_tac "t = Stuck")
   apply clarsimp
   apply (drule relpowp_imp_rtranclp)
   apply (fastforce dest: steps_Stuck_prop)
  apply (case_tac t, simp_all)
  apply (rename_tac f')
  apply (frule steps_Fault_prop[where s'="Fault f", OF relpowp_imp_rtranclp,
THEN sym], rule refl)
  apply clarify
  apply (cut_tac c=p1' in steps_Fault[where Γ=Γ and f=f])
  apply (drule rtranclp_imp_relpowp)
```

```
  apply clarsimp
  apply (rename_tac n')
  apply (rule_tac x="n'+1" in exI)
  apply simp
  apply (subst relpowp_commute[symmetric])
  apply (erule relcomppI, assumption)
 done

lemma Seq_decomp_star_Fault[rule_format, OF _ refl, OF _ refl, OF _ refl]
:
"Γ ⊢ cfg₁ →* cfg₂ ⟹ ∀p s p' f. cfg₁ = (p, Normal s) ⟶ cfg₂ = (Skip,
Fault f) ⟶
 (∀p₁ p₂. p = Seq p₁ p₂ ⟶
 (Γ ⊢ (p₁, Normal s) →* (Skip, Fault f))
 ∨ (∃s'. (Γ ⊢ (p₁,  Normal s) →* (Skip, Normal s')) ∧ Γ ⊢ (p₂, Normal
s') →* (Skip, Fault f)))"
  apply(erule converse_rtranclp_induct)
   apply clarsimp
  apply clarsimp
  apply (rename_tac c s' s f p₁ p₂)
  apply (case_tac "p₁ = Skip")
   apply simp
   apply (drule Seq_decomp_Skip)
   apply simp
  apply (case_tac "p₁ = Throw")
   apply simp
   apply (drule Seq_decomp_Throw)
   apply simp
  apply (drule Seq_decomp)
    apply simp+
  apply clarsimp
  apply (case_tac s')
    apply simp
    apply (erule disjE)
     apply (erule (1) converse_rtranclp_into_rtranclp)
    apply clarsimp
    apply (rename_tac s')
    apply (erule_tac x="s'" in allE)
    apply (drule (1) converse_rtranclp_into_rtranclp, fastforce)
   apply simp
   apply (frule steps_Fault_prop[THEN sym], rule refl)
   apply simp
   apply (cut_tac c=p₁' and f=f in steps_Fault[where Γ=Γ])
   apply (drule (2)  converse_rtranclp_into_rtranclp)
  apply clarsimp
  apply (frule steps_Stuck_prop[THEN sym], rule refl)
  apply simp
 done
```

```
lemma Seq_decomp_relpowp_Stuck:
"Γ ⊢ (Seq p1 p2, Normal s) →^n n (Skip, Stuck) ⟹
 (∃n1. Γ ⊢ (p1, Normal s) →^n n1 (Skip, Stuck)) ∨
 (∃t n1 n2. Γ ⊢ (p1, Normal s) →^n n1 (Skip, Normal t) ∧ n1 < n ∧ n2 <
n  ∧ Γ ⊢ (p2, Normal t) →^n n2 (Skip, Stuck))"
  apply (induct n arbitrary: s p1)
   apply (clarsimp simp: final_def)
  apply (simp only: relpowp.simps)
  apply (subst (asm) relpowp_commute[symmetric])
  apply clarsimp
  apply (rename_tac n s p1 a b)
  apply (case_tac "p1 = Skip")
   apply simp
   apply (drule Seq_decomp_Skip)
   apply clarify
   apply (drule_tac x=s in spec)
   apply (drule spec[where x=0])
   apply simp
  apply (case_tac "p1 = Throw")
   apply simp
   apply (drule Seq_decomp_Throw)
   apply fastforce
  apply (frule Seq_decomp )
    apply simp+
  apply clarsimp
  apply (rename_tac s p1 t p1')
  apply (case_tac "∃v. Normal v = t")
   apply clarsimp
   apply (rename_tac v)
   apply (drule_tac x=v in meta_spec)
   apply (drule_tac x=p1' in meta_spec)
   apply clarsimp
   apply (erule disjE)
    apply clarsimp
    apply (rename_tac n1)
    apply (rule_tac x="n1+1" in exI)
    apply (drule (1) relpowp_Suc_I2, fastforce)
   apply clarsimp
   apply (rename_tac t n1 n2)
   apply (drule_tac x=t in spec)
   apply (drule_tac x="n1+1" in spec)
   apply simp
   apply (subst (asm) relpowp_commute[symmetric])
   apply (drule mp)
   apply (erule relcomppI, assumption)
   apply clarsimp
  apply (case_tac "∃v. t = Fault v")
   apply clarsimp
```

```
   apply (drule relpowp_imp_rtranclp)
   apply (fastforce dest: steps_Fault_prop)
  apply (case_tac t, simp_all)
  apply (rename_tac p1')
  apply clarify
  apply (cut_tac c=p1' in steps_Stuck[where Γ=Γ])
  apply (drule rtranclp_imp_relpowp)
  apply clarsimp
  apply (rename_tac n')
  apply (rule_tac x="n'+1" in exI)
  apply simp
  apply (subst relpowp_commute[symmetric])
  apply (erule relcomppI, assumption)
 done

lemma Seq_decomp_star_Stuck[rule_format, OF _ refl, OF _ refl] :
"Γ ⊢ cfg₁ →* (Skip, Stuck) ⟹ ∀p s p'. cfg₁ = (p, Normal s) ⟶
 (∀p₁ p₂. p = Seq p₁ p₂ ⟶
 (Γ ⊢ (p₁, Normal s) →* (Skip, Stuck))
 ∨ (∃s'. (Γ ⊢ (p₁,  Normal s) →* (Skip, Normal s')) ∧ Γ ⊢ (p₂, Normal
s') →* (Skip, Stuck)))"
  apply(erule converse_rtranclp_induct)
   apply clarsimp
  apply clarsimp
  apply (rename_tac c s' s p₁ p₂)
  apply (case_tac "p₁ = Skip")
   apply simp
   apply (drule Seq_decomp_Skip)
   apply simp
  apply (case_tac "p₁ = Throw")
   apply simp
   apply (drule Seq_decomp_Throw)
   apply simp
  apply (drule Seq_decomp)
    apply simp+
  apply clarsimp
  apply (rename_tac s' s p₁ p₂ p₁')
  apply (case_tac s')
    apply simp
    apply (erule disjE)
     apply (erule (1) converse_rtranclp_into_rtranclp)
    apply clarsimp
    apply (rename_tac s')
    apply (erule_tac x="s'" in allE)
    apply (drule (1) converse_rtranclp_into_rtranclp, fastforce)
   apply simp
   apply (drule steps_Fault_prop, rule refl, fastforce)
   apply simp
   apply (cut_tac c=p₁' in steps_Stuck[where Γ=Γ])
```

```
  apply (frule steps_Stuck_prop[THEN sym], rule refl)
  apply simp
 done


lemma Catch_decomp_star[rule_format, OF _ refl, OF _ refl, OF _ _ refl]:
" Γ ⊢ cfg₁ →* cfg₂ ⟹
    ∀p s p' s'.
       cfg₁ = (p, Normal s) ⟶
       cfg₂ = (p', Normal s') ⟶
       final (p', Normal s') ⟶
       (∀p₁ p₂.
           p = Catch p₁ p₂ ⟶
           (∃t. Γ ⊢ (p₁, Normal s) →* (Throw, Normal t) ∧ Γ ⊢ (p₂, Normal
t) →* (p', Normal s')) ∨
           (Γ ⊢ (p₁, Normal s) →* (Skip, Normal s') ∧ p' = Skip))"
  apply (erule converse_rtranclp_induct)
   apply (clarsimp simp: final_def)
  apply clarsimp
  apply (rename_tac a b s p' s' p₁ p₂)
  apply (case_tac b)
    apply clarsimp
    apply (rename_tac x)
    apply (erule step_Normal_elim_cases)
      apply clarsimp
      apply (erule disjE)
       apply clarsimp
       apply (rename_tac t)
       apply (rule_tac x=t in exI)
       apply fastforce
      apply (erule impE)
       apply fastforce
      apply fastforce
     apply clarsimp
     apply (clarsimp simp: final_def)
     apply (erule disjE)
      apply fastforce
     apply clarsimp
     apply (fastforce dest: no_steps_final simp: final_def)
    apply (clarsimp simp: final_def)
    apply (erule disjE)
     apply clarsimp
     apply (rename_tac s s' p₂)
     apply (rule_tac x=s in exI)
     apply fastforce
    apply fastforce
   apply (fastforce dest: steps_Fault_prop)
  apply (fastforce dest: steps_Stuck_prop)
 done
```

```
lemma Catch_decomp_Skip[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₂. p = Catch Skip p₂ ⟶ s' = s ∧ p' = Skip"
  apply(erule step_induct, simp_all)
  apply clarsimp
  apply(erule step.cases, simp_all)
 done

lemma Catch_decomp[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₁ p₂. p = Catch p₁ p₂ ⟶ p₁ ≠ Skip ⟶ p₁ ≠ Throw ⟶
  (∃p₁'. Γ ⊢ (p₁, s) → (p₁', s') ∧ p' = Catch p₁' p₂)"
  apply(erule step_induct, simp_all)
   apply clarsimp
  apply(drule redex_size)
  apply simp
  apply clarsimp
  apply(drule redex_size)
  apply simp
 done

lemma Catch_decomp_star_Fault[rule_format, OF _ refl, OF _ refl, OF _
refl] :
"Γ ⊢ cfg₁ →* cfg₂ ⟹ ∀p s f. cfg₁ = (p, Normal s) ⟶ cfg₂ = (Skip,
Fault f) ⟶
 (∀p₁ p₂. p = Catch p₁ p₂ ⟶
 (Γ ⊢ (p₁, Normal s) →* (Skip, Fault f))
 ∨ (∃s'. (Γ ⊢ (p₁,  Normal s) →* (Throw, Normal s')) ∧ Γ ⊢ (p₂, Normal
s') →* (Skip, Fault f)))"
  apply(erule converse_rtranclp_induct)
   apply clarsimp
  apply clarsimp
  apply (rename_tac c s' s f p₁ p₂)
  apply (case_tac "p₁ = Skip")
   apply (fastforce dest: Catch_decomp_Skip)
  apply (case_tac "p₁ = Throw")
   apply simp
   apply (case_tac s')
     apply clarsimp
     apply (erule step_Normal_elim_cases)
       apply clarsimp
       apply (erule disjE)
        apply fastforce
       apply (fastforce elim: no_step_final simp:final_def)
      apply fastforce
     apply fastforce
    apply clarsimp
    apply (drule_tac x=s in spec)
```

```
    apply clarsimp
    apply (erule step_elim_cases, simp_all)
    apply clarsimp
    apply (cut_tac c=c1' and f=f in steps_Fault[where Γ=Γ])
    apply (drule steps_Fault_prop, rule refl)
    apply fastforce
   apply (fastforce dest: steps_Stuck_prop)
  apply (drule (2) Catch_decomp)
  apply clarsimp
  apply (rename_tac p1')
  apply (case_tac s')
    apply simp
    apply (erule disjE)
     apply (erule (1) converse_rtranclp_into_rtranclp)
    apply clarsimp
    apply (rename_tac s')
    apply (erule_tac x="s'" in allE)
    apply (drule (1) converse_rtranclp_into_rtranclp, fastforce)
   apply simp
   apply (frule steps_Fault_prop[THEN sym], rule refl)
   apply simp
   apply (cut_tac c=p1' and f=f in steps_Fault[where Γ=Γ])
   apply (drule (2)  converse_rtranclp_into_rtranclp)
  apply (fastforce dest: steps_Stuck_prop)
 done

lemma Catch_decomp_star_Stuck[rule_format, OF _ refl, OF _ refl, OF _
refl] :
"Γ ⊢ cfg1 →* cfg2 ⟹ ∀p s. cfg1 = (p, Normal s) ⟶ cfg2 = (Skip, Stuck)
⟶
 (∀p1 p2. p = Catch p1 p2 ⟶
 (Γ ⊢ (p1, Normal s) →* (Skip, Stuck))
 ∨ (∃s'. (Γ ⊢ (p1,  Normal s) →* (Throw, Normal s')) ∧ Γ ⊢ (p2, Normal
s') →* (Skip, Stuck)))"
  apply(erule converse_rtranclp_induct)
   apply clarsimp
  apply clarsimp
  apply (rename_tac c s' s p1 p2)
  apply (case_tac "p1 = Skip")
   apply (fastforce dest: Catch_decomp_Skip)
  apply (case_tac "p1 = Throw")
   apply simp
   apply (case_tac s')
     apply clarsimp
     apply (erule step_Normal_elim_cases)
       apply clarsimp
       apply (erule disjE)
        apply fastforce
       apply (fastforce elim: no_step_final simp:final_def)
```

```
     apply fastforce
    apply fastforce
   apply clarsimp
   apply (drule_tac x=s in spec)
   apply clarsimp
   apply (erule step_elim_cases, simp_all)
   apply clarsimp
   apply (rename_tac c1')
   apply (cut_tac c=c1' in steps_Fault[where Γ=Γ])
   apply (drule steps_Fault_prop, rule refl)
   apply fastforce

  apply (drule_tac x=s in spec)
  apply clarsimp
  apply (erule step_elim_cases, simp_all)
  apply clarsimp
  apply (cut_tac c=c1' in steps_Stuck[where Γ=Γ])
  apply (fastforce)

 apply (drule (2) Catch_decomp)
 apply clarsimp
 apply (rename_tac p1')
 apply (case_tac s')
   apply simp
   apply (erule disjE)
    apply (erule (1) converse_rtranclp_into_rtranclp)
   apply clarsimp
   apply (rename_tac s')
   apply (erule_tac x="s'" in allE)
   apply (drule (1) converse_rtranclp_into_rtranclp, fastforce)
  apply simp
  apply (frule steps_Fault_prop[THEN sym], rule refl)
  apply fastforce
 apply (cut_tac c=p1' in steps_Stuck[where Γ=Γ])
 apply fastforce
done

end
```

5 Soundness proof of Owicki-Gries w.r.t. COMPLX small-step semantics

```
theory OG_Soundness
imports
  OG_Hoare
  SeqCatch_decomp
begin
```

```
lemma pre_weaken_pre:
 " x ∈ pre P  ⟹ x ∈ pre (weaken_pre P P')"
 by (induct P, clarsimp+)

lemma oghoare_Skip[rule_format, OF _ refl]:
"Γ, Θ ⊢/F P c Q,A ⟹ c = Skip ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ Q)"
  apply (induct rule: oghoare_induct, simp_all)
  apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'')
  apply(case_tac P, simp_all)
  by force

lemma oghoare_Throw[rule_format, OF _ refl]:
"Γ, Θ ⊢/F P c Q,A ⟹ c = Throw ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ A)"
  apply (induct rule: oghoare_induct, simp_all)
  apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'')
  apply(case_tac P, simp_all)
  by force

lemma oghoare_Basic[rule_format, OF _ refl]:
"Γ, Θ ⊢/F P c Q,A ⟹ c = Basic f ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ {x. f x ∈ Q})"
  apply (induct rule: oghoare_induct, simp_all)
   apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'')
  apply(case_tac P, simp_all)
  by force

lemma oghoare_Spec[rule_format, OF _ refl]:
"Γ, Θ ⊢/F P c Q,A ⟹ c = Spec r ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ {s. (∀t. (s, t) ∈ r ⟶ t ∈ Q) ∧ (∃t.
(s, t) ∈ r)})"
  apply (induct rule: oghoare_induct, simp_all)
  apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'')
  apply(case_tac P, simp_all)
  by force

lemma oghoare_DynCom[rule_format, OF _ refl]:
"Γ, Θ ⊢/F P c Q,A ⟹ c = (DynCom c') ⟶
 (∃r ad. P = (AnnRec r ad) ∧ r ⊆ pre ad ∧ (∀s∈r. Γ, Θ ⊢/F ad (c' s)
Q,A))"
  apply (induct rule: oghoare_induct, simp_all)
   apply clarsimp
  apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'' x)
```

```
  apply(case_tac P, simp_all)
  apply clarsimp
  apply (rename_tac P s)
  apply (drule_tac x=s in bspec, simp)
  apply (rule Conseq)
  apply (rule_tac x="{}" in exI)
  apply (fastforce)
 done

lemma oghoare_Guard[rule_format, OF _ refl]:
"Γ,Θ⊢/F P c Q,A ⟹ c = Guard f g d ⟶
 (∃P' r . P = AnnRec r P' ∧
   Γ,Θ⊢/F P' d Q,A ∧
   r ∩ g ⊆ pre P' ∧
   (r ∩ -g ≠ {} ⟶ f ∈ F))"
  apply (induct rule: oghoare_induct, simp_all)
   apply force
  apply clarsimp
  apply (rename_tac Γ Θ F P Q A P' Q' A' P'' r)
  apply (case_tac P, simp_all)
  apply clarsimp
  apply (rename_tac r)
  apply(rule conjI)
   apply (rule Conseq)
   apply (rule_tac x="{}" in exI)
   apply (rule_tac x="Q'" in exI)
   apply (rule_tac x="A'" in exI)
   apply (clarsimp)
  apply (case_tac "(r ∪ P') ∩ g ≠ {}")
   apply fast
  apply clarsimp
  apply(drule equalityD1, force)
 done

lemma oghoare_Await[rule_format, OF _ refl]:
"Γ, Θ⊢/F P x Q,A ⟹ ∀b c. x = Await b c ⟶
 (∃r P' Q' A'. P = AnnRec r P' ∧ Γ, Θ⊩/F(r ∩ b) P' c Q',A' ∧ atom_com
c
                ∧ Q' ⊆ Q ∧ A' ⊆ A)"
  supply [[simproc del: defined_all]]
  apply (induct rule: oghoare_induct, simp_all)
    apply (rename_tac Γ Θ F r P Q A)
   apply (rule_tac x=Q in exI)
   apply (rule_tac x=A in exI)
   apply clarsimp
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp

  apply(case_tac P, simp_all)
```

```
  apply (rename_tac P'' Q'' A'' x y)
  apply (rule_tac x=Q'' in exI)
  apply (rule_tac x=A'' in exI)
  apply clarsimp
  apply (rule conjI[rotated])
   apply blast
  apply (erule SeqConseq[rotated])
    apply simp
   apply simp
  apply blast
 done

lemma oghoare_Seq[rule_format, OF _ refl]:
"Γ, Θ ⊢_/F P x Q,A ⟹ ∀p1 p2. x = Seq p1 p2 ⟶
 (∃ P_1 P_2 P' Q' A'. P = AnnComp P_1 P_2 ∧ Γ, Θ ⊢_/F P_1 p1 P', A' ∧ P' ⊆
pre P_2 ∧
         Γ, Θ ⊢_/F P_2 p2 Q',A' ∧
          Q' ⊆ Q ∧ A' ⊆ A)"
  apply (induct rule: oghoare_induct, simp_all)
   apply blast
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp
  apply (rename_tac P'' Q'' A'')
  apply(case_tac P, simp_all)
  apply clarsimp
  apply (rule_tac x="P''" in exI)
  apply (rule_tac x="Q''" in exI)
  apply (rule_tac x="A''" in exI)
  apply clarsimp
  apply (rule conjI)
   apply (rule Conseq)
   apply (rule_tac x="P'" in exI)
   apply (rule_tac x="P''" in exI)
   apply (rule_tac x="A''" in exI)
   apply simp
  apply fastforce
 done

lemma oghoare_Catch[rule_format, OF _ refl]:
"Γ, Θ ⊢_/F P x Q,A ⟹ ∀p1 p2. x = Catch p1 p2 ⟶
 (∃ P_1 P_2 P' Q' A'. P = AnnComp P_1 P_2 ∧ Γ, Θ ⊢_/F P_1 p1 Q', P' ∧ P' ⊆
pre P_2 ∧
         Γ, Θ ⊢_/F P_2 p2 Q',A' ∧
          Q' ⊆ Q ∧ A' ⊆ A)"
  apply (induct rule: oghoare_induct, simp_all)
   apply blast
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp
  apply(case_tac P, simp_all)
```

```
  apply clarsimp
  apply (rename_tac P'' Q'' A'' x)
  apply (rule_tac x="P''" in exI)
  apply (rule_tac x="Q''" in exI)
  apply clarsimp
   apply (rule conjI)
   apply (rule Conseq)
   apply (rule_tac x=P' in exI)
   apply fastforce
  apply (rule_tac x="A''" in exI)
  apply clarsimp
  apply fastforce
 done

lemma oghoare_Cond[rule_format, OF _ refl]:
 "Γ, Θ ⊢/F P x Q,A ⟹
∀c1 c2 b. x = (Cond b c1 c2) ⟶
(∃P' P1 P2 Q' A'. P = (AnnBin P' P1 P2) ∧
  P' ⊆ {s. (s∈b ⟶ s∈pre P1) ∧ (s∉b ⟶ s∈pre P2)} ∧
    Γ, Θ ⊢/F P1 c1 Q',A' ∧
    Γ, Θ ⊢/F P2 c2 Q',A' ∧ Q' ⊆ Q ∧ A' ⊆ A)"
  apply (induct rule: oghoare_induct, simp_all)
   apply (rule conjI)
    apply fastforce
   apply (rename_tac  Q A P2 c2 r b)
   apply (rule_tac x=Q in exI)
   apply (rule_tac x=A in exI)
   apply fastforce
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp
  apply (case_tac P, simp_all)
  apply fastforce
done

lemma oghoare_While[rule_format, OF _ refl]:
  "Γ, Θ ⊢/F P x Q,A ⟹
    ∀ b c. x = While b c ⟶
    (∃ r i P' A' Q'. P = AnnWhile r i P' ∧
     Γ, Θ⊢/F P' c i,A' ∧
     r ⊆ i ∧
     i ∩ b ⊆ pre P' ∧
     i ∩ -b ⊆ Q' ∧
      Q' ⊆ Q ∧ A' ⊆ A)"
  apply (induct rule: oghoare_induct, simp_all)
   apply blast
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp
  apply (rename_tac P' Q' A' b ca r i P'' A'' Q'')
  apply (case_tac P; simp)
```

```
  apply (rule_tac x= A'' in exI)
  apply (rule conjI)
   apply blast
  apply clarsimp
  apply (rule_tac x= "Q'" in exI)
  by fast


lemma oghoare_Call:
 "Γ,Θ⊢_/F P x Q,A ⟹
  ∀p. x = Call p ⟶
 (∃r n.
 P = (AnnCall r n) ∧
 (∃as P' f b.
 Θ p = Some as ∧
 (as ! n) = P' ∧
 r ⊆ pre P' ∧
 Γ p = Some b ∧
 n < length as ∧
 Γ,Θ ⊢_/F P' b Q,A))"
  apply (induct rule: oghoare_induct, simp_all)
  apply (rename_tac Γ Θ F P c Q A)
  apply clarsimp
  apply (case_tac P, simp_all)
  apply clarsimp
  apply (rule Conseq)
  apply (rule_tac x="{}" in exI)
  apply force
 done

lemma oghoare_Parallel[rule_format, OF _ refl]:
"Γ, Θ⊢_/F P x Q,A ⟹ ∀cs. x = Parallel cs ⟶
 (∃as. P = AnnPar as ∧
   length as = length cs ∧
   ⋂(set (map postcond as)) ⊆ Q ∧
   ⋃(set (map abrcond as)) ⊆ A ∧
   (∀i<length cs. ∃Q' A'. Γ,Θ⊢_/F (pres (as!i)) (cs!i) Q', A' ∧
            Q' ⊆ postcond (as!i) ∧ A' ⊆ abrcond (as!i)) ∧
  interfree Γ Θ F as cs)"
  apply (induct rule: oghoare_induct, simp_all)
   apply clarsimp
   apply (drule_tac x=i in spec)
   apply fastforce
  apply clarsimp
  apply (case_tac P, simp_all)
  apply blast
 done

lemma  ann_matches_weaken[OF _ refl]:
```

```
" ann_matches Γ Θ X c ⟹ X = (weaken_pre P P') ⟹ ann_matches Γ Θ P
c"
  apply (induct arbitrary: P P' rule: ann_matches.induct)
   apply (case_tac P, simp_all, fastforce simp add: ann_matches.intros)+
done


lemma oghoare_seq_imp_ann_matches:
 " Γ,Θ⊩/F P a c Q,A ⟹ ann_matches Γ Θ a c"
  apply (induct rule: oghoare_seq_induct, simp_all add: ann_matches.intros)
  apply (clarsimp, erule ann_matches_weaken)+
 done

lemma oghoare_imp_ann_matches:
 " Γ,Θ⊢/F a c Q,A ⟹ ann_matches Γ Θ a c"
  apply (induct rule: oghoare_induct, simp_all add: ann_matches.intros
oghoare_seq_imp_ann_matches)
  apply (clarsimp, erule ann_matches_weaken)+
 done



lemma SkipRule: "P ⊆ Q ⟹ Γ, Θ ⊢/F (AnnExpr P) Skip Q, A"
  apply (rule Conseq, simp)
  apply (rule exI, rule exI, rule exI)
  apply (rule conjI, rule Skip, auto)
done

lemma ThrowRule: "P ⊆ A ⟹ Γ, Θ ⊢/F (AnnExpr P) Throw Q, A"
  apply (rule Conseq, simp)
  apply (rule exI, rule exI, rule exI)
  apply (rule conjI, rule Throw, auto)
done

lemma BasicRule: "P ⊆ {s. (f s) ∈ Q} ⟹ Γ, Θ ⊢/F (AnnExpr P) (Basic
f) Q, A"
  apply (rule Conseq, simp, rule exI[where x="{s. (f s) ∈ Q}"])
  apply (rule exI[where x=Q], rule exI[where x=A])
  apply simp
  apply (subgoal_tac "(P ∪ {s. f s ∈ Q}) = {s. f s ∈ Q}")
  apply (auto intro: Basic)
done

lemma SpecRule:
  "P ⊆ {s. (∀t. (s, t) ∈ r ⟶ t ∈ Q) ∧ (∃t. (s, t) ∈ r)}
    ⟹ Γ, Θ ⊢/F (AnnExpr P) (Spec r) Q, A"
  apply (rule Conseq, simp, rule exI[where x="{s. (∀t. (s, t) ∈ r ⟶
t ∈ Q) ∧ (∃t. (s, t) ∈ r) }"])
  apply (rule exI[where x=Q], rule exI[where x=A])
```

```
  apply simp
  apply (subgoal_tac "(P ∪ {s. (∀t. (s, t) ∈ r ⟶ t ∈ Q) ∧ (∃t. (s,
t) ∈ r)}) = {s. (∀t. (s, t) ∈ r ⟶ t ∈ Q) ∧ (∃t. (s, t) ∈ r)}")
   apply (auto intro: Spec)
done

lemma CondRule:
 "⟦ P ⊆ {s. (s∈b ⟶ s∈pre P1) ∧ (s∉b ⟶ s∈pre P2)};
    Γ, Θ ⊢/F P1 c1 Q,A;
    Γ, Θ ⊢/F P2 c2 Q,A  ⟧
  ⟹ Γ, Θ ⊢/F (AnnBin P P1 P2) (Cond b c1 c2) Q,A"
  by (auto intro: Cond)

lemma WhileRule: "⟦ r ⊆ I; I ∩ b ⊆ pre P; (I ∩ -b) ⊆ Q;
                    Γ, Θ ⊢/F P c I,A ⟧
       ⟹ Γ, Θ ⊢/F (AnnWhile r I P) (While b c) Q,A"
  by (simp add: While)

lemma AwaitRule:
 "⟦atom_com c ; Γ, Θ ⊩/FP a c Q,A ; (r ∩ b) ⊆ P⟧ ⟹
  Γ, Θ ⊢/F (AnnRec r a) (Await b c) Q,A"
  apply (erule Await[rotated])
  apply (erule (1) SeqConseq, simp+)
 done

lemma rtranclp_1n_induct [consumes 1, case_names base step]:
  "⟦r** a b; P a; ⋀y z. ⟦r y z; r** z b; P y⟧ ⟹ P z⟧ ⟹ P b"
 by (induct rule: rtranclp.induct)
    (simp add: rtranclp.rtrancl_into_rtrancl)+

lemmas rtranclp_1n_induct2[consumes 1, case_names base step] =
  rtranclp_1n_induct[of _ "(ax,ay)" "(bx,by)", split_rule]

lemma oghoare_seq_valid:
 " Γ⊨/F P c1 R,A ⟹
   Γ⊨/F R c2 Q,A ⟹
   Γ⊨/F P Seq c1 c2 Q,A"
  apply (clarsimp simp add: valid_def)
  apply (rename_tac t c' s)
  apply (case_tac t)
    apply simp
    apply (drule (1) Seq_decomp_star)
    apply (erule disjE)
     apply fastforce
    apply clarsimp
    apply (rename_tac s1 t')
    apply (drule_tac x="Normal s" and y="Normal t'" in spec2)
    apply (erule_tac x="Skip" in allE)
    apply (fastforce simp: final_def)
```

```
   apply (clarsimp simp add: final_def)
   apply (drule Seq_decomp_star_Fault)
   apply (erule disjE)
    apply (rename_tac s2)
    apply (drule_tac x="Normal s" and y="Fault s2" in spec2)
    apply (erule_tac x="Skip" in allE)
    apply fastforce
   apply clarsimp
   apply (rename_tac s s2 s')
   apply (drule_tac x="Normal s" and y="Normal s'" in spec2)
   apply (erule_tac x="Skip" in allE)
   apply clarsimp
   apply (drule_tac x="Normal s'" and y="Fault s2" in spec2)
   apply (erule_tac x="Skip" in allE)
   apply clarsimp
  apply clarsimp
  apply (simp add: final_def)
  apply (drule Seq_decomp_star_Stuck)
  apply (erule disjE)
   apply fastforce
  apply clarsimp
  apply fastforce
 done

lemma oghoare_if_valid:
 "Γ⊨/F P1 c1 Q,A ⟹
  Γ⊨/F P2 c2 Q,A ⟹
  r ∩ b ⊆ P1 ⟹ r ∩ - b ⊆ P2 ⟹
  Γ⊨/F r Cond b c1 c2 Q,A"
  apply (simp (no_asm) add: valid_def)
  apply (clarsimp)
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply (erule step_Normal_elim_cases)
   apply (fastforce simp: valid_def[where c=c1])
  apply (fastforce simp: valid_def[where c=c2])
 done

lemma Skip_normal_steps_end:
"Γ ⊢ (Skip, Normal s) →* (c, s') ⟹ c = Skip ∧ s' = Normal s"
  apply  (erule converse_rtranclpE)
   apply simp
  apply (erule step_Normal_elim_cases)
 done

lemma Throw_normal_steps_end:
"Γ ⊢ (Throw, Normal s) →* (c, s') ⟹ c = Throw ∧ s' = Normal s"
  apply  (erule converse_rtranclpE)
   apply simp
```

```
  apply (erule step_Normal_elim_cases)
 done

lemma while_relpower_induct:
 "⋀t c' x .
       Γ⊨/F P c i,A ⟹
       i ∩ b ⊆ P ⟹
       i ∩ - b ⊆ Q ⟹
       final (c', t) ⟹
       x ∈ i ⟹
       t ∉ Fault ‘ F ⟹
       c' = Throw ⟶ t ∉ Normal ‘ A ⟹
       (step Γ ^^ n) (While b c, Normal x) (c', t) ⟹ c' = Skip ∧ t ∈
Normal ‘ Q"
  apply (induct n rule:less_induct)
  apply (rename_tac n t c' x)
  apply (case_tac n)
   apply (clarsimp simp: final_def)
  apply clarify
  apply (simp only: relpowp.simps)
  apply (subst (asm) relpowp_commute[symmetric])
  apply clarsimp
  apply (erule step_Normal_elim_cases)
   apply clarsimp
   apply (rename_tac t c' x v)
   apply(case_tac "t")
    apply clarsimp
    apply(drule Seq_decomp_relpow)
     apply(simp add: final_def)
     apply(erule disjE, erule conjE)
      apply clarify
      apply(drule relpowp_imp_rtranclp)
      apply (simp add: valid_def)
      apply (rename_tac x n t' n1)
      apply (drule_tac x="Normal x" in spec)
      apply (drule_tac x="Normal t'" in spec)
      apply (drule spec[where x=Throw])
      apply (fastforce simp add: final_def)
     apply clarsimp
     apply (rename_tac c' x n t' t n1 n2)
     apply (drule_tac x=n2 and y="Normal t'" in meta_spec2)
     apply (drule_tac x=c' and y="t" in meta_spec2)
     apply (erule meta_impE, fastforce)
     apply (erule meta_impE, fastforce)
     apply (erule meta_impE)
      apply(drule relpowp_imp_rtranclp)
      apply (simp add: valid_def)
      apply (drule_tac x="Normal x" and y="Normal t" in spec2)
      apply (drule spec[where x=Skip])
```

```
    apply (fastforce simp add: final_def)
   apply assumption
  apply clarsimp
  apply (rename_tac c' s n f)
  apply (subgoal_tac "c' = Skip", simp)
   prefer 2
   apply (case_tac c'; fastforce simp: final_def)
  apply (drule Seq_decomp_relpowp_Fault)
  apply (erule disjE)
   apply (clarsimp simp: valid_def)
   apply (drule_tac x="Normal s" and y="Fault f" in spec2)
   apply (drule spec[where x=Skip])
   apply(drule relpowp_imp_rtranclp)
   apply (fastforce simp: final_def)
  apply clarsimp
  apply (rename_tac t n1 n2)
  apply (subgoal_tac "t ∈ i")
   prefer 2
   apply (fastforce dest:relpowp_imp_rtranclp simp: final_def valid_def)
  apply (drule_tac x=n2 in meta_spec)
  apply (drule_tac x="Fault f" in meta_spec)
  apply (drule meta_spec[where x=Skip])
  apply (drule_tac x=t in meta_spec)
  apply (fastforce simp: final_def)
 apply clarsimp
 apply (rename_tac c' s n)
 apply (subgoal_tac "c' = Skip", simp)
  prefer 2
  apply (case_tac c'; fastforce simp: final_def)
  apply (drule Seq_decomp_relpowp_Stuck)
  apply clarsimp
  apply (erule disjE)
   apply (simp add:valid_def)
   apply (drule_tac x="Normal s" and y="Stuck" in spec2)
   apply clarsimp
   apply (drule spec[where x=Skip])
   apply(drule relpowp_imp_rtranclp)
   apply (fastforce)
  apply clarsimp
 apply (rename_tac t n1 n2)
 apply (subgoal_tac "t ∈ i")
  prefer 2
  apply (fastforce dest:relpowp_imp_rtranclp simp: final_def valid_def)
 apply (drule_tac x=n2 in meta_spec)
 apply (drule meta_spec[where x=Stuck])
 apply (drule meta_spec[where x=Skip])
 apply (drule_tac x=t in meta_spec)
 apply (fastforce simp: final_def)
apply clarsimp
```

```
  apply (drule relpowp_imp_rtranclp)
  apply (drule Skip_normal_steps_end)
  apply fastforce
done

lemma valid_weaken_pre:
 "Γ⊨/F P c Q,A ⟹
  P' ⊆ P ⟹ Γ⊨/F P' c Q,A"
 by (fastforce simp: valid_def)

lemma valid_strengthen_post:
 "Γ⊨/F P c Q,A ⟹
  Q ⊆ Q' ⟹ Γ⊨/F P c Q',A"
 by (fastforce simp: valid_def)

lemma valid_strengthen_abr:
 "Γ⊨/F P c Q,A ⟹
  A ⊆ A' ⟹ Γ⊨/F P c Q,A'"
 by (fastforce simp: valid_def)

lemma oghoare_while_valid:
 "Γ⊨/F P c i,A ⟹
  i ∩ b ⊆ P ⟹
  i ∩ - b ⊆ Q ⟹
  Γ⊨/F i While b c Q,A"
  apply (simp (no_asm) add: valid_def)
  apply (clarsimp simp add: )
  apply (drule rtranclp_imp_relpowp)
  apply (clarsimp)
  by (erule while_relpower_induct)

lemma oghoare_catch_valid:
 "Γ⊨/F P1 c1 Q,P2 ⟹
  Γ⊨/F P2 c2 Q,A ⟹
    Γ⊨/F P1 Catch c1 c2 Q,A"
  apply (clarsimp simp add: valid_def[where c="Catch _ _"])
  apply (rename_tac t c' s)
  apply (case_tac t)
    apply simp
    apply (drule Catch_decomp_star)
     apply (fastforce simp: final_def)
    apply clarsimp
    apply (erule disjE)
     apply (clarsimp simp add: valid_def[where c="c1"])
     apply (rename_tac s x t)
     apply (drule_tac x="Normal s" in spec)
     apply (drule_tac x="Normal t" in spec)
     apply (drule_tac x=Throw in spec)
     apply (fastforce simp: final_def valid_def)
```

```
    apply (clarsimp simp add: valid_def[where c="c1"])
    apply (rename_tac s t)
    apply (drule_tac x="Normal s" in spec)
    apply (drule_tac x="Normal t" in spec)
    apply (drule_tac x=Skip in spec)
    apply (fastforce simp: final_def)
   apply (rename_tac c' s t)
   apply (simp add: final_def)
   apply (drule Catch_decomp_star_Fault)
   apply clarsimp
   apply (erule disjE)
    apply (clarsimp simp: valid_def[where c=c1] final_def)
   apply (fastforce simp: valid_def final_def)
  apply (simp add: final_def)
  apply (drule Catch_decomp_star_Stuck)
  apply clarsimp
  apply (erule disjE)
   apply (clarsimp simp: valid_def[where c=c1] final_def)
  apply (fastforce simp: valid_def final_def)
 done

lemma ann_matches_imp_assertionsR:
  "ann_matches Γ Θ a c ⟹ ¬ pre_par a ⟹
    assertionsR Γ Θ Q A a c (pre a)"
  by (induct arbitrary: Q A  rule: ann_matches.induct , (fastforce intro:
assertionsR.intros )+)

lemma ann_matches_imp_assertionsR':
  "ann_matches Γ Θ a c ⟹ a' ∈ pre_set a ⟹
    assertionsR Γ Θ Q A a c a'"
 apply (induct arbitrary: Q A  rule: ann_matches.induct)
  apply ((fastforce intro:  assertionsR.intros )+)[12]
  apply simp
  apply (erule bexE)
  apply (simp only: in_set_conv_nth)
  apply (erule exE)
  apply (drule_tac x=i in spec)
  apply clarsimp
  apply (erule AsParallelExprs)
  apply simp
done

lemma rtranclp_conjD:
  "(λx1 x2. r1 x1 x2 ∧ r2 x1 x2)** x1 x2 ⟹
  r1** x1 x2 ∧ r2** x1 x2"
by (metis (no_types, lifting) rtrancl_mono_proof)

lemma rtranclp_mono' :
"r** a b ⟹ r ≤ s ⟹ s** a b"
```

```
by (metis rtrancl_mono_proof sup.orderE sup2CI)

lemma state_upd_in_atomicsR[rule_format, OF _ refl refl]:
 "Γ⊢ cf → cf' ⟹
   cf = (c, Normal s) ⟹
   cf' = (c', Normal t) ⟹
       s ≠ t ⟹
       ann_matches Γ Θ a c ⟹
       s ∈ pre a ⟹
       (∃p cm x. atomicsR Γ Θ a c (p, cm) ∧ s ∈ p ∧
       Γ ⊢ (cm, Normal s) → (x, Normal t) ∧ final (x, Normal t))"
supply [[simproc del: defined_all]]
  apply (induct arbitrary: c c' s t a rule: step.induct, simp_all)
       apply clarsimp
       apply (erule  ann_matches.cases, simp_all)
       apply (rule exI)+
       apply (rule conjI)
        apply (rule atomicsR.intros)
        apply clarsimp
        apply (rule_tac x="Skip" in exI)
        apply (simp add: final_def)
        apply (rule step.Basic)
       apply clarsimp
       apply (erule  ann_matches.cases, simp_all)
       apply (rule exI)+
       apply (rule conjI)
        apply (rule atomicsR.intros)
        apply clarsimp
        apply (rule_tac x="Skip" in exI)
        apply (simp add: final_def)
        apply (erule step.Spec)
      apply clarsimp
      apply (erule  ann_matches.cases, simp_all)
      apply clarsimp
      apply (drule meta_spec)+
      apply (erule meta_impE, rule conjI, (rule refl)+)+
      apply clarsimp
      apply (erule (1) meta_impE)
      apply (erule meta_impE, fastforce)
      apply clarsimp
      apply (rule exI)+
      apply (rule conjI)
      apply (erule AtSeqExpr1)
     apply fastforce
    apply clarsimp
    apply (erule  ann_matches.cases, simp_all)
    apply clarsimp
    apply (drule meta_spec)+
    apply (erule meta_impE, rule conjI, (rule refl)+)+
```

```
    apply clarsimp
    apply (erule (1) meta_impE)
    apply (erule meta_impE, fastforce)
    apply clarsimp
    apply (rule exI)+
    apply (rule conjI)
    apply (erule AtCatchExpr1)
    apply fastforce
    apply (erule  ann_matches.cases, simp_all)
    apply clarsimp
    apply (drule meta_spec)+
    apply (erule meta_impE, rule conjI, (rule refl)+)+
    apply clarsimp
    apply (erule meta_impE)
     apply fastforce
    apply (erule meta_impE)
     apply (case_tac "i=0"; fastforce)
    apply clarsimp
    apply (rule exI)+
    apply (rule conjI)
     apply (erule AtParallelExprs)
    apply fastforce
   apply (drule_tac x=i in spec)
   apply clarsimp
   apply fastforce
  apply (erule ann_matches.cases, simp_all)
  apply clarsimp
  apply (rule exI)+
  apply (rule conjI)
   apply (rule AtAwait)
   apply clarsimp
   apply (rename_tac c' sa t aa e r ba)
   apply (rule_tac x=c' in exI)
   apply (rule conjI)
    apply (erule step.Await)
   apply (erule rtranclp_mono')
   apply clarsimp
   apply assumption+
   apply (simp add: final_def)
 done

lemma oghoare_atom_com_sound:
  "Γ, Θ ⊩/FP a c Q,A ⟹ atom_com c ⟹ Γ ⊨/F P c Q, A"
 unfolding valid_def
  proof (induct rule: oghoare_seq_induct)
    case SeqSkip thus ?case
      by (fastforce
            elim: converse_rtranclpE step_Normal_elim_cases(1))
  next
```

```
  case SeqThrow thus ?case
  by (fastforce
         elim: converse_rtranclpE step_Normal_elim_cases)
next
  case SeqBasic thus ?case
    by (fastforce
         elim: converse_rtranclpE step_Normal_elim_cases
         simp: final_def)
next
  case (SeqSpec Γ Θ F r Q A) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply (erule step_Normal_elim_cases)
   apply (fastforce elim!: converse_rtranclpE step_Normal_elim_cases)
  by clarsimp
next
  case (SeqSeq Γ Θ F P₁ c₁ P₂ A c₂ Q) show ?case
   using SeqSeq
   by (fold valid_def)
      (fastforce intro: oghoare_seq_valid simp: valid_weaken_pre)
 next
  case (SeqCatch Γ Θ F P₁ c₁ Q P₂ c₂ A) thus ?case
  apply (fold valid_def)
  apply simp
  apply (fastforce elim:  oghoare_catch_valid)+
  done
 next
  case (SeqCond Γ Θ F P b c1 Q A c2) thus ?case
   by (fold valid_def)
      (fastforce intro:oghoare_if_valid)
 next
  case (SeqWhile Γ Θ F P c A b) thus ?case
   by (fold valid_def)
      (fastforce elim: valid_weaken_pre[rotated] oghoare_while_valid)
 next
  case (SeqGuard Γ Θ F P c Q A r g f) thus ?case
  apply (fold valid_def)
  apply (simp (no_asm) add: valid_def)
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (fastforce simp: final_def)
  apply clarsimp
  apply (erule step_Normal_elim_cases)
   apply (case_tac "r ∩ - g ≠ {}")
    apply clarsimp
    apply (fastforce simp: valid_def)
   apply clarsimp
   apply (fastforce simp: valid_def)
```

```
    apply clarsimp
     apply (case_tac "r ∩ - g ≠ {}")
      apply (fastforce dest: no_steps_final simp:final_def)
     apply (fastforce dest: no_steps_final simp:final_def)
   done
   next
    case (SeqCall Γ p f Θ F P Q A) thus ?case
     by simp
   next

    case (SeqDynCom r fa Γ Θ F P c Q A) thus ?case
    apply -
    apply clarsimp
    apply (erule converse_rtranclpE)
     apply (clarsimp simp: final_def)
    apply clarsimp
    apply (erule step_Normal_elim_cases)
    apply clarsimp
    apply (rename_tac t c' x)

    apply (drule_tac x=x in spec)
    apply (drule_tac x=x in bspec, fastforce)
    apply clarsimp
    apply (drule_tac x="Normal x" in spec)
    apply (drule_tac x="t" in spec)
    apply (drule_tac x="c'" in spec)
    apply fastforce+
   done
   next
    case (SeqConseq Γ Θ F P c Q A) thus ?case
     apply (clarsimp)
     apply (rename_tac t c' x)
     apply (erule_tac x="Normal x" in allE)
     apply (erule_tac x="t" in allE)
     apply (erule_tac x="c'" in allE)
     apply (clarsimp simp: pre_weaken_pre)
     apply force
    done
qed simp_all

lemma ParallelRuleAnn:
" length as = length cs ⟹
  ∀i<length cs. Γ,Θ ⊢/F (pres (as ! i)) (cs ! i) (postcond (as ! i)),(abrcond
(as ! i)) ⟹
  interfree Γ Θ F as cs ⟹
  ⋂(set (map postcond as)) ⊆ Q ⟹
  ⋃(set (map abrcond as)) ⊆ A ⟹ Γ,Θ ⊢/F (AnnPar as) (Parallel cs)
Q,A"
 apply (erule (3) Parallel)
```

```
 apply auto
done

lemma oghoare_step[rule_format, OF _ refl refl]:
shows
 "Γ ⊢ cf → cf' ⟹ cf = (c, Normal s) ⟹ cf' = (c', Normal t) ⟹
  Γ,Θ⊢/F a c Q,A ⟹
 s ∈ pre a ⟹
∃a'. Γ,Θ⊢/F a' c' Q,A ∧ t ∈ pre a' ∧
        (∀as. assertionsR Γ Θ Q A a' c' as ⟶ assertionsR Γ Θ Q A a
c as) ∧
        (∀pm cm. atomicsR Γ Θ a' c' (pm, cm) ⟶ atomicsR Γ Θ  a c (pm,
cm))"
 proof (induct arbitrary:c c' s a t Q A rule: step.induct)
   case (Parallel i cs s c' s' ca c'a sa a t Q A) thus ?case
     supply [[simproc del: defined_all]]
     apply (clarsimp simp:)
     apply (drule oghoare_Parallel)
     apply clarsimp
     apply (rename_tac as)
     apply (frule_tac x=i in spec, erule (1) impE)
     apply (elim exE conjE)


     apply (drule meta_spec)+
     apply (erule meta_impE, rule conjI, (rule refl)+)+
     apply (erule meta_impE)
      apply (rule_tac P="(pres (as ! i))" in Conseq)
      apply (rule exI[where x="{}"])
      apply (rule_tac x="Q'" in exI)
      apply (rule_tac x="A'" in exI)
      apply (simp)
     apply (erule meta_impE, simp)
     apply clarsimp
     apply (rule_tac x="AnnPar (as[i:=(a',postcond(as!i), abrcond(as!i))])"
in exI)
     apply (rule conjI)
      apply (rule ParallelRuleAnn, simp)
          apply clarsimp
          apply (rename_tac j)
          apply (drule_tac x=j in spec)
          apply clarsimp
          apply (case_tac "i = j")
           apply (clarsimp simp: )
           apply (rule Conseq)
           apply (rule exI[where x="{}"])
           apply (fastforce)
          apply simp
          apply (clarsimp simp: interfree_def)
```

```
       apply (rename_tac i' j')
       apply (drule_tac x=i' and y=j' in spec2)
       apply clarsimp
       apply (case_tac "i = i'")
        apply clarsimp
        apply (simp add: interfree_aux_def prod.case_eq_if )
       apply clarsimp
       apply (case_tac "j' = i")
        apply (clarsimp)
        apply (simp add: interfree_aux_def prod.case_eq_if)
        apply clarsimp
       apply (clarsimp)
      apply (erule subsetD)
      apply (clarsimp simp: in_set_conv_nth)
      apply (rename_tac a' x a b c i')
      apply (case_tac "i' = i")
       apply clarsimp
       apply (drule_tac x="(a', b, c)" in bspec, simp)
        apply (fastforce simp add: in_set_conv_nth)
       apply fastforce
      apply (drule_tac x="(a, b, c)" in bspec, simp)
       apply (simp add: in_set_conv_nth)
       apply (rule_tac x=i' in exI)
       apply clarsimp
      apply fastforce
  apply clarsimp
  apply (erule_tac A="(⋃x∈set as. abrcond x) " in  subsetD)
  apply (clarsimp simp: in_set_conv_nth)
  apply (rename_tac a b c j)
  apply (case_tac "j = i")
   apply clarsimp
   apply (rule_tac x="as!i" in bexI)
    apply simp
  apply clarsimp
 apply clarsimp
 apply (rule_tac x="(a,b,c)" in bexI)
  apply simp
 apply (clarsimp simp:in_set_conv_nth)
 apply (rule_tac x=j in exI)
 apply fastforce
apply (rule conjI)
 apply (case_tac "s = Normal t")
  apply clarsimp
  apply (clarsimp simp: in_set_conv_nth)
  apply (rename_tac a b c j)
  apply (case_tac "j = i")
   apply clarsimp
  apply clarsimp
  apply (drule_tac x="as!j" in bspec)
```

```
      apply (clarsimp simp add: in_set_conv_nth)
      apply (rule_tac x=j in exI)
      apply fastforce
     apply clarsimp
    apply (frule state_upd_in_atomicsR, simp)
      apply (erule oghoare_imp_ann_matches)
     apply (clarsimp simp: in_set_conv_nth)
     apply fastforce
    apply (clarsimp simp: in_set_conv_nth)
    apply (rename_tac j)
    apply (case_tac "j = i")
     apply clarsimp
    apply clarsimp
    apply (thin_tac "Γ,Θ ⊢/F a' c'  (postcond (as ! i)),(abrcond (as
! i))")
    apply (simp add: interfree_def interfree_aux_def)
    apply (drule_tac x="j" and y=i in spec2)
    apply (simp add: prod.case_eq_if)
    apply (drule spec2, drule (1) mp)
    apply clarsimp
    apply (case_tac "pre_par a")
     apply (subst pre_set)
     apply clarsimp
     apply (drule_tac x="as!j" in bspec)
      apply (clarsimp simp: in_set_conv_nth)
      apply (rule_tac x=j in exI)
      apply fastforce
     apply clarsimp
     apply (frule (1) pre_imp_pre_set)
     apply (rename_tac as Q' A' a' a b c p cm x j X)
     apply (drule_tac x="X" in spec, erule_tac P="assertionsR Γ Θ b c
a (cs ! j) X" in impE)
      apply (rule ann_matches_imp_assertionsR')
       apply (drule_tac x=j in spec, clarsimp)
       apply (erule (1) oghoare_imp_ann_matches)
     apply (rename_tac a b c p cm x j X )
     apply (thin_tac "Γ⊨/F (b ∩ p) cm b,b")
     apply (thin_tac " Γ⊨/F (c ∩ p) cm c,c")
     apply (simp add: valid_def)
     apply (drule_tac x="Normal sa" in spec)
     apply (drule_tac x="Normal t" in spec)
     apply (drule_tac x=x in spec)
     apply (erule impE, fastforce)
     apply force

   apply (drule_tac x=j in spec)
   apply clarsimp
   apply (rename_tac a b c p cm x j Q'' A'')
   apply (drule_tac x="pre a" in spec,erule impE, rule ann_matches_imp_assertionsR)
```

```
   apply (erule (1) oghoare_imp_ann_matches)
   apply (thin_tac " Γ⊨_/F (b ∩ p) cm b,b")
   apply (thin_tac " Γ⊨_/F (c ∩ p) cm c,c")
  apply (simp add: valid_def)
  apply (drule_tac x="Normal sa" in spec)
  apply (drule_tac x="Normal t" in spec)
  apply (drule_tac x=x in spec)
  apply (erule impE, fastforce)
  apply clarsimp
  apply (drule_tac x="as ! j" in bspec)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rule_tac x=j in exI, fastforce)
  apply clarsimp
  apply fastforce
apply (rule conjI)
  apply (clarsimp simp: )
  apply (erule assertionsR.cases ; simp)
    apply (clarsimp simp: )
    apply (rename_tac j a)
    apply (case_tac "j = i")
     apply clarsimp
     apply (drule_tac x=a in spec, erule (1) impE)
     apply (erule (1)  AsParallelExprs)
    apply (subst (asm) nth_list_update_neq, simp)
    apply (erule_tac i=j in  AsParallelExprs)
    apply fastforce
   apply clarsimp
  apply (rule  AsParallelSkips)
  apply (clarsimp simp:)
  apply (rule equalityI)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rename_tac a' x a b c j)
   apply (case_tac "j = i")
    apply (thin_tac "∀a∈set as. sa ∈ precond a")
    apply clarsimp
    apply (drule_tac x="(a', b, c)" in bspec)
    apply (clarsimp simp: in_set_conv_nth)
    apply (rule_tac x="i" in exI)
    apply fastforce
   apply fastforce
  apply (drule_tac x="as ! j" in bspec)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rule_tac x=j in exI)
   apply fastforce
 apply clarsimp
 apply (drule_tac x=" as ! j" in bspec)
  apply (clarsimp simp: in_set_conv_nth)
  apply (rule_tac x=j in exI, fastforce)
  apply fastforce
```

```
   apply (clarsimp simp: in_set_conv_nth)
   apply (rename_tac x a b c j)
   apply (thin_tac "∀a∈set as. sa ∈ precond a")
   apply (case_tac "j = i")
    apply clarsimp
    apply (drule_tac x="as!i" in bspec, fastforce)
    apply fastforce
   apply clarsimp
   apply (drule_tac x="as!j" in bspec)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rule_tac x=j in exI, fastforce)
   apply fastforce
  apply clarsimp
  apply (erule atomicsR.cases ; simp)
  apply clarsimp
  apply (rename_tac j atc atp)
  apply (case_tac "j = i")
   apply clarsimp
   apply (drule_tac x=atc and y=atp in spec2, erule impE)
    apply (clarsimp)
   apply (erule (1)  AtParallelExprs)
  apply (subst (asm) nth_list_update_neq, simp)
  apply (erule_tac i=j in  AtParallelExprs)
  apply (clarsimp)
 done
next
 case (Basic f s c c' sa a t Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Basic)
  apply clarsimp
  apply (rule_tac x="AnnExpr Q" in exI)
  apply clarsimp
  apply (rule conjI)
   apply (rule SkipRule)
   apply fastforce
  apply (rule conjI)
   apply fastforce
  apply clarsimp
  apply (drule assertionsR.cases, simp_all)
  apply (rule assertionsR.AsBasicSkip)
 done
next
  case (Spec s t r c c' sa a ta Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Spec)
  apply clarsimp
  apply (rule_tac x="AnnExpr Q" in exI)
  apply clarsimp
  apply (rule conjI)
```

```
   apply (rule SkipRule)
   apply fastforce
  apply (rule conjI)
   apply force
  apply clarsimp
  apply (erule assertionsR.cases, simp_all)
  apply clarsimp
  apply (rule assertionsR.AsSpecSkip)
 done
next
  case (Guard s g f c ca c' sa a t Q A) thus ?case
  apply -
  apply clarsimp
  apply (drule oghoare_Guard)
  apply clarsimp
  apply (rule exI, rule conjI, assumption)
   by (fastforce dest: oghoare_Guard
                intro:assertionsR.intros atomicsR.intros)
next
  case (GuardFault s g f c ca c' sa a t Q A) thus ?case
   by (fastforce dest: oghoare_Guard
                intro:assertionsR.intros atomicsR.intros)
next
  case (Seq c1 s c1' s' c2 c c' sa a t A Q) thus ?case
    supply [[simproc del: defined_all]]
    apply (clarsimp simp:)
    apply (drule oghoare_Seq)
    apply clarsimp
    apply (drule meta_spec)+
    apply (erule meta_impE, rule conjI, (rule refl)+)+
    apply (erule meta_impE)
     apply (rule Conseq)
     apply (rule exI[where x="{}"])
     apply (rule exI)+
     apply (rule conjI)
      apply (simp)
     apply (erule (1) conjI)
    apply clarsimp
    apply (rule_tac x="AnnComp a' P2" in exI, rule conjI)
     apply (rule oghoare_oghoare_seq.Seq)
      apply (rule Conseq)
      apply (rule_tac x="{}" in exI)
      apply (fastforce)
     apply (rule Conseq)
     apply (rule_tac x="{}" in exI)
     apply (fastforce)
    apply clarsimp
    apply (rule conjI)
     apply clarsimp
```

```
     apply (erule assertionsR.cases, simp_all)
      apply clarsimp
      apply (drule_tac x=a in spec, simp)
      apply (erule AsSeqExpr1)
      apply clarsimp
     apply (erule AsSeqExpr2)
    apply clarsimp
    apply (erule atomicsR.cases, simp_all)
      apply clarsimp
      apply (drule_tac x="a" and y=b in spec2, simp)
      apply (erule AtSeqExpr1)
      apply clarsimp
     apply (erule AtSeqExpr2)
   done
next
  case (SeqSkip c2 s c c' sa a t Q A) thus ?case
   apply clarsimp
   apply (drule oghoare_Seq)
    apply clarsimp
    apply (rename_tac P1 P2 P' Q' A')
    apply (rule_tac x=P2 in exI)
    apply (rule conjI, rule Conseq)
     apply (rule_tac x="{}" in exI)
     apply (fastforce)
    apply (rule conjI)
     apply (drule oghoare_Skip)
     apply fastforce
    apply (rule conjI)
     apply clarsimp
    apply (erule assertionsR.AsSeqExpr2)
   apply clarsimp
   apply (fastforce intro: atomicsR.intros)
  done
next
  case (SeqThrow c2 s c c' sa a t Q A) thus ?case
   apply clarsimp
   apply (drule oghoare_Seq)
   apply clarsimp
   apply (rename_tac P1 P2 P' Q' A')
   apply (rule_tac x=P1 in exI)
   apply (drule oghoare_Throw)
   apply clarsimp
   apply (rename_tac P'')
   apply (rule conjI, rule Conseq)
    apply (rule_tac x="{}" in exI)
    apply (rule_tac x="Q'" in exI)
    apply (rule_tac x="P''" in exI)
    apply (fastforce intro: Throw)
   apply clarsimp
```

```
   apply (erule assertionsR.cases, simp_all)
   apply clarsimp
   apply (rule AsSeqExpr1)
   apply (rule AsThrow)
  done
next
  case (CondTrue s b c₁ c₂ c sa c' s' ann) thus ?case
   apply (clarsimp)
   apply (drule oghoare_Cond)
   apply clarsimp
   apply (rename_tac P' P₁ P₂ Q' A')
   apply (rule_tac x= P₁ in exI)
   apply (rule conjI)
    apply (rule Conseq, rule_tac x="{}" in exI, fastforce)
   apply (rule conjI, fastforce)
   apply (rule conjI)
    apply (fastforce elim: assertionsR.cases intro: AsCondExpr1)
   apply (fastforce elim: atomicsR.cases intro: AtCondExpr1)
  done
next
  case (CondFalse s b c₁ c₂ c sa c' s' ann) thus ?case
   apply (clarsimp)
   apply (drule oghoare_Cond)
   apply clarsimp
   apply (rename_tac P' P₁ P₂ Q' A')
   apply (rule_tac x= P₂ in exI)
   apply (rule conjI)
    apply (rule Conseq, rule_tac x="{}" in exI, fastforce)
   apply (rule conjI, fastforce)
   apply (rule conjI)
    apply (fastforce elim: assertionsR.cases intro: AsCondExpr2)
   apply (fastforce elim: atomicsR.cases intro: AtCondExpr2)
  done
next
  case (WhileTrue s b c ca sa c' s' ann) thus ?case
  apply clarsimp
  apply (frule oghoare_While)
  apply clarsimp
  apply (rename_tac r i P' A' Q')
  apply (rule_tac x="AnnComp P' (AnnWhile i i P')" in exI)
  apply (rule conjI)
   apply (rule Seq)
     apply (rule Conseq)
     apply (rule_tac x="{}" in exI)
     apply (rule_tac x="i" in exI)
     apply (rule_tac x="A'" in exI)
     apply (subst weaken_pre_empty)
     apply clarsimp
    apply (rule While)
```

```
      apply (rule Conseq)
      apply (rule_tac x="{}" in exI)
      apply (rule_tac x="i" in exI)
      apply (rule_tac x="A'" in exI)
      apply (subst weaken_pre_empty)
      apply clarsimp
     apply clarsimp
    apply force
   apply simp
  apply simp
 apply (rule conjI)
  apply blast
 apply (rule conjI)
  apply clarsimp
  apply (erule assertionsR.cases, simp_all)
   apply clarsimp
   apply (rule AsWhileExpr)
  apply clarsimp
  apply (erule assertionsR.cases,simp_all)
     apply clarsimp
     apply (erule AsWhileExpr)
    apply clarsimp
    apply (rule AsWhileInv)
   apply clarsimp
   apply (rule AsWhileInv)
  apply clarsimp
  apply (rule AsWhileSkip)
 apply clarsimp
 apply (erule atomicsR.cases, simp_all)
  apply clarsimp
  apply (rule AtWhileExpr)
 apply clarsimp+
 apply (erule atomicsR.cases, simp_all)
  apply clarsimp
 apply (erule AtWhileExpr)
 done
next
 case (WhileFalse s b c ca sa c' ann s' Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_While)
  apply clarsimp
  apply (rule_tac x="AnnExpr Q" in exI)
  apply (rule conjI)
   apply (rule SkipRule)
   apply blast
  apply (rule conjI)
   apply fastforce
  apply clarsimp
  apply (erule assertionsR.cases, simp_all)
```

```
   apply (drule sym, simp, rule AsWhileSkip)
  done
next
  case (Call p bs s c c' sa a t Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Call)
  apply clarsimp
  apply (rename_tac n as)
  apply (rule_tac x="as ! n" in exI)
  apply clarsimp
  apply (rule conjI, fastforce)
  apply (rule conjI)
   apply clarsimp
   apply (erule (2) AsCallExpr)
   apply fastforce
  apply clarsimp
  apply (erule (2) AtCallExpr)
  apply simp
 done
next
  case (DynCom c s ca c' sa a t Q A) thus ?case
  apply -
  apply clarsimp
  apply (drule oghoare_DynCom)
  apply clarsimp
  apply (drule_tac x=t in bspec, assumption)
  apply (rule exI)
  apply (erule conjI)
  apply (rule conjI, fastforce)
  apply (rule conjI)
   apply clarsimp
   apply (erule (1) AsDynComExpr)
  apply (clarsimp)
  apply (erule (1) AtDynCom)
  done
next
  case (Catch c1 s c1' s' c2 c c' sa a t Q A) thus ?case
    supply [[simproc del: defined_all]]
    apply (clarsimp simp:)
    apply (drule oghoare_Catch)
    apply clarsimp
    apply (drule meta_spec)+
    apply (erule meta_impE, rule conjI, (rule refl)+)+
    apply (erule meta_impE)
     apply (rule Conseq)
     apply (rule exI[where x="{}"])
     apply (rule exI)+
     apply (rule conjI)
      apply (simp)
```

```
     apply (erule (1) conjI)
    apply clarsimp
    apply (rename_tac P1 P2 P' Q' A' a')
    apply (rule_tac x="AnnComp a' P2" in exI, rule conjI)
     apply (rule oghoare_oghoare_seq.Catch)
      apply (rule Conseq)
      apply (rule_tac x="{}" in exI)
      apply (fastforce)
     apply (rule Conseq)
     apply (rule_tac x="{}" in exI)
     apply (fastforce)
    apply clarsimp
   apply (rule conjI)
    apply clarsimp
     apply (erule assertionsR.cases, simp_all)
      apply clarsimp
      apply (rename_tac a)
      apply (drule_tac x=a in spec, simp)
      apply (erule AsCatchExpr1)
     apply clarsimp
     apply (erule AsCatchExpr2)
    apply clarsimp
    apply (erule atomicsR.cases, simp_all)
      apply clarsimp
      apply (rename_tac a b a2)
      apply (drule_tac x="a" and y=b in spec2, simp)
      apply (erule AtCatchExpr1)
      apply clarsimp
     apply (erule AtCatchExpr2)
   done
next
  case (CatchSkip c2 s c c' sa a t Q A) thus ?case
   apply clarsimp
   apply (drule oghoare_Catch, clarsimp)
   apply (rename_tac P1 P2 P' Q' A')
   apply (rule_tac x=P1 in exI)
   apply clarsimp
   apply (rule conjI)
    apply (rule Conseq)
    apply (rule_tac x="{}" in exI)
    apply (drule oghoare_Skip)
    apply clarsimp
    apply (rule_tac x=Q' in exI)
    apply (rule_tac x=A' in exI)
    apply (rule conjI, erule SkipRule)
    apply clarsimp
   apply clarsimp
   apply (rule AsCatchExpr1)
   apply (erule assertionsR.cases, simp_all)
```

```
   apply (rule assertionsR.AsSkip)
  done
next
  case (CatchThrow c₂ s c c' sa a t Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Catch, clarsimp)
  apply (rename_tac P₁ P₂ P' Q' A')
  apply (rule_tac x=P₂ in exI)
  apply (rule conjI)
   apply (rule Conseq)
   apply (rule_tac x="{}" in exI)
   apply (fastforce )
  apply (rule conjI)
   apply (drule oghoare_Throw)
   apply clarsimp
   apply fastforce
  apply (rule conjI)
   apply (clarsimp)
   apply (erule AsCatchExpr2)
  apply clarsimp
  apply (erule AtCatchExpr2)
 done
next
 case (ParSkip cs s c c' sa a t Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Parallel)
  apply clarsimp
  apply (rename_tac as)

  apply (rule_tac x="AnnExpr (⋂x∈set as. postcond x)" in exI)
  apply (rule conjI, rule SkipRule)
   apply blast
  apply (rule conjI)
   apply simp
    apply clarsimp
    apply (simp only: in_set_conv_nth)
    apply clarsimp
    apply (drule_tac x="i" in spec)
    apply clarsimp
    apply (drule_tac x="cs!i" in bspec)
     apply clarsimp
    apply clarsimp
    apply (drule oghoare_Skip)
    apply clarsimp
    apply (drule_tac x="as!i" in bspec)
     apply (clarsimp simp: in_set_conv_nth)
     apply (rule_tac x=i in exI, fastforce)
     apply clarsimp
     apply blast
```

```
  apply clarsimp
  apply (erule assertionsR.cases; simp)
  apply clarsimp
  apply (rule AsParallelSkips; clarsimp)
 done
next
 case (ParThrow cs s c c' sa a t Q A) thus ?case
  apply clarsimp
  apply (drule oghoare_Parallel)
  apply (clarsimp simp: in_set_conv_nth)
  apply (drule_tac x=i in spec)
  apply clarsimp
  apply (drule oghoare_Throw)
  apply clarsimp
  apply (rename_tac i as Q' A' P')
  apply (rule_tac x="AnnExpr P'" in exI)
  apply (rule conjI)
   apply (rule ThrowRule)
   apply clarsimp
   apply (erule_tac A="(⋃x∈set as. abrcond x)" in subsetD[where B=A],
force)
  apply (rule conjI)
   apply (drule_tac x="as!i" in bspec)
    apply (clarsimp simp: in_set_conv_nth)
    apply (rule_tac x=i in exI, fastforce)
   apply (fastforce)
  apply clarsimp
  apply (erule AsParallelExprs)
  apply clarsimp
  apply (erule assertionsR.cases, simp_all)
  apply (rule AsThrow)
 done
next
 case (Await x b c c' x' c'' c''' x'' a x''' Q A) thus ?case
   apply (clarsimp)
   apply (drule oghoare_Await)
   apply clarsimp

   apply (drule rtranclp_conjD)
   apply clarsimp
   apply (erule disjE)
    apply clarsimp
    apply (rename_tac P' Q' A')
    apply (rule_tac x="AnnExpr Q" in exI)
    apply clarsimp
    apply (rule conjI)
     apply (rule Skip)
    apply (rule conjI)
     apply (drule (1) oghoare_atom_com_sound)
```

```
     apply (fastforce simp: final_def valid_def)
    apply clarsimp
    apply (erule assertionsR.cases, simp_all)
    apply clarsimp
    apply (rule AsAwaitSkip)

   apply (rule_tac x="AnnExpr A" in exI)
   apply clarsimp
   apply (rule conjI)
    apply (rule Throw)
    apply (rule conjI)
     apply (drule (1) oghoare_atom_com_sound)
     apply (fastforce simp: final_def valid_def)
    apply clarsimp
    apply (erule assertionsR.cases, simp_all)
    apply clarsimp
    apply (rule AsAwaitThrow)
  done
qed simp_all

lemma oghoare_steps[rule_format, OF _ refl refl]:
 "Γ ⊢ cf →* cf' ⟹ cf = (c, Normal s) ⟹ cf' = (c', Normal t) ⟹
  Γ,Θ⊢/F a c Q,A ⟹
 s ∈ pre a ⟹
∃a'. Γ,Θ⊢/F a' c' Q,A ∧ t ∈ pre a' ∧
        (∀as. assertionsR Γ Θ Q A a' c' as ⟶ assertionsR Γ Θ Q A a
c as) ∧
        (∀pm cm. atomicsR Γ Θ a' c' (pm, cm) ⟶ atomicsR Γ Θ a c (pm,
cm))"
  apply (induct arbitrary: a c s c' t rule: converse_rtranclp_induct)
   supply [[simproc del: defined_all]]
   apply fastforce
  apply clarsimp
  apply (frule Normal_pre_star)
  apply clarsimp
  apply (drule (2) oghoare_step)
  apply clarsimp
  apply ((drule meta_spec)+, (erule meta_impE, rule conjI, (rule refl)+)+)
  apply (erule (1) meta_impE)+
  apply clarsimp
  apply (rule exI)
  apply (rule conjI, fastforce)+
  apply force
 done

lemma oghoare_sound_Parallel_Normal_case[rule_format, OF _ refl refl]:
 "Γ ⊢ (c, s) →* (c', t) ⟹
   ∀P x y cs. c = Parallel cs  ⟶ s = Normal x ⟶
      t = Normal y ⟶
```

```
      Γ,Θ⊢/F P c Q,A ⟶ final (c', t) ⟶
       x ∈ pre P ⟶ t ∉ Fault ‘ F ⟶ (c' = Throw ∧ t ∈ Normal ‘ A)
∨ (c' = Skip ∧ t ∈ Normal ‘ Q)"
  apply(erule converse_rtranclp_induct2)
   apply (clarsimp simp: final_def)
  apply(erule step.cases, simp_all)
— Parallel
    apply clarsimp
    apply (frule Normal_pre_star)
    apply (drule oghoare_Parallel)
    apply clarsimp
    apply (rename_tac i cs c1' x y  s' as)
    apply (subgoal_tac "Γ⊢ (Parallel cs, Normal x) → (Parallel (cs[i
:= c1']), Normal s')")
     apply (frule_tac c="Parallel cs" and
                      a="AnnPar as"  and
                      Q="(⋂x∈set as. postcond x)" and A ="(⋃x∈set as.
abrcond x)"
                  in oghoare_step[where Θ=Θ and F=F])
       apply(rule Parallel, simp)
          apply clarsimp
          apply (rule Conseq, rule exI[where x="{}"], fastforce)
         apply clarsimp
        apply force
       apply force
      apply clarsimp
     apply clarsimp
     apply (rename_tac a')
     apply (drule_tac x=a' in spec)
     apply (drule mp, rule Conseq)
      apply (rule_tac x="{}" in exI)
      apply (rule_tac x="(⋂x∈set as. postcond x)" in exI)
      apply (rule_tac x="(⋃x∈set as. abrcond x)" in exI)
      apply (simp)
     apply clarsimp
    apply(erule (1) step.Parallel)
— ParSkip
   apply (frule no_steps_final, simp add: final_def)
   apply clarsimp
   apply (drule oghoare_Parallel)
   apply clarsimp
   apply (rule imageI)
   apply (erule subsetD)
   apply clarsimp
   apply (clarsimp simp: in_set_conv_nth)
   apply (rename_tac i)
   apply (frule_tac x="i" in spec)
   apply clarsimp
   apply (frule_tac x="cs!i" in bspec)
```

```
    apply (clarsimp simp: in_set_conv_nth)
    apply (rule_tac x="i" in exI)
    apply clarsimp
   apply clarsimp
   apply (drule_tac x="as ! i" in bspec)
    apply (clarsimp simp: in_set_conv_nth)
    apply fastforce
   apply (drule oghoare_Skip)
   apply fastforce
— ParThrow
  apply clarsimp
  apply (frule no_steps_final, simp add: final_def)
  apply clarsimp
  apply (drule oghoare_Parallel)
  apply (clarsimp simp: in_set_conv_nth)
  apply (drule_tac x=i in spec)
  apply clarsimp
  apply (drule oghoare_Throw)
  apply clarsimp
  apply (rename_tac i as Q' A' P')
  apply (drule_tac x="as ! i" in bspec)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rule_tac x=i in exI, fastforce)
  apply clarsimp
  apply (rule imageI)
  apply (erule_tac  A="(⋃x∈set as. abrcond x)" in subsetD)
  apply clarsimp
  apply (rule_tac x="as!i" in bexI)
   apply blast
  apply clarsimp
  done

lemma oghoare_step_Fault[rule_format, OF _ refl refl]:
 "Γ⊢ cf → cf' ⟹
   cf = (c, Normal x) ⟹
   cf' = (c', Fault f) ⟹
   x ∈ pre P ⟹
   Γ,Θ⊢/F P c Q,A ⟹ f ∈ F"
  supply [[simproc del: defined_all]]
  apply (induct arbitrary: x c c' P Q A f rule: step.induct, simp_all)
       apply clarsimp
       apply (drule oghoare_Guard)
       apply clarsimp
       apply blast
      apply clarsimp
      apply (drule oghoare_Seq)
      apply clarsimp
     apply clarsimp
     apply (drule oghoare_Catch)
```

```
     apply clarsimp
    apply clarsimp
    apply (rename_tac i cs c' x P Q A f)
    apply (drule oghoare_Parallel)
    apply clarsimp
    apply (rename_tac i cs c' x Q A f as)
    apply (drule_tac x="i" in spec)
    apply clarsimp
    apply (drule meta_spec)+
    apply (erule meta_impE, rule conjI, (rule refl)+)+
    apply (drule_tac x="as!i" in bspec)
    apply (clarsimp simp: in_set_conv_nth)
    apply (rule_tac x="i" in exI, fastforce)
   apply (erule (1) meta_impE)
   apply (erule (2) meta_impE)
  apply clarsimp
  apply (drule rtranclp_conjD[THEN conjunct1])
  apply (drule oghoare_Await)
  apply clarsimp
  apply (rename_tac b c c' x Q A f r P' Q' A')
  apply (drule (1) oghoare_atom_com_sound)
  apply (simp add: valid_def)
  apply (drule_tac x="Normal x" in spec)
  apply (drule_tac x="Fault f" in spec)
  apply (drule_tac x=Skip in spec)
  apply clarsimp
  apply (erule impE)
   apply (cut_tac f=f and c=c' in  steps_Fault[where Γ=Γ])
   apply fastforce
   apply (fastforce simp: final_def steps_Fault)
 done

lemma oghoare_step_Stuck[rule_format, OF _ refl refl]:
 "Γ⊢ cf → cf' ⟹
   cf = (c, Normal x) ⟹
   cf' = (c', Stuck) ⟹
   x ∈ pre P ⟹
   Γ,Θ⊢/F P c Q,A ⟹ P'"
  apply (induct arbitrary: x c c' P Q A  rule: step.induct, simp_all)
        apply clarsimp
        apply (drule oghoare_Spec)
        apply force
       apply clarsimp
      apply (drule oghoare_Seq)
      apply clarsimp
     apply clarsimp
     apply (drule oghoare_Call)
     apply clarsimp
    apply clarsimp
```

```
    apply (drule oghoare_Catch)
    apply clarsimp
   apply clarsimp
   apply (drule oghoare_Parallel)
   apply clarsimp
   apply (rename_tac i cs c' x Q A as)
   apply (drule_tac x="i" in spec)
   apply clarsimp
   apply (drule meta_spec)+
   apply (drule_tac x="as!i" in bspec)
   apply (clarsimp simp: in_set_conv_nth)
   apply (rule_tac x="i" in exI, fastforce)
   apply (erule  meta_impE[OF _ refl])
   apply (erule (1) meta_impE)
   apply (erule (2) meta_impE)
  apply clarsimp
  apply (drule rtranclp_conjD[THEN conjunct1])
  apply (drule oghoare_Await)
  apply clarsimp
  apply (rename_tac b c c' x Q A  r P'' Q' A')
 apply (drule (1) oghoare_atom_com_sound)
 apply (simp add: valid_def)
 apply (drule_tac x="Normal x" in spec)
 apply (drule_tac x=Stuck in spec)
 apply (drule_tac x=Skip in spec)
 apply clarsimp
 apply (erule impE)
  apply (cut_tac c=c' in  steps_Stuck[where Γ=Γ])
  apply fastforce
  apply (fastforce simp: final_def steps_Fault)
 apply clarsimp
 apply (drule oghoare_Await)
 apply clarsimp
done

lemma oghoare_steps_Fault[rule_format, OF _ refl refl]:
 "Γ⊢ cf →* cf' ⟹
   cf = (c, Normal x) ⟹
   cf' = (c', Fault f) ⟹
   x ∈ pre P ⟹
   Γ,Θ⊢/F P c Q,A ⟹ f ∈ F"
  apply (induct arbitrary: x c c' f rule: rtranclp_induct)
   supply [[simproc del: defined_all]]
   apply fastforce
  apply clarsimp
  apply (rename_tac b x c c' f)
  apply (case_tac b)
    apply clarsimp
    apply (drule (2) oghoare_steps)
```

```
    apply clarsimp
    apply (drule (3) oghoare_step_Fault)
   apply clarsimp
   apply (drule meta_spec)+
   apply (erule meta_impE, (rule conjI, (rule refl)+))+
    apply simp
   apply (drule step_Fault_prop ; simp)
   apply simp
  apply clarsimp
  apply (drule step_Stuck_prop ; simp)
 done

lemma oghoare_steps_Stuck[rule_format, OF _ refl refl]:
 "Γ⊢ cf →* cf' ⟹
   cf = (c, Normal x) ⟹
   cf' = (c', Stuck) ⟹
   x ∈ pre P ⟹
   Γ,Θ⊢/F P c Q,A ⟹ P'"
  apply (induct arbitrary: x c c' rule: rtranclp_induct)
   apply fastforce
  apply clarsimp
  apply (rename_tac b x c c')
  apply (case_tac b)
    apply clarsimp
    apply (drule (2) oghoare_steps)
    apply clarsimp
    apply (drule (3) oghoare_step_Stuck)
   apply clarsimp
   apply (drule step_Fault_prop ; simp)
   apply simp
 done

lemma oghoare_sound_Parallel_Fault_case[rule_format, OF _ refl refl]:
 "Γ ⊢ (c, s) →* (c', t) ⟹
   ∀P x f cs. c = Parallel cs  ⟶ s = Normal x ⟶
      x ∈ pre P ⟶ t = Fault f ⟶
      Γ,Θ⊢/F P c Q,A ⟶ final (c', t) ⟶
      f ∈ F"
  apply(erule converse_rtranclp_induct2)
   apply (clarsimp simp: final_def)
  apply clarsimp

  apply (rename_tac c s P x f cs)
  apply (case_tac s)
    apply clarsimp

    apply(erule step.cases, simp_all)
         apply (clarsimp simp: final_def)
         apply (drule oghoare_Parallel)
```

```
         apply clarsimp
         apply (rename_tac x f s' i cs c1' as)
         apply (subgoal_tac "Γ⊢ (Parallel cs, Normal x) → (Parallel (cs[i
:= c1']), Normal s')")
         apply (frule_tac c="Parallel cs" and a="AnnPar as"  and
                     Q="(⋂x∈set as. postcond x)" and A="(⋃x∈set as.
abrcond x)"
                     in oghoare_step[where Θ=Θ and F=F])
          apply(rule Parallel)
               apply simp
              apply clarsimp
              apply (rule Conseq, rule exI[where x="{}"], fastforce)
             apply assumption
            apply clarsimp
           apply clarsimp
          apply simp
         apply clarsimp
        apply (rename_tac a')
        apply (drule_tac x=a' in spec)
        apply clarsimp
        apply (erule notE[where P="oghoare _ _ _ _ _ _ _"])
        apply (rule Conseq, rule exI[where x="{}"])
        apply (clarsimp)
        apply (rule_tac x="(⋂x∈set as. postcond x)" in exI)
        apply (rule_tac x="(⋃x∈set as. abrcond x)" in exI ; simp)
       apply(erule (1) step.Parallel)
      apply clarsimp
      apply (fastforce dest: no_steps_final simp: final_def)+
   apply (clarsimp simp: final_def)
   apply (drule oghoare_Parallel)
   apply (erule step_Normal_elim_cases, simp_all)
    apply clarsimp
    apply (rename_tac f cs f' i c1' as)
   apply (drule_tac x="i" in spec)
   apply (erule  impE, fastforce)
   apply clarsimp
   apply (drule_tac x="as!i" in bspec)
    apply (clarsimp simp: in_set_conv_nth)
    apply (rule_tac x="i" in exI, fastforce)
   apply (drule_tac P="pres (as ! i)" in  oghoare_step_Fault[where Θ=Θ
and F=F])
     apply assumption+
   apply (drule steps_Fault_prop ; simp)
   apply simp
  apply (drule steps_Stuck_prop ;simp)
 done

lemma oghoare_soundness:
  "(Γ, Θ ⊢_/F P c Q,A ⟶ Γ ⊨_/F (pre P) c Q, A) ∧
```

```
 (Γ, Θ ⊩/F P' P c Q,A ⟶ Γ ⊨/F P' c Q, A)"
unfolding valid_def
proof (induct rule: oghoare_oghoare_seq.induct)
  case SeqSkip thus ?case
    by (fastforce
          elim: converse_rtranclpE step_Normal_elim_cases(1))
next
  case SeqThrow thus ?case
  by (fastforce
          elim: converse_rtranclpE step_Normal_elim_cases)
next
  case SeqBasic thus ?case
    by (fastforce
          elim: converse_rtranclpE step_Normal_elim_cases
          simp: final_def)
next
  case (SeqSpec Γ Θ F r Q A) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply (erule step_Normal_elim_cases)
   apply (fastforce elim!: converse_rtranclpE step_Normal_elim_cases)
  by clarsimp
next
  case (SeqSeq Γ Θ F P₁ c₁ P₂ A c₂ Q) show ?case
   using SeqSeq
   by (fold valid_def)
      (fastforce intro: oghoare_seq_valid simp: valid_weaken_pre)
next
  case (SeqCatch Γ Θ F P₁ c₁ Q P₂ c₂ A) thus ?case
  by (fold valid_def)
     (fastforce elim:  oghoare_catch_valid)+
next
  case (SeqCond Γ Θ F P b c1 Q A c2) thus ?case
   by (fold valid_def)
      (fastforce intro:oghoare_if_valid)
next
  case (SeqWhile Γ Θ F P c A b) thus ?case
   by (fold valid_def)
      (fastforce elim: valid_weaken_pre[rotated] oghoare_while_valid)
next
  case (SeqGuard Γ Θ F P c Q A r g f) thus ?case
  apply (fold valid_def)
  apply (simp (no_asm) add: valid_def)
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (fastforce simp: final_def)
  apply clarsimp
  apply (erule step_Normal_elim_cases)
```

```
   apply (case_tac "r ∩ - g ≠ {}")
    apply clarsimp
    apply (fastforce simp: valid_def)
   apply clarsimp
   apply (fastforce simp: valid_def)
  apply clarsimp
   apply (case_tac "r ∩ - g ≠ {}")
    apply (fastforce dest: no_steps_final simp:final_def)
   apply (fastforce dest: no_steps_final simp:final_def)
 done
next
  case (SeqCall Γ p f Θ F P Q A) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp add: final_def)
  apply (erule step_Normal_elim_cases)
   apply (clarsimp simp: final_def)
   apply fastforce
  apply fastforce
 done
next
  case (SeqDynCom r P fa Γ Θ F c Q A) thus ?case
  apply -
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply clarsimp
  apply (erule step_Normal_elim_cases)
  apply clarsimp
  apply (rename_tac t c' x)
  apply (drule_tac x=x in bspec, fastforce)
  apply clarsimp
  apply (drule_tac x="Normal x" in spec)
  apply (drule_tac x="t" in spec)
  apply (drule_tac x="c'" in spec)
  apply fastforce+
 done
 next
  case (SeqConseq Γ Θ F P c Q A) thus ?case
   apply (clarsimp)
   apply (rename_tac t c' x)
   apply (erule_tac x="Normal x" in allE)
   apply (erule_tac x="t" in allE)
   apply (erule_tac x="c'" in allE)
   apply (clarsimp simp: pre_weaken_pre)
   apply force
  done
next
  case (SeqParallel as P Γ Θ F cs Q A) thus ?case
```

```
  by (fold valid_def)
     (erule (1) valid_weaken_pre)
next
  case (Call Θ p as n P Q A r Γ f F) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp add: final_def)
  apply (erule step_Normal_elim_cases)
  apply (clarsimp simp: final_def)
   apply (erule disjE)
   apply clarsimp
    apply fastforce
   apply fastforce
  apply fastforce
 done
 next
  case (Await Γ Θ F P c Q A r b) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp add: final_def)
  apply (erule step_Normal_elim_cases)
     apply (erule converse_rtranclpE)
      apply (fastforce simp add: final_def )
     apply (force dest!:no_step_final simp: final_def)
    apply clarsimp
    apply (rename_tac x c'')
    apply (drule_tac x="Normal x" in spec)
    apply (drule_tac x="Stuck" in spec)
    apply (drule_tac x="Skip" in spec)
    apply (clarsimp simp: final_def)
    apply (erule impE[where P="rtranclp _ _ _"])
     apply (cut_tac c="c''" in  steps_Stuck[where Γ=Γ])
     apply fastforce
    apply fastforce
   apply clarsimp
   apply (rename_tac x c'' f)
   apply (drule_tac x="Normal x" in spec)
   apply (drule_tac x="Fault f" in spec)
   apply (drule_tac x="Skip" in spec)
   apply (erule impE[where P="rtranclp _ _ _"])
    apply (cut_tac c="c''" and f=f in  steps_Fault[where Γ=Γ])
    apply fastforce
   apply clarsimp
   apply (erule converse_rtranclpE)
     apply fastforce
   apply (erule step_elim_cases)
  apply (fastforce)
 done
 next
```

```
    case (Parallel as cs Γ Θ F Q A ) thus ?case
    apply (fold valid_def)
    apply (simp only:pre.simps)
    apply (simp (no_asm) only: valid_def)
    apply clarsimp
    apply (rename_tac t c' x')
    apply (case_tac t)
      apply clarsimp
      apply (drule oghoare_sound_Parallel_Normal_case[where Θ=Θ and
Q=Q and A=A and F=F and P="AnnPar as", OF _ refl])
          apply (rule oghoare_oghoare_seq.Parallel)
              apply simp
             apply clarsimp
            apply assumption
           apply (clarsimp)
          apply clarsimp
         apply (clarsimp simp: final_def)
        apply (clarsimp )
       apply clarsimp
      apply clarsimp
     apply (drule oghoare_sound_Parallel_Fault_case[where Θ=Θ and Q=Q
and A=A and F=F and P="AnnPar as", OF _ ])
         apply clarsimp
        apply assumption
        apply (rule oghoare_oghoare_seq.Parallel)
            apply simp
           apply clarsimp
          apply assumption
         apply clarsimp
        apply clarsimp
       apply (simp add: final_def)
      apply (fastforce simp add: final_def)
     apply (clarsimp simp: final_def)
     apply (erule  oghoare_steps_Stuck[where Θ=Θ and F=F and Q=Q and
A=A and P="AnnPar as"])
      apply simp
     apply (rule  oghoare_oghoare_seq.Parallel)
         apply simp
        apply simp
       apply simp
      apply clarsimp
     apply clarsimp
    done
  next
    case Skip thus ?case
      by (fastforce
            elim: converse_rtranclpE step_Normal_elim_cases(1))
  next
    case Basic thus ?case
```

```
    by (fastforce
          elim: converse_rtranclpE step_Normal_elim_cases
          simp: final_def)
next
  case (Spec Γ Θ F r Q A) thus ?case
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply (erule step_Normal_elim_cases)
   apply (fastforce elim!: converse_rtranclpE step_Normal_elim_cases)
  by clarsimp
next
  case (Seq Γ Θ F P₁ c₁ P₂ A c₂ Q) show ?case
   using Seq
   by (fold valid_def)
      (fastforce intro: oghoare_seq_valid simp: valid_weaken_pre)
 next
  case (Cond Γ Θ F P₁ c₁ Q A P₂ c₂ r b) thus ?case
   by (fold valid_def)
      (fastforce intro:oghoare_if_valid)
 next
  case (While Γ Θ F P c i A b Q r) thus ?case
   by (fold valid_def)
      (fastforce elim: valid_weaken_pre[rotated] oghoare_while_valid)
 next
  case Throw thus ?case
  by (fastforce
          elim: converse_rtranclpE step_Normal_elim_cases)
 next
  case (Catch Γ Θ F P₁ c₁ Q P₂ c₂ A) thus ?case
  apply (fold valid_def)
  apply (fastforce elim: oghoare_catch_valid)+
  done
 next
  case (Guard Γ Θ F P c Q A r g f) thus ?case
  apply (fold valid_def)
  apply (simp)
  apply (frule (1) valid_weaken_pre[rotated])
  apply (simp (no_asm) add: valid_def)
  apply clarsimp
  apply (erule converse_rtranclpE)
   apply (fastforce simp: final_def)
  apply clarsimp
  apply (erule step_Normal_elim_cases)
   apply (case_tac "r ∩ - g ≠ {}")
    apply clarsimp
    apply (fastforce simp: valid_def)
   apply clarsimp
   apply (fastforce simp: valid_def)
```

```
    apply clarsimp
     apply (case_tac "r ∩ - g ≠ {}")
      apply clarsimp
      apply (fastforce dest: no_steps_final simp:final_def)
     apply (clarsimp simp: ex_in_conv[symmetric])
    done
   next
    case (DynCom r Γ Θ F P c Q A) thus ?case

    apply clarsimp
    apply (erule converse_rtranclpE)
     apply (clarsimp simp: final_def)
    apply clarsimp
    apply (erule step_Normal_elim_cases)
    apply clarsimp
    apply (rename_tac t c' x)
    apply (erule_tac x=x in ballE)
    apply clarsimp
    apply (drule_tac x="Normal x" in spec)
    apply (drule_tac x="t" in spec)
    apply (drule_tac x="c'" in spec)

    apply fastforce+
   done
   next
    case (Conseq Γ Θ F P c Q A) thus ?case
     apply (clarsimp)
     apply (rename_tac P' Q' A' t c' x)
     apply (erule_tac x="Normal x" in allE)
     apply (erule_tac x="t" in allE)
     apply (erule_tac x="c'" in allE)
     apply (clarsimp simp: pre_weaken_pre)
     apply force
    done
qed

lemmas oghoare_sound = oghoare_soundness[THEN conjunct1, rule_format]
lemmas oghoare_seq_sound = oghoare_soundness[THEN conjunct2, rule_format]

end


theory Cache_Tactics
imports Main
begin

ML ‹
signature CACHE_TACTICS =
sig
```

```
type cache_id = int

val new_subgoal_cache: unit -> cache_id
val SUBGOAL_CACHE: cache_id -> (term * int -> tactic) -> int -> tactic
val clear_subgoal_cache: cache_id -> unit
val cacheify_tactic:
  int -> (Proof.context * (cache_id list) -> int -> tactic) ->
  Proof.context -> int -> tactic

val PARALLEL_GOALS_CACHE: cache_id -> tactic -> tactic

end;

>
```

ML<

```
structure Cache_Tactics: CACHE_TACTICS =
struct
type cache_id = int
(* Stateful cache of intermediate tactic results.

   USE WITH CARE: Each "function" (pair of function and configuration
parameters)
   needs a unique cache.

   Some more experiments with which sub-caches
   get the best speedup?
*)

val caches = Synchronized.var "caches" ((~1,[]) : int * ((int * (thm option
Termtab.table)) list))

fun new_subgoal_cache () = Synchronized.change_result caches
      (fn (maxidx,cache_list) => (
      (maxidx,(maxidx + 1,(maxidx,Termtab.empty) :: cache_list))))

fun SUBGOAL_CACHE id (tac : (term * int) -> tactic) i st =
CSUBGOAL (fn (prem,i) =>
  (fn st =>
   let
      val tab = case (AList.lookup (op =) (Synchronized.value caches |>
snd) id)
      of SOME x => x | NONE => error "Missing cache"

      val pprem = Thm.term_of prem;
      val othm =
        (case (Termtab.lookup tab pprem)
```

```
          of SOME (othm) => othm
            | NONE =>
               let
                val othm = Option.map fst (Seq.pull (tac (pprem,1) (Goal.init
prem)))
                val _ = Synchronized.change caches (fn (maxidx,nets) =>
                        (maxidx,AList.map_entry (op =) id
                            (Termtab.update (pprem,othm)) nets))
               in
                othm
               end)

   in
      case othm of
      SOME thm => Thm.bicompose NONE {flatten = false, match = false,
incremented = false}
        (false, Goal.conclude thm, Thm.nprems_of thm) i st
     | NONE => Seq.empty
   end)


  ) i st


fun clear_subgoal_cache id = Synchronized.change caches
  (fn (maxidx,cache_list) => (maxidx,AList.delete (op =) id cache_list))

local

val pcaches = Synchronized.var "pcaches" ([] : ((int * (term Termtab.table))
list))

val (_,cache_oracle) = Context.>>> (Context.map_theory_result (Thm.add_oracle
(@{binding "cache"},I)))

exception FAILED of unit;

fun retrofit st' =
  rotate_prems ~1 #>
  Thm.bicompose NONE {flatten = false, match = false, incremented = false}
      (false, Goal.conclude st', Thm.nprems_of st') 1;

in

fun PARALLEL_GOALS_CACHE cache_id tac = if cache_id < 0 then
  (Synchronized.change pcaches (AList.map_default (op =) (~cache_id,Termtab.empty)
(K Termtab.empty));PARALLEL_GOALS tac) else
  Thm.adjust_maxidx_thm ~1 #>
  (fn st =>
```

```
    if not (Future.enabled ()) orelse Thm.maxidx_of st >= 0 orelse Thm.nprems_of
st <= 1
    then DETERM tac st
    else
      let
         val tab = Synchronized.change_result pcaches (fn caches =>
              case (AList.lookup (op =) caches cache_id) of SOME net =>
(net,caches)
                | NONE => (Termtab.empty,(AList.update (op =) (cache_id,Termtab.empty)
caches)))

        fun do_cache p result =
        let
          val cresult = Thm.global_cterm_of (Thm.theory_of_cterm p) result
        in
          try cache_oracle cresult
        end

        fun try_cache (i,p) (cached,uncached)  = case (Termtab.lookup
tab (Thm.term_of p))
        of SOME result => (case do_cache p result of SOME thm => ((i,thm)
:: cached,uncached)
                                                   | NONE => (cached,
(i,p) :: uncached))
        | NONE => (cached,(i,p) :: uncached)

        fun try_tac (i,p) =
          (case SINGLE tac (Goal.init p) of
            NONE => raise FAILED ()
          | SOME g' => (i,g'))

        val goals = Drule.strip_imp_prems (Thm.cprop_of st);
        val (cached,uncached) = fold try_cache (goals |> tag_list 0) ([],[])

        val results = Par_List.map (`try_tac) uncached;

        val _ = Synchronized.change pcaches (
          AList.map_default (op =) (cache_id,Termtab.empty)
            (fold (fn ((_,result),(_,goal))  =>
               Termtab.update (Thm.term_of goal,Thm.prop_of result)
             ) results))

        val _ = if null cached then ()
          else (warning
          ("WARNING: Used cached result for PARALLEL_GOALS_CACHE.\n" ^
          "Give ~x to clear cache x and ensure correctness."))

        val full_results = (map fst results) @ cached |> order_list;
```

```
      in EVERY (map retrofit (rev full_results)) st end
      handle FAILED () => Seq.empty);

fun cacheify_tactic n tac ctxt i st =
  let
    val caches = List.tabulate (n, fn _ => new_subgoal_cache ());
    fun clear_caches _ = app clear_subgoal_cache caches ;
  in
      st |>
      (tac (ctxt,caches) i
      THEN (PRIMITIVE Drule.implies_intr_hyps)
      THEN (PRIMITIVE (fn thm => (clear_caches ();thm))))

  end


end;

end;
›
```

end

6 Verfication Condition Generator for COMPLX OG

theory OG_Tactics
imports
 OG_Soundness
 "lib/Cache_Tactics"
begin

6.1 Seq oghoare derivation

lemmas SeqSkipRule = SeqSkip
lemmas SeqThrowRule = SeqThrow
lemmas SeqBasicRule = SeqBasic
lemmas SeqSpecRule = SeqSpec
lemmas SeqSeqRule = SeqSeq

lemma SeqCondRule:
 "⟦ Γ, Θ ⊨$_{/F}$ C1 P$_1$ c$_1$ Q,A;
    Γ, Θ ⊨$_{/F}$ C2 P$_2$ c$_2$ Q,A ⟧
   ⟹ Γ, Θ ⊨$_{/F}$ {s. (s∈b ⟶ s∈C1) ∧ (s∉b ⟶ s∈C2)} (AnnBin r P$_1$ P$_2$)
                  (Cond b c$_1$ c$_2$) Q,A"
  apply (rule SeqCond)
   apply (erule SeqConseq[rotated]; clarsimp)
  apply (erule SeqConseq[rotated]; clarsimp)

done

lemma SeqWhileRule:
"⟦ Γ, Θ ⊨$_{/F}$ (i ∩ b) a c i,A; i ∩ - b ⊆ Q ⟧
⟹ Γ, Θ ⊨$_{/F}$ i (AnnWhile r i a) (While b c) Q,A"
apply (rule SeqConseq[OF _ SeqWhile])
prefer 2 **apply** assumption
by simp+

lemma DynComRule:
"⟦ r ⊆ pre a; ⋀s. s∈r ⟹ Γ, Θ ⊢$_{/F}$ a (c s) Q,A ⟧ ⟹
Γ, Θ ⊢$_{/F}$ (AnnRec r a) (DynCom c) Q,A"
by (erule DynCom) clarsimp

lemma SeqDynComRule:
"⟦r ⊆ pre a;
⋀s. s∈r ⟹ Γ, Θ ⊨$_{/F}$P a (c s) Q,A;
P⊆r ⟧ ⟹
Γ, Θ ⊨$_{/F}$ P (AnnRec r a) (DynCom c) Q,A"
by (erule SeqDynCom) clarsimp

lemma SeqCallRule:
"⟦ P' ⊆ P; Γ, Θ ⊨$_{/F}$ P P'' f Q,A;
n < length as; Γ p = Some f;
as ! n = P''; Θ p = Some as⟧
⟹ Γ, Θ ⊨$_{/F}$ P' (AnnCall r n) (Call p) Q,A"
by (simp add: SeqCall SeqConseq)

lemma SeqGuardRule:
"⟦ P ∩ g ⊆ P'; P ∩ -g ≠ {} ⟹ f ∈ F;
Γ, Θ ⊨$_{/F}$ P' a c Q,A ⟧ ⟹
Γ, Θ ⊨$_{/F}$ P (AnnRec r a) (Guard f g c) Q,A"
by (simp add: SeqGuard SeqConseq)

6.2 Parallel-mode rules

lemma GuardRule:
"⟦ r ∩ g ⊆ pre P; r ∩ -g ≠ {} ⟶ f ∈ F;
Γ, Θ ⊢$_{/F}$ P c Q,A ⟧ ⟹
Γ, Θ ⊢$_{/F}$ (AnnRec r P) (Guard f g c) Q,A"
by (simp add: Guard)

lemma CallRule:
"⟦ r ⊆ pre P; Γ, Θ ⊢$_{/F}$ P f Q,A;
n < length as; Γ p = Some f;
as ! n = P; Θ p = Some as⟧
⟹ Γ, Θ ⊢$_{/F}$ (AnnCall r n) (Call p) Q,A"
by (simp add: Call)

```
definition map_ann_hoare :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒
'f set
             ⇒ ('s, 'p, 'f) ann_triple list ⇒ ('s,'p,'f) com list ⇒
bool"
    (‹(4_,_/[⊩]⇘_⇙ (_/ (_)))› [60,60,60,1000,20]60) where
  "Γ, Θ [⊩]⇘F⇙ Ps Ts ≡ ∀i < length Ts. Γ, Θ ⊢⇘F⇙ (pres (Ps!i)) (Ts!i)
(postcond (Ps!i)), (abrcond (Ps!i))"

lemma MapAnnEmpty: "Γ, Θ [⊩]⇘F⇙ [] []"
 by(simp add:map_ann_hoare_def)

lemma MapAnnList: "⟦ Γ, Θ ⊢⇘F⇙ P c Q, A;
                     Γ, Θ  [⊩]⇘F⇙ Ps Ts ⟧
                ⟹ Γ, Θ [⊩]⇘F⇙ ((P, Q, A)#Ps) (c#Ts)"
  apply(simp add:map_ann_hoare_def)
  apply clarify
  apply (rename_tac i)
  apply(case_tac i,simp+)
 done

lemma MapAnnMap:
   "∀k. i≤k ∧ k<j ⟶ Γ, Θ ⊢⇘F⇙ (P k) (c k) (Q k), (A k)
⟹ Γ, Θ [⊩]⇘F⇙ (map (λk. (P k, Q k, A k)) [i..<j]) (map c [i..<j])"
 by (simp add: add.commute le_add1 map_ann_hoare_def)

lemma ParallelRule:
  "⟦Γ, Θ [⊩]⇘F⇙ Ps Cs;
    interfree Γ Θ F Ps Cs;
    length Cs = length Ps
 ⟧ ⟹ Γ, Θ ⊢⇘F⇙ (AnnPar Ps)
                   (Parallel Cs)
                   (⋂i∈{i. i<length Ps}. postcond (Ps!i)), (⋃i∈{i.
i<length Ps}. abrcond (Ps!i))"
  apply (clarsimp simp add:neq_Nil_conv Parallel map_ann_hoare_def )+
  apply (rule Parallel)
      apply fastforce
     apply fastforce
    apply fastforce
   apply clarsimp
  apply (clarsimp simp: in_set_conv_nth)
  apply (rename_tac i)
  apply (rule_tac x=i in exI, fastforce)
 done

lemma ParallelConseqRule:
  "⟦ Γ, Θ ⊢⇘F⇙ (AnnPar Ps)
                   (Parallel Ts)
                   (⋂i∈{i. i<length Ps}. postcond (Ps!i)), (⋃i∈{i.
i<length Ps}. abrcond (Ps!i));
```

```
    (⋂i∈{i. i<length Ps}. postcond (Ps!i)) ⊆ Q;
    (⋃i∈{i. i<length Ps}. abrcond (Ps!i)) ⊆ A
⟧ ⟹ Γ, Θ ⊢/F (AnnPar Ps) (Parallel Ts) Q, A"
 apply (rule Conseq)
 apply (rule exI[where x="(⋂i∈{i. i<length Ps}. precond (Ps!i))"])
 apply (rule exI[where x="(⋂i∈{i. i<length Ps}. postcond (Ps!i))"])
 apply (rule exI[where x="(⋃i∈{i. i<length Ps}. abrcond (Ps!i))"])
 apply (clarsimp)
done
```

See Soundness.thy for the rest of Parallel-mode rules

6.3 VCG tactic helper definitions and lemmas

```
definition interfree_aux_right :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns
⇒ 'f set ⇒ ('s assn × ('s,'p,'f) com × ('s, 'p, 'f) ann) ⇒ bool" where
  "interfree_aux_right Γ Θ F ≡ λ(q, cmd, ann). (∀aa ac.  atomicsR Γ
Θ ann cmd (aa,ac) ⟶ (Γ ⊨/F (q ∩ aa) ac q, q))"

lemma pre_strengthen: "¬pre_par a ⟹ pre (strengthen_pre a a') = pre
a ∩ a'"
  by (induct a arbitrary: a', simp_all)

lemma Basic_inter_right:
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) (Basic f) q, q ⟹ interfree_aux_right
Γ Θ F (q, Basic f, AnnExpr r)"
  by (auto simp: interfree_aux_right_def
           elim!: atomicsR.cases
           dest: oghoare_sound)

lemma Skip_inter_right:
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) Skip q, q ⟹ interfree_aux_right Γ Θ
F (q, Skip, AnnExpr r)"
  by (auto simp: interfree_aux_right_def
           elim!: atomicsR.cases
           dest: oghoare_sound)

lemma Throw_inter_right:
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) Throw q, q ⟹ interfree_aux_right Γ Θ
F (q, Throw, AnnExpr r)"
  by (auto simp: interfree_aux_right_def
           elim!: atomicsR.cases
           dest: oghoare_sound)

lemma Spec_inter_right:
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) (Spec rel) q, q ⟹ interfree_aux_right
Γ Θ F (q, Spec rel, AnnExpr r)"
  by (auto simp: interfree_aux_right_def
           elim!: atomicsR.cases
```

```
          dest: oghoare_sound)

lemma valid_Await:
 "atom_com c ⟹ Γ⊨_/F (P ∩ b) c Q,A ⟹ Γ⊨_/F P Await b c Q,A"
  apply (clarsimp simp: valid_def)
  apply (erule converse_rtranclpE)
   apply (clarsimp simp: final_def)
  apply clarsimp
  apply (erule step_Normal_elim_cases)
      apply clarsimp
      apply (erule disjE)
       apply (fastforce dest: no_steps_final simp: final_def)
      apply (fastforce dest: no_steps_final simp: final_def)
     apply clarsimp
     apply (drule no_steps_final, simp add: final_def)
     apply clarsimp
     apply (rename_tac x c'')
     apply (drule_tac x="Normal x" in spec)
     apply (drule spec[where x=Stuck])
     apply (drule spec[where x=Skip])
     apply (erule impE)
     apply (cut_tac c=c'' in steps_Stuck[where Γ=Γ])
     apply fastforce
    apply clarsimp
   apply (drule no_steps_final, simp add: final_def)
   apply clarsimp
   apply (drule_tac x="Normal x" in spec)
   apply (drule_tac x="Fault f" in spec)
   apply (drule spec[where x=Skip])
   apply (erule impE)
    apply (rename_tac c'' f)
    apply (cut_tac c=c'' and f=f in steps_Fault[where Γ=Γ])
    apply fastforce
   apply clarsimp
  apply clarsimp
 done

lemma atomcom_imp_not_prepare:
 "ann_matches Γ Θ a c ⟹ atom_com c ⟹
   ¬ pre_par a"
 by (induct rule:ann_matches.induct, simp_all)

lemma Await_inter_right:
  "atom_com c ⟹
  Γ, Θ ⊩_/F P a c q,q ⟹
  q ∩ r ∩ b ⊆ P ⟹
  interfree_aux_right Γ Θ F (q, Await b c, AnnRec r a)"
  apply (simp add: interfree_aux_right_def  )
  apply clarsimp
```

```
  apply (erule atomicsR.cases, simp_all)
  apply clarsimp
  apply (rule valid_Await, assumption)
  apply (drule oghoare_seq_sound)
  apply (erule valid_weaken_pre)
  apply blast
 done

lemma Call_inter_right:
  "⟦interfree_aux_right Γ Θ F (q, f, P);
     n < length as; Γ p = Some f;
     as ! n = P; Θ p = Some as⟧ ⟹
  interfree_aux_right Γ Θ F (q, Call p, AnnCall r n)"
  by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma DynCom_inter_right:
  "⟦⋀s. s ∈ r ⟹ interfree_aux_right Γ Θ F (q, f s, P) ⟧ ⟹
  interfree_aux_right Γ Θ F (q, DynCom f, AnnRec r P)"
  by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma Guard_inter_right:
  "interfree_aux_right Γ Θ F (q, c, a)
     ⟹ interfree_aux_right Γ Θ F (q, Guard f g c, AnnRec r a)"
 by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma Parallel_inter_right_empty:
  "interfree_aux_right Γ Θ F (q, Parallel [], AnnPar [])"
 by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma Parallel_inter_right_List:
  "⟦interfree_aux_right Γ Θ F (q, c, a);
    interfree_aux_right Γ Θ F (q, Parallel cs, AnnPar as)⟧
     ⟹ interfree_aux_right Γ Θ F (q, Parallel (c#cs), AnnPar ((a, Q,
A) #as))"
  apply (clarsimp simp: interfree_aux_right_def)
  apply (erule atomicsR.cases; clarsimp)
  apply (rename_tac i aa b)
  apply (case_tac i, simp)
  apply (fastforce intro: AtParallelExprs)
  done

lemma Parallel_inter_right_Map:
  "∀k. i≤k ∧ k<j ⟶ interfree_aux_right Γ Θ F (q, c k, a k)
     ⟹ interfree_aux_right Γ Θ F
                (q, Parallel (map c [i..<j]), AnnPar (map (λi. (a i, Q,
A)) [i..<j]))"
  apply (clarsimp simp: interfree_aux_right_def)
  apply (erule atomicsR.cases; clarsimp)
  apply (rename_tac ia aa b)
```

```
  apply (drule_tac x="i+ia" in spec)
  by fastforce

lemma Seq_inter_right:
  "⟦ interfree_aux_right Γ Θ F (q, c₁, a1); interfree_aux_right Γ Θ F
(q, c₂, a2) ⟧⟹
  interfree_aux_right Γ Θ F (q, Seq c₁ c₂, AnnComp a1 a2)"
 by (auto simp add: interfree_aux_right_def elim: atomicsR.cases)

lemma Catch_inter_right:
  "⟦ interfree_aux_right Γ Θ F (q, c₁, a1); interfree_aux_right Γ Θ F
(q, c₂, a2) ⟧⟹
  interfree_aux_right Γ Θ F (q, Catch c₁ c₂, AnnComp a1 a2)"
 by (auto simp add: interfree_aux_right_def elim: atomicsR.cases)

lemma While_inter_aux_any: "interfree_aux Γ Θ F (Any, (AnyAnn, q, abr),
c, P) ⟹
  interfree_aux Γ Θ F (Any, (AnyAnn, q, abr), While b c, AnnWhile R I
P)"
 by (auto simp add:  interfree_aux_def
          elim: atomicsR.cases[where ?a1.0="AnnWhile _ _ _"] )

lemma While_inter_right:
  "interfree_aux_right Γ Θ F (q, c, a)
     ⟹ interfree_aux_right Γ Θ F (q, While b c, AnnWhile r i a)"
 by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma Cond_inter_aux_any:
  "⟦ interfree_aux Γ Θ F (Any, (AnyAnn, q, a), c₁, a1); interfree_aux
Γ Θ F (Any, (AnyAnn, q, a), c₂, a2) ⟧⟹
   interfree_aux Γ Θ F (Any, (AnyAnn, q, a), Cond b c₁ c₂, AnnBin r a1
a2)"
 by (fastforce simp add:  interfree_aux_def
               elim: atomicsR.cases[where ?a1.0="AnnBin _ _ _"])

lemma Cond_inter_right:
  "⟦ interfree_aux_right Γ Θ F (q, c₁, a1); interfree_aux_right Γ Θ F
(q, c₂, a2) ⟧⟹
   interfree_aux_right Γ Θ F (q, Cond b c₁ c₂, AnnBin r a1 a2)"
 by(auto simp: interfree_aux_right_def elim: atomicsR.cases)

lemma Basic_inter_aux:
  "⟦interfree_aux_right Γ Θ F (r, com, ann);
    interfree_aux_right Γ Θ F (q, com, ann);
    interfree_aux_right Γ Θ F (a, com, ann) ⟧ ⟹
    interfree_aux Γ Θ F (Basic f, (AnnExpr r, q, a), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Skip_inter_aux:
```

```
  "⟦interfree_aux_right Γ Θ F (r, com, ann);
    interfree_aux_right Γ Θ F (q, com, ann);
    interfree_aux_right Γ Θ F (a, com, ann) ⟧ ⟹
    interfree_aux Γ Θ F (Skip, (AnnExpr r, q, a), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Throw_inter_aux:
  "⟦interfree_aux_right Γ Θ F (r, com, ann);
    interfree_aux_right Γ Θ F (q, com, ann);
    interfree_aux_right Γ Θ F (a, com, ann) ⟧ ⟹
    interfree_aux Γ Θ F (Throw, (AnnExpr r, q, a), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Spec_inter_aux:
  "⟦interfree_aux_right Γ Θ F (r, com, ann);
    interfree_aux_right Γ Θ F (q, com, ann);
    interfree_aux_right Γ Θ F (a, com, ann) ⟧ ⟹
    interfree_aux Γ Θ F (Spec rel, (AnnExpr r, q, a), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)



lemma Seq_inter_aux:
  "⟦ interfree_aux Γ Θ F (c₁, (r₁, pre r₂, A), com, ann);
     interfree_aux Γ Θ F (c₂, (r₂, Q, A), com, ann) ⟧
   ⟹ interfree_aux Γ Θ F (Seq c₁ c₂, (AnnComp r₁ r₂, Q, A), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Catch_inter_aux:
  "⟦ interfree_aux Γ Θ F (c₁, (r₁, Q, pre r₂), com, ann);
     interfree_aux Γ Θ F (c₂, (r₂, Q, A), com, ann) ⟧
   ⟹ interfree_aux Γ Θ F (Catch c₁ c₂, (AnnComp r₁ r₂, Q, A), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Cond_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
    interfree_aux Γ Θ F (c₁, (r₁, Q, A), com, ann);
    interfree_aux Γ Θ F (c₂, (r₂, Q, A), com, ann) ⟧
 ⟹  interfree_aux Γ Θ F (Cond b c₁ c₂, (AnnBin r r₁ r₂, Q, A), com,
ann)"
by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)


lemma While_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
     interfree_aux_right Γ Θ F (Q, com, ann);
     interfree_aux Γ Θ F (c, (P, i, A), com, ann) ⟧ ⟹
     interfree_aux Γ Θ F (While b c, (AnnWhile r i P, Q, A), com, ann)"
by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)
```

```
lemma Await_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
     interfree_aux_right Γ Θ F (Q, com, ann);
     interfree_aux_right Γ Θ F (A, com, ann) ⟧
  ⟹ interfree_aux Γ Θ F (Await b e, (AnnRec r ae, Q, A), com, ann)"
  by (auto simp: interfree_aux_def interfree_aux_right_def elim: assertionsR.cases)

lemma Call_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
     interfree_aux Γ Θ F (f, (P, Q, A), com, ann);
     n < length as; Γ p = Some f;
     as ! n = P; Θ p = Some as ⟧ ⟹
     interfree_aux Γ Θ F (Call p, (AnnCall r n, Q, A), com, ann)"
by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma DynCom_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
     interfree_aux_right Γ Θ F (Q, com, ann);
     interfree_aux_right Γ Θ F (A, com, ann);
     ⋀s. s∈r ⟹ interfree_aux Γ Θ F (f s, (P, Q, A), com, ann) ⟧ ⟹
     interfree_aux Γ Θ F (DynCom f, (AnnRec r P, Q, A), com, ann)"
 by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

lemma Guard_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (r, com, ann);
     interfree_aux_right Γ Θ F (Q, com, ann);
     interfree_aux Γ Θ F (c, (P, Q, A), com, ann) ⟧ ⟹
     interfree_aux Γ Θ F (Guard f g c, (AnnRec r P, Q, A), com, ann)"
by (auto elim: assertionsR.cases simp: interfree_aux_def interfree_aux_right_def)

definition
  inter_aux_Par :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
⇒
                    (('s, 'p, 'f) com list × (('s, 'p, 'f) ann_triple)
list × ('s, 'p, 'f) com × ('s, 'p, 'f) ann) ⇒ bool" where
  "inter_aux_Par Γ Θ F ≡
     λ(cs, as, c, a). ∀i < length cs. interfree_aux Γ Θ F (cs ! i, as
! i, c, a)"

lemma inter_aux_Par_Empty: "inter_aux_Par Γ Θ F ([], [], c, a)"
by(simp add:inter_aux_Par_def)

lemma inter_aux_Par_List:
  "⟦ interfree_aux Γ Θ F (x, a, y, a');
  inter_aux_Par Γ Θ F (xs, as, y, a')⟧
  ⟹ inter_aux_Par Γ Θ F (x#xs, a#as, y, a')"
  apply (simp add: inter_aux_Par_def)
  apply (rule allI)
```

```
  apply (rename_tac v)
  apply (case_tac v)
    apply simp_all
  done

lemma inter_aux_Par_Map: "∀k. i≤k ∧ k<j ⟶ interfree_aux Γ Θ F (c
k, Q k, x, a)
 ⟹ inter_aux_Par Γ Θ F (map c [i..<j], map Q [i..<j], x, a)"
  by(force simp add: inter_aux_Par_def less_diff_conv)

lemma Parallel_inter_aux:
  "⟦ interfree_aux_right Γ Θ F (Q, com, ann);
     interfree_aux_right Γ Θ F (A, com, ann);
     interfree_aux_right Γ Θ F (⋂ (set (map postcond as)), com, ann);
     inter_aux_Par Γ Θ F (cs, as, com, ann) ⟧ ⟹
     interfree_aux Γ Θ F (Parallel cs, (AnnPar as, Q, A), com, ann)"
  apply (clarsimp simp: interfree_aux_def interfree_aux_right_def inter_aux_Par_def)
  by (erule assertionsR.cases; fastforce)

definition interfree_swap :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns
⇒ 'f set ⇒ (('s, 'p, 'f) com × (('s, 'p, 'f) ann × 's assn × 's assn)
× ('s, 'p, 'f) com list × (('s, 'p, 'f) ann × 's assn × 's assn) list)
⇒ bool" where
  "interfree_swap Γ Θ F ≡ λ(x, a, xs, as). ∀y < length xs. interfree_aux
Γ Θ F (x, a, xs ! y, pres (as ! y))
  ∧ interfree_aux Γ Θ F (xs ! y, as ! y, x, fst a)"

lemma interfree_swap_Empty: "interfree_swap  Γ Θ F (x, a, [], [])"
by(simp add:interfree_swap_def)

lemma interfree_swap_List:
  "⟦ interfree_aux Γ Θ F (x, a, y, fst (a'));
  interfree_aux Γ Θ F (y, a', x, fst a);
  interfree_swap Γ Θ F (x, a, xs, as)⟧
  ⟹ interfree_swap Γ Θ F (x, a, y#xs, a'#as)"
  apply (simp add: interfree_swap_def)
  apply (rule allI)
  apply (rename_tac v)
  apply (case_tac v)
    apply simp_all
  done

lemma interfree_swap_Map: "∀k. i≤k ∧ k<j ⟶ interfree_aux Γ Θ F (x,
a, c k, fst (Q k))
 ∧ interfree_aux Γ Θ F (c k, (Q k), x, fst a)
 ⟹ interfree_swap Γ Θ F (x, a, map c [i..<j], map Q [i..<j])"
by(force simp add: interfree_swap_def less_diff_conv)

lemma interfree_Empty: "interfree Γ Θ F [] []"
```

```
by(simp add:interfree_def)

lemma interfree_List:
  "⟦ interfree_swap Γ Θ F (x, a, xs, as); interfree Γ Θ F as xs ⟧ ⟹
interfree Γ Θ F (a#as) (x#xs)"
  apply (simp add: interfree_swap_def interfree_def)
  apply clarify
  apply (rename_tac i j)
  apply (case_tac i)
   apply (case_tac j)
    apply simp_all
  apply (case_tac j)
   apply simp_all
 done

lemma interfree_Map:
  "(∀i j. a≤i ∧ i<b ∧ a≤j ∧ j<b  ∧ i≠j ⟶ interfree_aux Γ Θ F (c
i, A i, c j, pres (A j)))
  ⟹ interfree Γ Θ F (map (λk. A k) [a..<b]) (map (λk. c k) [a..<b])"
by (force simp add: interfree_def less_diff_conv)

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" strengthen_pre.simps
lemmas ParallelConseq_list = INTER_eq Collect_conj_eq length_map length_upt
length_append
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


ML ‹val hyp_tac = CSUBGOAL (fn (prem,i) => PRIMITIVE (fn thm =>
    let
      val thm' = Thm.permute_prems 0 (i-1) thm |> Goal.protect 1
      val asm = Thm.assume prem
    in
      case (try (fn thm' => (Thm.implies_elim thm' asm)) thm') of
      SOME thm => thm |> Goal.conclude |> Thm.permute_prems 0 (~(i-1))
     | NONE => error ("hyp_tac failed:" ^ (@{make_string} (thm',asm)))
    end
  ))
›
ML ‹
```

```
(*Remove a premise of the form 's ∈ _' if s is not referred to anywhere*)
fun remove_single_Bound_mem ctxt = SUBGOAL (fn (t, i) => let
    val prems = Logic.strip_assums_hyp t
    val concl = Logic.strip_assums_concl t
    fun bd_member t = (case HOLogic.dest_Trueprop t
        of Const (@{const_name "Set.member"}, _) $ Bound j $ _ => SOME
j
        | _ => NONE) handle TERM _ => NONE
  in filter_prems_tac ctxt
    (fn prem => case bd_member prem of NONE => true
      | SOME j => let val xs = (filter (fn t => loose_bvar1 (t, j)) (concl
:: prems))
        in length xs <> 1 end)
    i
  end handle Subscript => no_tac)
›
```

named_theorems proc_simp
named_theorems oghoare_simps

lemmas guards.simps[oghoare_simps add]
ann_guards.simps[oghoare_simps add]

ML ‹

```
fun rt ctxt t i =
  resolve_tac ctxt [t] i

fun rts ctxt xs i =
  resolve_tac ctxt xs i

fun conjI_Tac ctxt tac i st = st |>
       ( (EVERY [rt ctxt conjI i,
          conjI_Tac ctxt tac (i+1),
          tac i]) ORELSE (tac i) )

fun get_oghoare_simps ctxt =
 Proof_Context.get_thms ctxt "oghoare_simps"

fun simp ctxt extra =
  simp_tac (put_simpset HOL_basic_ss ctxt addsimps extra)

fun simp_only ctxt simps =
  simp_tac ((clear_simpset ctxt) addsimps simps)

fun prod_sel_simp ctxt =
  simp_only ctxt @{thms prod.sel}

fun oghoare_simp ctxt =
```

```
   simp_only ctxt (get_oghoare_simps ctxt)

fun ParallelConseq ctxt =
  clarsimp_tac (put_simpset HOL_basic_ss ctxt addsimps (@{thms ParallelConseq_list}
@ @{thms my_simp_list}))

val enable_trace = false;
fun trace str = if enable_trace then tracing str else ();

fun HoareRuleTac (ctxt' as (ctxt,args)) i st =
  (Cache_Tactics.SUBGOAL_CACHE (nth args 0)
  (fn (_,i) => (SUBGOAL (fn (_,i) =>
    (EVERY[rts ctxt @{thms Seq Catch SeqSeq SeqCatch} i,
        HoareRuleTac ctxt' (i+1),
        HoareRuleTac ctxt' i]
    ORELSE
    (FIRST[rts ctxt (@{thms SkipRule SeqSkipRule}) i,
         rts ctxt (@{thms BasicRule SeqBasicRule}) i,
         rts ctxt (@{thms ThrowRule SeqThrowRule}) i,
         rts ctxt (@{thms SpecRule SeqSpecRule}) i,
         EVERY[rt ctxt (@{thm SeqParallel}) i,
               HoareRuleTac ctxt' (i+1)],
         EVERY[rt ctxt  (@{thm ParallelConseqRule}) i,
               ParallelConseq ctxt (i+2),
               ParallelConseq ctxt (i+1),
               ParallelTac ctxt' i],
         EVERY[rt ctxt (@{thm CondRule}) i,
               HoareRuleTac ctxt' (i+2),
               HoareRuleTac ctxt' (i+1)],
         EVERY[rt ctxt @{thm SeqCondRule} i,
               HoareRuleTac ctxt' (i+1),
               HoareRuleTac ctxt' i],
         EVERY[rt ctxt  (@{thm WhileRule}) i,
               HoareRuleTac ctxt' (i+3)],
         EVERY[rt ctxt  (@{thm SeqWhileRule}) i,
               HoareRuleTac ctxt' i],
         EVERY[rt ctxt (@{thm AwaitRule}) i,
               HoareRuleTac ctxt' (i+1),
               simp ctxt (@{thms atom_com.simps}) i],
         EVERY[rts ctxt (@{thms CallRule SeqCallRule}) i,
               Call_asm_inst ctxt (i+2),
               HoareRuleTac ctxt' (i+1)],
         EVERY[rts ctxt (@{thms DynComRule SeqDynComRule}) i,
               HoareRuleTac ctxt' (i+1)],
         EVERY[rts ctxt (@{thms GuardRule}) i,
               HoareRuleTac ctxt' (i+2)],
         K all_tac i ])))
         THEN_ALL_NEW hyp_tac) i)) i st
```

```
and Call_asm_inst ctxt i =
  let val proc_simps = Proof_Context.get_thms ctxt "proc_simp" @ @{thms
list_lemmas} in
    EVERY[rts ctxt proc_simps (i+3),
         rts ctxt proc_simps (i+2),
         rts ctxt proc_simps (i+1),
        ParallelConseq ctxt i]
    end

and ParallelTac (ctxt' as (ctxt,args)) i =
          EVERY[rt ctxt (@{thm ParallelRule}) i,
                               ParallelConseq ctxt (i+2),
                               interfree_Tac ctxt' (i+1),
                               ParallelConseq ctxt i,
                               MapAnn_Tac ctxt' i]

and MapAnn_Tac (ctxt' as (ctxt,args)) i st = st |>
    (FIRST[rt ctxt (@{thm MapAnnEmpty}) i,
           EVERY[rt ctxt (@{thm MapAnnList}) i,
                 MapAnn_Tac ctxt' (i+1),
                 HoareRuleTac ctxt' i],
            EVERY[rt ctxt (@{thm MapAnnMap}) i,
                 rt ctxt (@{thm allI}) i, rt ctxt (@{thm impI}) i,
                 HoareRuleTac ctxt' i]])

and interfree_Tac (ctxt' as (ctxt,args)) i st = st |>
   (FIRST[rt ctxt (@{thm interfree_Empty}) i,
          EVERY[rt ctxt (@{thm interfree_List}) i,
                interfree_Tac ctxt' (i+1),
                interfree_swap_Tac ctxt' i],
          EVERY[rt ctxt (@{thm interfree_Map}) i,
                rt ctxt (@{thm allI}) i, rt ctxt (@{thm allI}) i, rt ctxt
(@{thm impI}) i,
                interfree_aux_Tac ctxt' i ]])

and interfree_swap_Tac (ctxt' as (ctxt,args)) i st = st |>
    (FIRST[rt ctxt (@{thm interfree_swap_Empty}) i,
           EVERY[rt ctxt (@{thm interfree_swap_List}) i,
                 interfree_swap_Tac ctxt' (i+2),
                 interfree_aux_Tac ctxt' (i+1),
                 interfree_aux_Tac ctxt' i ],
           EVERY[rt ctxt (@{thm interfree_swap_Map}) i,
                 rt ctxt (@{thm allI}) i, rt ctxt (@{thm impI}) i,
                 conjI_Tac ctxt (interfree_aux_Tac ctxt') i]])

and inter_aux_Par_Tac (ctxt' as (ctxt,args)) i st = st |>
    (FIRST[rt ctxt (@{thm inter_aux_Par_Empty}) i,
           EVERY[rt ctxt (@{thm inter_aux_Par_List}) i,
                 inter_aux_Par_Tac ctxt' (i+1),
```

```
              interfree_aux_Tac ctxt' i ],
          EVERY[rt ctxt (@{thm inter_aux_Par_Map}) i,
              rt ctxt (@{thm allI}) i, rt ctxt (@{thm impI}) i,
              interfree_aux_Tac ctxt' i]])

and interfree_aux_Tac ctxt' i = dest_inter_aux_Tac ctxt' i

and dest_inter_aux_Tac (ctxt' as (ctxt,args)) i st =
  (Cache_Tactics.SUBGOAL_CACHE (nth args 1)
  (fn (_,i) => (SUBGOAL (fn (_,i) =>
     (TRY (REPEAT (EqSubst.eqsubst_tac ctxt [0] @{thms prod.sel} i)) THEN
     FIRST[EVERY[rts ctxt (@{thms Skip_inter_aux Throw_inter_aux Basic_inter_aux
Spec_inter_aux}) i,
              dest_inter_right_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rts ctxt (@{thms Seq_inter_aux Catch_inter_aux}) i,
              dest_inter_aux_Tac ctxt' (i+1),
              dest_inter_aux_Tac ctxt' (i+0)],
          EVERY[rt ctxt (@{thm Cond_inter_aux}) i,
              dest_inter_aux_Tac ctxt' (i+2),
              dest_inter_aux_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rt ctxt (@{thm While_inter_aux}) i,
              dest_inter_aux_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rt ctxt (@{thm Await_inter_aux}) i,
              dest_inter_right_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' (i+0)],
          EVERY[rt ctxt (@{thm Call_inter_aux}) i,
              Call_asm_inst ctxt (i+2),
              dest_inter_aux_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rt ctxt (@{thm DynCom_inter_aux}) i,
              dest_inter_aux_Tac ctxt' (i+3),
              dest_inter_right_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rt ctxt (@{thm Guard_inter_aux}) i,
              dest_inter_aux_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
          EVERY[rt ctxt (@{thm Parallel_inter_aux}) i,
              inter_aux_Par_Tac ctxt' (i+3),
              dest_inter_right_Tac ctxt' (i+2),
              dest_inter_right_Tac ctxt' (i+1),
              dest_inter_right_Tac ctxt' i],
```

```
          dest_inter_right_Tac ctxt' i]))
        THEN_ALL_NEW hyp_tac) i)) i st

and dest_inter_right_Tac (ctxt' as (ctxt,args)) i st =
  (Cache_Tactics.SUBGOAL_CACHE (nth args 2)
  (fn (_,i) =>
     FIRST[EVERY[rts ctxt (@{thms Skip_inter_right Throw_inter_right
                                  Basic_inter_right Spec_inter_right})
i,
                 HoareRuleTac ctxt' i],
           EVERY[rts ctxt (@{thms Seq_inter_right Catch_inter_right})
i,
                 dest_inter_right_Tac ctxt' (i+1),
                 dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm Cond_inter_right}) i,
                 dest_inter_right_Tac ctxt' (i+1),
                 dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm While_inter_right}) i,
                 dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm Await_inter_right}) i,
                 HoareRuleTac ctxt' (i+1),
                 simp ctxt (@{thms atom_com.simps}) i],
           EVERY[rt ctxt (@{thm Call_inter_right}) i,
                   Call_asm_inst ctxt (i+1),
                   dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm DynCom_inter_right}) i,
                   dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm Guard_inter_right}) i,
                   dest_inter_right_Tac ctxt' i],
           rt ctxt (@{thm Parallel_inter_right_empty}) i,
           EVERY[rt ctxt (@{thm Parallel_inter_right_List}) i,
                 dest_inter_right_Tac ctxt' (i+1),
                 dest_inter_right_Tac ctxt' i],
           EVERY[rt ctxt (@{thm Parallel_inter_right_Map}) i,
                 rt ctxt (@{thm allI}) i, rt ctxt (@{thm impI}) i,
                 dest_inter_right_Tac ctxt' i],
           K all_tac i])) i st
>
```

ML ‹

```
fun oghoare_tac ctxt =
   SUBGOAL (fn (_, i) =>
   TRY (prod_sel_simp ctxt i)
   THEN TRY (oghoare_simp ctxt i)
   THEN Cache_Tactics.cacheify_tactic 3 HoareRuleTac ctxt i)

(* oghoare_tac' fails if oghoare_tac does not do anything *)
fun oghoare_tac' ctxt i goal =
```

```
  let
    val results = oghoare_tac ctxt i goal;
  in
    if (Thm.eq_thm (results |> Seq.hd, goal) handle Option => false)
    then no_tac goal
    else results
  end;

fun oghoare_parallel_tac ctxt i =
  TRY (oghoare_simp ctxt i) THEN
  Cache_Tactics.cacheify_tactic 3 ParallelTac ctxt i
fun oghoare_interfree_tac ctxt i =
  TRY (oghoare_simp ctxt i) THEN
  Cache_Tactics.cacheify_tactic 3 interfree_Tac ctxt i
fun oghoare_interfree_aux_tac ctxt i =
  TRY (oghoare_simp ctxt i) THEN
  Cache_Tactics.cacheify_tactic 3 interfree_aux_Tac ctxt i
›

method_setup oghoare = ‹
  Scan.succeed (SIMPLE_METHOD' o oghoare_tac')›
  "verification condition generator for the oghoare logic"

method_setup oghoare_parallel = ‹
  Scan.succeed (SIMPLE_METHOD' o oghoare_parallel_tac)›
  "verification condition generator for the oghoare logic"

method_setup oghoare_interfree = ‹
  Scan.succeed (SIMPLE_METHOD' o oghoare_interfree_tac)›
  "verification condition generator for the oghoare logic"

method_setup oghoare_interfree_aux = ‹
  Scan.succeed (SIMPLE_METHOD' o oghoare_interfree_aux_tac)›
  "verification condition generator for the oghoare logic"

end
```

7 Shallowly-embedded syntax for COMPLX programs

```
theory OG_Syntax
imports
  OG_Hoare
  OG_Tactics
begin

datatype ('s, 'p, 'f) ann_com =
  AnnCom "('s, 'p, 'f) ann" "('s,'p,'f) com"
```

```
fun ann where "ann (AnnCom p q) = p"
fun com where "com (AnnCom p q) = q"

lemmas ann.simps[oghoare_simps] com.simps[oghoare_simps]

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
›

syntax
  "_fst" :: "'a × 'b ⇒ 'a"  ("⌜_₁⌝ [60] 61)
  "_snd" :: "'a × 'b ⇒ 'b"  ("⌜_₂⌝ [60] 61)

parse_translation ‹
  let
    fun fst_tr ((Const (@{const_syntax Pair}, _) $ p $ c)
          :: ts) = p;

    fun snd_tr ((Const (@{const_syntax Pair}, _) $ p $ c)
          :: ts) = c;
  in
   [(@{syntax_const "_fst"}, K fst_tr),
    (@{syntax_const "_snd"}, K snd_tr)]
  end
›
```

Syntax for commands and for assertions and boolean expressions in commands `com` and annotated commands `ann_com`.

```
syntax
  "_Annotation" :: "('s,'p,'f) ann_com ⇒ ('s, 'p, 'f) ann"  ("⌜_?⌝ [60]
61)
  "_Command" :: "('s,'p,'f) ann_com ⇒ ('s,'p,'f) com"  ("⌜_!⌝ [60] 61)

parse_translation ‹
  let
    fun ann_tr ((Const (@{const_syntax AnnCom}, _) $ p $ c)
```

```
          :: ts) = p
      | ann_tr (p :: ts) =
          Const (@{const_syntax ann}, dummyT) $ p
      | ann_tr x = raise TERM ("ann_tr", x);

    fun com_tr ((Const (@{const_syntax AnnCom}, _) $ p $ c)
          :: ts) = c
      | com_tr (c :: ts) =
          Const (@{const_syntax com}, dummyT) $ c
      | com_tr x = raise TERM ("com_tr", x);
  in
   [(@{syntax_const "_Annotation"}, K ann_tr),
    (@{syntax_const "_Command"}, K com_tr)]
  end
›
```

syntax

```
  "_Seq"  :: "('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(_,,/ _)› [55, 56] 55)
  "_AnnSeq"  :: "('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                  (‹(_;;//_)› [55, 56] 55)
```

translations

```
  "_Seq c1 c2" ⇀ "CONST AnnCom (CONST AnnComp (c1?) (c2?)) (CONST Seq
(c1!) (c2!))"
  "_AnnSeq c1 c2" ⇀ "CONST AnnCom (CONST AnnComp (c1?) (c2?)) (CONST Seq
(c1!) (c2!))"
```

syntax

```
  "_Assign"    :: "idt ⇒ 'b ⇒ ('s,'p,'f) ann_com"
                  (‹(´_ :=/ _)› [70, 65] 61)
  "_AnnAssign" :: "'s assn ⇒ idt ⇒ 'b ⇒ ('s,'p,'f) ann_com"
                  (‹(_//´_ :=/ _)› [90,70,65] 61)
```

definition "FAKE_ANN ≡ UNIV"

translations

```
  "r ´x := a" ⇀ "CONST AnnCom (CONST AnnExpr r)
                                (CONST Basic «´(_update_name x (λ_. a))»)"
  "´x := a" ⇌ "CONST FAKE_ANN ´x := a"
```

abbreviation

```
  "update_var f S s ≡ (λv. f (λ_. v) s) ‘ S"
```

abbreviation

```
  "fun_to_rel f ≡ ⋃ ((λs. (λv. (s, v)) ‘ f s) ‘ UNIV)"
```

syntax

```
  "_Spec"      :: "idt ⇒ 'b ⇒ ('s,'p,'f) ann_com"
                  (‹(´_ :∈/ _)› [70, 65] 61)
  "_AnnSpec"   :: "'a assn ⇒ idt ⇒ 'b ⇒ ('s,'p,'f) ann_com"
                  (‹(_//´_ :∈/ _)› [90,70,65] 61)
```

translations

```
  "r ´x :∈ S" ⇀ "CONST AnnCom (CONST AnnExpr r)
                               (CONST Spec (CONST fun_to_rel «´(CONST
update_var (_update_name x) S)»))"
  "´x :∈ S" ⇌ "CONST FAKE_ANN ´x :∈ S"
```

nonterminal grds **and** grd

syntax

```
  "_AnnCond1"    :: "'s assn ⇒ 's bexp  ⇒ ('s,'p,'f) ann_com  ⇒ ('s,'p,'f)
ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(_//IF _//(2THEN/ (_))//(2ELSE/ (_))//FI)›  [90,0,0,0]
61)
  "_AnnCond2"    :: "'s assn ⇒ 's bexp  ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                  (‹(_//IF _//(2THEN/ (_))//FI)›  [90,0,0] 61)
  "_AnnWhile"    :: "'s assn ⇒ 's bexp  ⇒ 's assn ⇒ ('s,'p,'f) ann_com
⇒ ('s,'p,'f) ann_com"
                  (‹(_//WHILE _/ INV _//(2DO/ (_))//OD)›  [90,0,0,0]
61)
  "_AnnAwait"    :: "'s assn ⇒ 's bexp  ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                  (‹(_//AWAIT _/ (2THEN/ (_))/ END)›  [90,0,0] 61)
  "_AnnAtom"     :: "'s assn  ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(_//⟨_⟩)› [90,0] 61)
  "_AnnWait"     :: "'s assn ⇒ 's bexp ⇒ ('s,'p,'f) ann_com"
                  (‹(_//WAIT _/ END)› [90,0] 61)

  "_Cond"        :: "'s bexp ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com
⇒ ('s,'p,'f) ann_com"
                  (‹(IF _//(2THEN/ (_))//(2ELSE/ (_))//FI)› [0, 0,
0] 61)
  "_Cond2"       :: "'s bexp ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(IF _//(2THEN/ (_))//FI)› [0,0] 56)
  "_While_inv"   :: "'s bexp ⇒ 's assn ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                  (‹(WHILE _/ INV _//(2DO/ (_))//OD)›  [0, 0, 0] 61)
  "_While"       :: "'s bexp ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(WHILE _//(2DO/ (_))//OD)›  [0, 0] 61)
  "_Await"       :: "'s bexp  ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(AWAIT _/ (2THEN/ (_))/ END)›  [0,0] 61)
  "_Atom"        :: "('s,'p,'f) ann_com ⇒ ('s,'p,'f) ann_com"
                  (‹(⟨_⟩)› [0] 61)
```

```
  "_Wait"        :: "'s bexp ⇒ ('s,'p,'f) ann_com"
                    (‹(WAIT _/ END)› [0] 61)
  "_grd"         :: "'f ⇒ 's bexp ⇒ grd"
                    (‹'(_, _')› [1000] 1000)
  "_last_grd"    :: "grd ⇒ grds"   (‹_› 1000)
  "_grds"        :: "[grd, grds] ⇒ grds"
                    (‹(_,/ _)› [999,1000] 1000)
  "_guards"      :: "'s assn ⇒ grds  ⇒ ('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                    (‹(_//(2_ ⟼/ (_)))› [90, 0, 56] 61)
  "_Throw"       :: "('s,'p,'f) ann_com"
                    (‹THROW› 61)
  "_AnnThrow"    :: "'s assn ⇒ ('s,'p,'f) ann_com"
                    (‹(_/ THROW)› [90] 61)
  "_Try_Catch"   :: "('s,'p,'f) ann_com ⇒('s,'p,'f) ann_com ⇒ ('s,'p,'f)
ann_com"
                    (‹((2TRY/ (_))//(2CATCH/ (_))/ END)›  [0,0] 71)
  "_AnnCallX"    :: "'s assn ⇒ ('s ⇒ 's) ⇒ 's assn  ⇒ 'p ⇒ nat ⇒
('s ⇒ 's ⇒ 's) ⇒ ('s⇒'s⇒('s,'p,'f) com) ⇒ 's assn ⇒ 's assn ⇒ 's
assn ⇒ 's assn ⇒ ('s,'p,'f) ann_com"
                    (‹(_//(2CALLX/ (_)//_/ _/ _//_/ _//_/ _//_/ _))›
                      [90,1000,0,1000,0,1000,1000,0,0,0,0] 61)
  "_AnnSCall"    :: "'s assn ⇒ 'p ⇒ nat ⇒ ('s,'p,'f) ann_com"
                    (‹(_//SCALL _/ _)› [90,0,0] 61)
  "_Skip"        :: "('s,'p,'f) ann_com"
                    (‹SKIP› 61)
  "_AnnSkip"     :: "'s assn ⇒ ('s,'p,'f) ann_com"
                    (‹(_/ SKIP)› [90] 61)
```

translations

```
  "r IF b THEN c1 ELSE c2 FI" ⇀
    "CONST AnnCom (CONST AnnBin r (c1?) (c2?)) (CONST Cond {|b|} (c1!) (c2!))"
  "r IF b THEN c FI" ⇀ "r IF b THEN c ELSE SKIP FI"
  "r WHILE b INV i DO c OD" ⇀
    "CONST AnnCom (CONST AnnWhile r i (c?)) (CONST While {|b|} (c!))"
  "r AWAIT b THEN c END" ⇀
    "CONST AnnCom (CONST AnnRec r (c?)) (CONST Await {|b|} (c!))"
  "r ⟨c⟩" ⇌ "r AWAIT CONST True THEN c END"
  "r WAIT b END" ⇌ "r AWAIT b THEN SKIP END"

  "IF b THEN c1 ELSE c2 FI" ⇌ "CONST FAKE_ANN IF b THEN c1 ELSE c2 FI"
  "IF b THEN c FI" ⇌ "CONST FAKE_ANN IF b THEN c ELSE SKIP FI"
  "WHILE b DO c OD" ⇌ "CONST FAKE_ANN WHILE b INV CONST FAKE_ANN DO c
OD"
  "WHILE b INV i DO c OD" ⇌ "CONST FAKE_ANN WHILE b INV i DO c OD"
  "AWAIT b THEN c END" ⇌ "CONST FAKE_ANN AWAIT b THEN c END"
  "⟨c⟩" ⇌ "CONST FAKE_ANN AWAIT CONST True THEN c END"
  "WAIT b END" ⇌ "AWAIT b THEN SKIP END"
```

```
  "_grd f g" ⇀ "(f, g)"
  "_grds g gs" ⇀ "g#gs"
  "_last_grd g" ⇀ "[g]"
  "_guards r gs c" ⇀
    "CONST AnnCom (CONST ann_guards r gs (c?)) (CONST guards gs (c!))"

  "ai CALLX init r p n restore return arestoreq areturn arestorea A" ⇀
    "CONST AnnCom (CONST ann_call ai r n arestoreq areturn arestorea A)
                  (CONST call init p restore return)"
  "r SCALL p n" ⇀ "CONST AnnCom (CONST AnnCall r n) (CONST Call p)"

  "r THROW" ⇌ "CONST AnnCom (CONST AnnExpr r) (CONST Throw)"
  "THROW" ⇌ "CONST FAKE_ANN THROW"
  "TRY c1 CATCH c2 END" ⇀ "CONST AnnCom (CONST AnnComp (c1?) (c2?))
                                        (CONST Catch (c1!) (c2!))"

  "r SKIP" ⇌ "CONST AnnCom (CONST AnnExpr r) (CONST Skip)"
  "SKIP" ⇌ "CONST FAKE_ANN SKIP"
```

nonterminal prgs

syntax

```
  "_PAR" :: "prgs ⇒ 'a"              (‹(COBEGIN//_//COEND)› [57] 56)
  "_prg" :: "['a, 'a, 'a] ⇒ prgs"        (‹(2  _//_,/ _)› [60, 90, 90]
57)
  "_prgs" :: "['a, 'a, 'a, prgs] ⇒ prgs"  (‹(2  _//_,/ _)//‖//_› [60,90,90,57]
57)

  "_prg_scheme" :: "['a, 'a, 'a, 'a, 'a, 'a] ⇒ prgs"
                 (‹  (2SCHEME [_ ≤ _ < _]//_//_,/ _)› [0,0,0,60, 90,90]
57)
```

translations

```
  "_prg c q a" ⇀ "([((c?), q, a)], [(c!)])"
  "_prgs c q a ps" ⇀ "(((c?), q, a) # (ps¸), (c!) # (ps˙))"
  "_PAR ps" ⇀ "CONST AnnCom (CONST AnnPar (ps¸)) (CONST Parallel (ps˙))"

  "_prg_scheme j i k c q a" ⇀ "(CONST map (λi. ((c?), q, a)) [j..<k],
CONST map (λi. (c!)) [j..<k])"
```

syntax

```
  "_oghoare" :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
             ⇒ ('s,'p,'f) ann_com ⇒ 's assn ⇒ 's assn ⇒ bool"
    (‹(4_),/ (4_)/ (⊢›/_ (_//_, _))› [60,60,60,20,1000,1000]60)

  "_oghoare_seq" :: "('s,'p,'f) body ⇒ ('s,'p,'f) proc_assns ⇒ 'f set
            ⇒'s assn ⇒ ('s,'p,'f) ann_com ⇒ 's assn ⇒ 's assn ⇒
bool"
    (‹(4_),/ (4_)/ (⊩›/_ (_//_//_, _))› [60,60,60,1000,20,1000,1000]60)
```

translations

```
  "_oghoare Γ Θ F c Q A" ⇀ "Γ, Θ ⊢/F (c?) (c!) Q, A"
  "_oghoare_seq Γ Θ F P c Q A" ⇀ "Γ, Θ ⊫/F P (c?) (c!) Q, A"
```

ML ‹`val syntax_debug = false`›

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 annbexp_tr' name (r :: (Const (@{const_syntax Collect}, _) $ t)
:: ts) =
        annquote_tr' (Syntax.const name) (r :: t :: ts)
    | annbexp_tr' name (r :: Const (@{const_syntax UNIV}, _) :: ts) =
        annquote_tr' (Syntax.const name)
                     (r :: Abs ("s", dummyT, Const (@{const_syntax True},
dummyT)) :: ts)
    | annbexp_tr' name (r :: Const (@{const_syntax Set.empty}, _) :: ts)
=
        annquote_tr' (Syntax.const name)
                     (r :: Abs ("s", dummyT, Const (@{const_syntax False},
dummyT)) :: ts)
    | annbexp_tr' name x =
        let val _ = if syntax_debug then writeln ("annbexp_tr'\n " ^ @{make_string}
x) else () in
        raise Match end;

  fun annassign_tr' (r :: Abs (x, _, f $ k $ Bound 0) :: ts) =
        quote_tr' (Syntax.const @{syntax_const "_AnnAssign"} $ r $ Syntax_Trans.update_name_
f)
          (Abs (x, dummyT, Syntax_Trans.const_abs_tr' k) :: ts)
    | annassign_tr' r = let val _ = writeln ("annassign_tr'\n " ^ @{make_string}
r) in
     raise Match end;

  fun dest_list (Const (@{const_syntax Nil}, _)) = []
    | dest_list (Const (@{const_syntax Cons}, _) $ x $ xs) = x :: dest_list
xs
    | dest_list _ = raise Match;
```

```
  fun guards_lst_tr' [fg] = fg
    | guards_lst_tr' (t :: ts) =
        Syntax.const @{syntax_const "_grds"} $ t $ guards_lst_tr' ts
    | guards_lst_tr' [] = raise Match;

  fun new_AnnCom r c =
    (Const (@{const_syntax AnnCom}, dummyT) $ r $ c)

  fun new_Pair a b =
    (Const (@{const_syntax Pair}, dummyT) $ a $ b)

  fun dest_prod (Const (@{const_syntax Pair}, _) $ a $ b) = (a, b)
    | dest_prod _ = raise Match;

  fun prgs_tr' f pqa c =
        let val (p, qa) = dest_prod pqa
            val (q, a) = dest_prod qa
        in [new_AnnCom (f p) (f c), f q, f a] end

  fun prgs_lst_tr' [p] [c] =
        list_comb (Syntax.const @{syntax_const "_prg"},
                   prgs_tr' I p c)
    | prgs_lst_tr' (p :: ps) (c :: cs) =
        list_comb (Syntax.const @{syntax_const "_prgs"},
                   prgs_tr' I p c) $ prgs_lst_tr' ps cs
    | prgs_lst_tr' _ _= raise Match;

  fun AnnCom_tr (Const (@{const_syntax AnnPar}, _) $
                 (Const (@{const_syntax map}, _) $
                  Abs (i, T, p) $ (Const _ $ j $ k)) ::
                 Const (@{const_syntax Parallel}, _) $
                 (Const (@{const_syntax map}, _) $
                  Abs (_, _, c) $ _) :: ts) =
        let val _ = if syntax_debug then writeln "prg_scheme" else ()
            fun dest_abs body = snd (Term.dest_abs_global (Abs (i, T,
body))) in
          Syntax.const @{syntax_const "_PAR"} $
            list_comb (Syntax.const @{syntax_const "_prg_scheme"} $
              j $ Free (i, T) $ k, prgs_tr' dest_abs p c)
        end
    | AnnCom_tr (Const (@{const_syntax AnnPar}, _) $ ps ::
              Const (@{const_syntax Parallel}, _) $ cs :: ts) =
        let val _ = if syntax_debug then writeln "Par" else ()
            val (ps', cs') = (dest_list ps, dest_list cs)
        in Syntax.const @{syntax_const "_PAR"} $
             prgs_lst_tr' ps' cs' end
    | AnnCom_tr (Const (@{const_syntax AnnExpr}, _) $ r ::
              Const (@{const_syntax Basic}, _) $ Abs (x, _, f $ k $ Bound
0) :: ts) =
```

```
      let val _ = if syntax_debug then writeln "Basic'" else () in
        quote_tr' (Syntax.const @{syntax_const "_AnnAssign"} $ r $ Syntax_Trans.update_name_
f)
          (Abs (x, dummyT, Syntax_Trans.const_abs_tr' k) :: ts) end
    | AnnCom_tr (Const (@{const_syntax AnnExpr}, _) $ r ::
              Const (@{const_syntax Basic}, _) $ (f $ k) :: ts) =
      let val _ = if syntax_debug then writeln "Basic" else () in
        quote_tr' (Syntax.const @{syntax_const "_AnnAssign"} $ r $ Syntax_Trans.update_name_
f)
          (k :: ts) end
    | AnnCom_tr (Const (@{const_syntax AnnExpr}, _) $ r ::
              Const (@{const_syntax Spec}, _) $ (_ $ _ $ Abs (_,_, _
$ _ $ ((_ $ f) $ S $ _))) :: ts) =
      let val _ = if syntax_debug then writeln ("Spec") else () in
        (Syntax.const @{syntax_const "_AnnSpec"} $ r $
           Syntax_Trans.update_name_tr' f $
           Syntax_Trans.antiquote_tr' @{syntax_const "_antiquote"} S)
      end
    | AnnCom_tr (Const (@{const_syntax AnnComp}, _) $ r $ r' ::
              Const (@{const_syntax Seq}, _) $ c $ c' :: ts) =
      let val _ = if syntax_debug then writeln "Seq" else ()
      in Syntax.const @{syntax_const "_AnnSeq"} $
               new_AnnCom r c $ new_AnnCom r' c' end
    | AnnCom_tr (Const (@{const_syntax AnnRec}, _) $ r $ p ::
              Const (@{const_syntax Await}, _) $ b $ c :: ts) =
       let val _ = if syntax_debug then writeln "Await" else ()
          in annbexp_tr' @{syntax_const "_AnnAwait"}
                  (r :: b :: new_AnnCom p c :: ts) end
    | AnnCom_tr (Const (@{const_syntax AnnWhile}, _) $ r $ i $ p ::
              Const (@{const_syntax While}, _) $ b $ c :: ts) =
       let val _ = if syntax_debug then writeln "While" else ()
          in annbexp_tr' @{syntax_const "_AnnWhile"}
                  (r :: b :: i :: new_AnnCom p c :: ts) end
    | AnnCom_tr (Const (@{const_syntax AnnBin}, _) $ r $ p $ p' ::
              Const (@{const_syntax Cond}, _) $ b $ c $ c':: ts) =
       let val _ = if syntax_debug then writeln "Cond" else ()
          in annbexp_tr' @{syntax_const "_AnnCond1"}
                  (r :: b :: new_AnnCom p c :: new_AnnCom p' c' :: ts)
end
    | AnnCom_tr (Const (@{const_syntax ann_guards}, _) $ r $ gs $ p ::
              Const (@{const_syntax guards}, _) $ _ $ c :: ts) =
       let val _ = if syntax_debug then writeln "guards" else ()
          in Syntax.const @{syntax_const "_guards"} $ r $
                guards_lst_tr' (dest_list gs) $ new_AnnCom p c end
    | AnnCom_tr (Const (@{const_syntax AnnRec}, _) $ r $ p ::
              Const (@{const_syntax Guard}, _) $ f $ g $ c :: ts) =
       let val _ = if syntax_debug then writeln "guards'" else ()
          in Syntax.const @{syntax_const "_guards"} $ r $
                new_Pair f g $ new_AnnCom p c end
```

```
    | AnnCom_tr (Const (@{const_syntax AnnCall}, _) $ r $ n ::
              Const (@{const_syntax Call}, _) $ p :: ts) =
       let val _ = if syntax_debug then writeln "SCall" else ()
          in Syntax.const @{syntax_const "_AnnSCall"} $ r $ p $ n end
    | AnnCom_tr (Const (@{const_syntax ann_call}, _) $
                  ai $ r $ n $ arestoreq $ areturn $ arestorea $ A ::
              Const (@{const_syntax call}, _) $
                  init $ p $ restore $ return :: ts) =
       let val _ = if syntax_debug then writeln "CallX" else ()
          in Syntax.const @{syntax_const "_AnnCallX"} $ ai $ init $
                  r $ p $ n $ restore $ return $ arestoreq $ areturn $
                  arestorea $ A end
    | AnnCom_tr (Const (@{const_syntax AnnComp}, _) $ r $ r' ::
              Const (@{const_syntax Catch}, _) $ c $ c' :: ts) =
      let val _ = if syntax_debug then writeln "Catch" else ()
      in Syntax.const @{syntax_const "_Try_Catch"} $
               new_AnnCom r c $ new_AnnCom r' c' end
    | AnnCom_tr (Const (@{const_syntax ann}, _) $ p ::
              Const (@{const_syntax com}, _) $ p' :: ts) =
       let val _ = if syntax_debug then writeln "ann_com" else ()
          in if p = p' then p else raise Match end
    | AnnCom_tr x = let val _ = if syntax_debug then writeln ("AnnCom_tr\n
" ^ @{make_string} x) else ()
          in raise Match end;

    fun oghoare_tr (gamma :: sigma :: F :: r :: c :: Q :: A :: ts) =
        let val _ = if syntax_debug then writeln "oghoare" else ()
          in Syntax.const @{syntax_const "_oghoare"} $
               gamma $ sigma $ F $ new_AnnCom r c $ Q $ A
          end
      | oghoare_tr x = let val _ = writeln ("oghoare_tr\n " ^ @{make_string}
x)
            in raise Match end;

    fun oghoare_seq_tr (gamma :: sigma :: F :: P :: r :: c :: Q :: A ::
ts) =
        let val _ = if syntax_debug then writeln "oghoare_seq" else ()
          in Syntax.const @{syntax_const "_oghoare_seq"} $
               gamma $ sigma $ F $ P $ new_AnnCom r c $ Q $ A
          end
      | oghoare_seq_tr x = let val _ = writeln ("oghoare_seq_tr\n " ^
@{make_string} x)
            in raise Match end;

  in
   [(@{const_syntax Collect}, K assert_tr'),
    (@{const_syntax AnnCom}, K AnnCom_tr),
    (@{const_syntax oghoare}, K oghoare_tr),
    (@{const_syntax oghoare_seq}, K oghoare_seq_tr)]
```

```
  end
›
```

end

8 Examples

theory Examples
imports
  "../OG_Syntax"
begin

record test =
  x :: nat
  y :: nat

This is a sequence of two commands, the first being an assign protected by two guards. The guards use booleans as their faults.

definition
  "test_guard ≡ ⦃True⦄ (True, ⦃´x=0⦄),
                    (False, ⦃(0::nat)=0⦄) ⟼ ⦃True⦄ ´y := 0;;
                   ⦃True⦄ ´x := 0"

lemma
  "Γ, Θ ⊢$_{/\{True\}}$
    COBEGIN test_guard ⦃True⦄,⦃True⦄
         ∥ ⦃True⦄ ´y:=0 ⦃True⦄, ⦃True⦄
    COEND ⦃True⦄,⦃True⦄"
  unfolding test_guard_def
  apply oghoare
          apply simp_all
  done

definition
  "test_try_throw ≡ TRY ⦃True⦄ ´y := 0;;
                    ⦃True⦄ THROW
                   CATCH ⦃True⦄ ´x := 0
                   END"

8.1 Parameterized Examples

8.1.1 Set Elements of an Array to Zero

record Example1 =
  ex1_a :: "nat ⇒ nat"

lemma Example1:
 "Γ, Θ⊨$_{/F}$⦃True⦄

```
   COBEGIN SCHEME [0≤i<n] {|True|} ´ex1_a:=´ex1_a (i:=0) {| ´ex1_a i=0|},
{|False|} COEND
  {|∀i < n. ´ex1_a i = 0|}, X"
  apply oghoare
        apply (simp ; fail)+
 done
```

Same example but with a Call.

```
definition
   "Example1'a ≡ {|True|} ´ex1_a:=´ex1_a (0:=0)"

definition
   "Example1'b ≡ {|True|} ´ex1_a:=´ex1_a (1:=0)"

definition "Example1' ≡
  COBEGIN Example1'a {| ´ex1_a 0=0|}, {|False|}
         ||
         {|True|} SCALL 0 0
         {| ´ex1_a 1=0|}, {|False|}
  COEND"

definition "Γ' = Map.empty(0 ↦ com Example1'b)"
definition "Θ' = Map.empty(0 :: nat ↦ [ann Example1'b])"

lemma Example1_proc_simp[unfolded Example1'b_def oghoare_simps]:
 "Γ' 0 = Some (com (Example1'b))"
 "Θ' 0 = Some ([ ann(Example1'b)])"
 "[ ann(Example1'b)]!0 = ann(Example1'b)"
 by (simp add: Γ'_def Θ'_def)+

lemma Example1':
notes Example1_proc_simp[proc_simp]
shows
 "Γ', Θ' |⊢/F Example1' {|∀i < 2. ´ex1_a i = 0|}, {|False|}"
  unfolding Example1'_def
  apply simp
  unfolding Example1'a_def Example1'b_def
  apply oghoare
             apply simp+
   using less_2_cases apply blast
  apply simp
  apply (erule disjE ; clarsimp)
done

type_synonym routine = nat
```

Same example but with a Call.

```
record Example2 =
  ex2_n :: "routine ⇒ nat"
```

```
  ex2_a :: "nat ⇒ string"

definition
  Example2'n :: "routine ⇒ (Example2, string × nat, 'f) ann_com"
where
  "Example2'n i ≡ {| ´ex2_n i= i|} ´ex2_a:=´ex2_a((´ex2_n i):='''')"

lemma Example2'n_map_of_simps[simp]:
  "i < n ⟹
    map_of (map (λi. ((p, i), g i)) [0..<n])
       (p, i) = Some (g i)"
  apply (rule map_of_is_SomeI)
   apply (clarsimp simp: distinct_map o_def)
   apply (meson inj_onI prod.inject)
  apply clarsimp
  done

definition "Γ'' n ≡
  map_of (map (λi. ((''f'', i), com (Example2'n i))) [0..<n])"

definition "Θ'' n ≡
  map_of (map (λi. ((''f'', i), [ann (Example2'n i)])) [0..<n])"

lemma  Example2'n_proc_simp[unfolded Example2'n_def oghoare_simps]:
 "i<n ⟹ Γ'' n (''f'',i) = Some ( com(Example2'n i))"
 "i<n ⟹ Θ'' n (''f'',i) = Some ([ ann(Example2'n i)])"
 "[ ann(Example2'n i)]!0 = ann(Example2'n i)"
 by (simp add: Γ''_def Θ''_def)+

lemmas Example2'n_proc_simp[proc_simp add]

lemma Example2:
notes Example2'n_proc_simp[proc_simp]
shows
 "Γ'' n, Θ'' n
 |⊨_/F{|True|}
   COBEGIN SCHEME [0≤i<n]
     {|True|}
       CALLX (λs. s(|ex2_n:=(ex2_n s)(i:=i)|)) {| ´ex2_n i = i|} (''f'', i)
0
       (λs t. t(|ex2_n:= (ex2_n t)(i:=(ex2_n s) i)|)) (λx y. Skip)
       {| ´ex2_a (´ex2_n i)='''' ∧ ´ex2_n i = i|} {| ´ex2_a i='''' |} {|False|}
{|False|}
     {| ´ex2_a i='''' |}, {|False|}
   COEND
  {|∀i < n. ´ex2_a i = '''' |}, {|False|}"
  unfolding Example2'n_def ann_call_def call_def block_def
  apply oghoare
  apply (clarsimp ; fail)+
```

```
done
```

lemmas `Example2'n_proc_simp[proc_simp del]`

Same example with lists as auxiliary variables.

```
record Example2_list =
  ex2_A :: "nat list"

lemma Example2_list:
 "Γ, Θ |⊨_/F{|n < length ´ex2_A|}
   COBEGIN
     SCHEME [0≤i<n] {|n < length ´ex2_A|} ´ex2_A:=´ex2_A[i:=0] {|´ex2_A!i=0|},{|False|}

   COEND
    {|∀i < n. ´ex2_A!i = 0|}, X"
  apply oghoare
       apply force+
 done


lemma exceptions_example:
  "Γ, Θ |⊢_/F
   TRY
   {|True |} ´y := 0;;
   {| ´y = 0 |} THROW
   CATCH
     {|´y = 0|} ´x := ´y + 1
   END
   {| ´x = 1 ∧ ´y = 0|}, {|False|}"
  by oghoare simp_all

lemma guard_example:
  "Γ, Θ |⊢_/{42,66}
  {|True|} (42, {|´x=0|}),
   (66, {|´y=0|}) ⟼ {|´x = 0|}
   ´y := 0;;
   {|True|} ´x := 0
  {| ´x = 0|}, {|False|}"
  apply oghoare
       apply simp_all
 done
```

8.1.2 Peterson's mutex algorithm I (from Hoare-Parallel)

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 peterson_thread_1:
 "Γ, Θ |⊢/F {| ´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|},{|False|}
"
  apply oghoare
        apply (((auto)[1]) ; fail)+
 done


lemma peterson_thread_2:
 "Γ, Θ |⊢/F  {| ´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|},{|False|}
 "
  apply oghoare
  by auto

lemma Petersons_mutex_1:
  "Γ, Θ |⊩/F {| ´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|},{|False|}
  ‖
```

```
  {| ´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|},{|False|}
  COEND
  {| ´pr1=0 ∧ ¬ ´in1 ∧ ´pr2=0 ∧ ¬ ´in2|},{|False|}"
  apply oghoare
— 81 verification conditions.
  by auto

end
```

9 Case-study

```
theory SumArr
imports
  "../OG_Syntax"
  Word_Lib.Word_32
begin

unbundle bit_operations_syntax

type_synonym routine = nat
type_synonym word32 = "32 word"
type_synonym funcs = "string × nat"
datatype faults = Overflow | InvalidMem
type_synonym 'a array = "'a list"
```

Sumarr computes the combined sum of all the elements of multiple arrays. It does this by running a number of threads in parallel, each computing the sum of elements of one of the arrays, and then adding the result to a global variable gsum shared by all threads.

```
record sumarr_state =
 — local variables of threads
  tarr :: "routine ⇒ word32 array"
  tid :: "routine ⇒ word32"
  ti :: "routine ⇒ word32"
  tsum :: "routine ⇒ word32"
 — global variables
  glock :: nat
  gsum :: word32
  gdone :: word32
```

```
  garr :: "(word32 array) array"
```
— ghost variables
```
  ghost_lock :: "routine ⇒ bool"
```

definition
```
 NSUM :: word32
```
where
```
 "NSUM = 10"
```

definition
```
 MAXSUM :: word32
```
where
```
 "MAXSUM = 1500"
```

definition
```
 array_length :: "'a array ⇒ word32"
```
where
```
 "array_length arr ≡ of_nat (length arr)"
```

definition
```
 array_nth :: "'a array ⇒ word32 ⇒'a"
```
where
```
 "array_nth arr n ≡ arr ! unat n"
```

definition
```
 array_in_bound :: "'a array ⇒ word32 ⇒ bool"
```
where
```
 "array_in_bound arr idx ≡ unat idx < (length arr)"
```

definition
```
  array_nat_sum :: "('a :: len) word array ⇒ nat"
```
where
```
  "array_nat_sum arr ≡ sum_list (map unat arr)"
```

definition
```
  "local_sum arr ≡ of_nat (min (unat MAXSUM) (array_nat_sum arr))"
```

definition
```
  "global_sum arr ≡ sum_list (map local_sum arr)"
```

definition
```
  "tarr_inv s i ≡
    length (tarr s i) = unat NSUM ∧ tarr s i = garr s ! i"
```

abbreviation
```
  "sumarr_inv_till_lock s i ≡ ¬ bit (gdone s) i ∧ ((¬ (ghost_lock s)
(1 - i)) ⟶ ((gdone s = 0 ∧ gsum s = 0) ∨
    (bit (gdone s) (1 - i) ∧ gsum s = local_sum (garr s !(1 - i)))))"
```

abbreviation
```
  "lock_inv s ≡
    (glock s = fromEnum (ghost_lock s 0) + fromEnum (ghost_lock s 1))
∧
    (¬(ghost_lock s) 0 ∨ ¬(ghost_lock s) 1)"
```

abbreviation
```
  "garr_inv s i ≡ (∃a b. garr s = [a, b]) ∧
    length (garr s ! (1-i)) = unat NSUM"
```

abbreviation
```
  "sumarr_inv s i ≡ lock_inv s ∧ tarr_inv s i ∧ garr_inv s i ∧
    tid s i = (of_nat i + 1)"
```

definition
```
  lock :: "routine ⇒ (sumarr_state, funcs, faults) ann_com"
```
where
```
  "lock i ≡
    {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´sumarr_inv_till_lock
i|}
    AWAIT ´glock = 0
    THEN ´glock:=1,, ´ghost_lock:=´ghost_lock (i:= True)
    END"
```

definition
```
 "sumarr_in_lock1 s i ≡ ¬bit (gdone s) i ∧ ((gdone s = 0 ∧ gsum s = local_sum
(tarr s i)) ∨
   (bit (gdone s) (1 - i) ∧ ¬ bit (gdone s) i ∧ gsum s = global_sum (garr
s)))"
```

definition
```
 "sumarr_in_lock2 s i ≡ (bit (gdone s) i ∧ ¬ bit (gdone s) (1 - i) ∧
gsum s = local_sum (tarr s i)) ∨
   (bit (gdone s) i ∧ bit (gdone s) (1 - i) ∧ gsum s = global_sum (garr
s))"
```

definition
```
  unlock :: "routine ⇒ (sumarr_state, funcs, faults) ann_com"
```
where
```
  "unlock i ≡
    {|  ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´glock = 1 ∧
    ´ghost_lock i ∧ bit ´gdone (unat (´tid i - 1)) ∧ ´sumarr_in_lock2
i |}
    ⟨´glock := 0,, ´ghost_lock:=´ghost_lock (i:= False)⟩"
```

definition
```
 "local_postcond s i ≡ (¬ (ghost_lock s) (1 - i) ⟶ gsum s = (if bit
(gdone s) 0 ∧ bit (gdone s) 1
              then global_sum (garr s)
```

```
            else local_sum (garr s ! i))) ∧ bit (gdone s) i ∧ ¬ghost_lock
s i"

definition
  sumarr :: "routine ⇒ (sumarr_state, funcs, faults) ann_com"
where
  "sumarr i ≡
  {| ´sumarr_inv i ∧ ´sumarr_inv_till_lock i|}
  ´tsum:=´tsum(i:=0) ;;
  {| ´tsum i = 0 ∧ ´sumarr_inv i ∧ ´sumarr_inv_till_lock i|}
  ´ti:=´ti(i:=0) ;;
  TRY
    {| ´tsum i = 0 ∧ ´sumarr_inv i ∧ ´ti i = 0 ∧ ´sumarr_inv_till_lock
i|}
    WHILE ´ti i < NSUM
    INV {| ´sumarr_inv i ∧ ´ti i ≤ NSUM ∧ ´tsum i ≤ MAXSUM ∧
          ´tsum i = local_sum (take (unat (´ti i)) (´tarr i)) ∧ ´sumarr_inv_till_lock
i|}
    DO
     {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ ´tsum i ≤ MAXSUM ∧
       ´tsum i = local_sum (take (unat (´ti i)) (´tarr i)) ∧ ´sumarr_inv_till_lock
i|}
     (InvalidMem, {| array_in_bound (´tarr i)  (´ti i) |}) ⟼
       {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ ´tsum i ≤ MAXSUM ∧
         ´tsum i = local_sum (take (unat (´ti i)) (´tarr i)) ∧ ´sumarr_inv_till_lock
i|}
       ´tsum:=´tsum(i:=´tsum i + array_nth (´tarr i) (´ti i));;
     {| ´sumarr_inv i ∧ ´ti i < NSUM ∧
        local_sum (take (unat (´ti i)) (´tarr i)) ≤ MAXSUM ∧
        (´tsum i < MAXSUM ∧ array_nth (´tarr i) (´ti i) < MAXSUM ⟶
       ´tsum i = local_sum (take (Suc (unat (´ti i))) (´tarr i))) ∧
        (array_nth (´tarr i) (´ti i) ≥ MAXSUM ∨ ´tsum i ≥ MAXSUM⟶
          local_sum (´tarr i) = MAXSUM)  ∧
       ´sumarr_inv_till_lock i |}
     (InvalidMem, {| array_in_bound (´tarr i)  (´ti i) |}) ⟼
       {| ´sumarr_inv i ∧ ´ti i < NSUM ∧
         (´tsum i < MAXSUM ∧ array_nth (´tarr i) (´ti i) < MAXSUM ⟶
           ´tsum i = local_sum (take (Suc (unat (´ti i))) (´tarr i)))
∧
         (array_nth (´tarr i) (´ti i) ≥ MAXSUM ∨ ´tsum i ≥ MAXSUM ⟶
           local_sum (´tarr i) = MAXSUM) ∧
         ´sumarr_inv_till_lock i|}
       IF array_nth (´tarr i) (´ti i) ≥ MAXSUM ∨ ´tsum i ≥ MAXSUM
       THEN
         {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ local_sum (´tarr i) = MAXSUM
∧ ´sumarr_inv_till_lock i|}
         ´tsum:=´tsum(i:=MAXSUM);;
         {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ ´tsum i ≤ MAXSUM ∧
           ´tsum i = local_sum (´tarr i) ∧ ´sumarr_inv_till_lock i |}
```

```
        THROW
      ELSE
        {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ ´tsum i ≤ MAXSUM ∧
          ´tsum i = local_sum (take (Suc (unat (´ti i))) (´tarr i))
∧ ´sumarr_inv_till_lock i|}
        SKIP
      FI;;
    {| ´sumarr_inv i ∧ ´ti i < NSUM ∧ ´tsum i ≤ MAXSUM ∧
      ´tsum i = local_sum (take (Suc (unat (´ti i))) (´tarr i)) ∧ ´sumarr_inv_till_lock
i |}
    ´ti:=´ti(i:=´ti i + 1)
  OD
 CATCH
   {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´sumarr_inv_till_lock
i|} SKIP
 END;;
 {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´sumarr_inv_till_lock
i|}
 SCALL (''lock'', i) 0;;
 {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´glock = 1 ∧
   ´ghost_lock i ∧ ´sumarr_inv_till_lock i |}
 ´gsum:=´gsum + ´tsum i ;;
 {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´glock = 1 ∧
   ´ghost_lock i ∧ ´sumarr_in_lock1 i |}
 ´gdone:=(´gdone OR ´tid i) ;;
 {| ´sumarr_inv i ∧ ´tsum i = local_sum (´tarr i) ∧ ´glock = 1 ∧
   ´ghost_lock i ∧ bit ´gdone (unat (´tid i - 1)) ∧ ´sumarr_in_lock2
i |}
 SCALL (''unlock'', i) 0"
```

definition
precond
where

```
"precond s ≡ (glock s) = 0 ∧ (gsum s) = 0∧ (gdone s) = 0 ∧
              (∃a b. garr s = [a, b]) ∧
              (∀xs∈set (garr s). length xs = unat NSUM) ∧
              (ghost_lock s) 0 = False ∧ (ghost_lock s) 1 = False"
```

definition
postcond
where

```
"postcond s ≡ (gsum s) = global_sum (garr s) ∧
              (∀i < 2. bit (gdone s) i)"
```

definition

```
 "call_sumarr i ≡
   {|length (´garr ! i) = unat NSUM ∧ ´lock_inv ∧ ´garr_inv i ∧
    ´sumarr_inv_till_lock i|}
   CALLX (λs. s(|tarr:=(tarr s)(i:=garr s ! i),
```

```
              tid:=(tid s)(i:=of_nat i+1),
              ti:=(ti s)(i:=undefined),
              tsum:=(tsum s)(i:=undefined)⦈)
          {⦇´sumarr_inv i ∧ ´sumarr_inv_till_lock i⦈}
          (''sumarr'', i) 0
          (λs t. t⦇tarr:= (tarr t)(i:=(tarr s) i),
                   tid:=(tid t)(i:=(tid s i)),
                   ti:=(ti t)(i:=(ti s i)),
                   tsum:=(tsum t)(i:=(tsum s i))⦈)
          (λ_ _. Skip)
          {⦇´local_postcond i⦈} {⦇´local_postcond i⦈}
          {⦇False⦈} {⦇False⦈}"

definition
  "Γ ≡ map_of (map (λi. (((''sumarr'', i), com (sumarr i))) [0..<2]) ++
  map_of (map (λi. (((''lock'', i), com (lock i))) [0..<2]) ++
  map_of (map (λi. (((''unlock'', i), com (unlock i))) [0..<2])"

definition
  "Θ ≡ map_of (map (λi. (((''sumarr'', i), [ann (sumarr i)])) [0..<2])
++
  map_of (map (λi. (((''lock'', i), [ann (lock i)])) [0..<2]) ++
  map_of (map (λi. (((''unlock'', i), [ann (unlock i)])) [0..<2])"

declare [[goals_limit = 10]]

lemma [simp]:
  "local_sum [] = 0"
  by (simp add: local_sum_def array_nat_sum_def)

lemma MAXSUM_le_plus:
  "x < MAXSUM ⟹ MAXSUM ≤ MAXSUM + x"
  unfolding MAXSUM_def
  apply (rule word_le_plus[rotated], assumption)
  apply clarsimp
 done

lemma local_sum_Suc:
  "⟦n < length arr; local_sum (take n arr) + arr ! n < MAXSUM;
    arr ! n < MAXSUM⟧ ⟹
    local_sum (take n arr) + arr ! n =
      local_sum (take (Suc n) arr)"
  apply (subst take_Suc_conv_app_nth)
   apply clarsimp
  apply (clarsimp simp: local_sum_def array_nat_sum_def )
   apply (subst (asm) min_def, clarsimp split: if_splits)
   apply (clarsimp simp: MAXSUM_le_plus word_not_le[symmetric])
  apply (subst min_absorb2)
   apply (subst of_nat_mono_maybe_le[where 'a=32])
```

```
    apply (clarsimp simp: MAXSUM_def)
   apply (clarsimp simp: MAXSUM_def)
   apply unat_arith
  apply (clarsimp simp: MAXSUM_def)
  apply unat_arith
 apply clarsimp
done

lemma local_sum_MAXSUM:
 "k < length arr ⟹ MAXSUM ≤ arr ! k ⟹ local_sum arr = MAXSUM"
 apply (clarsimp simp: local_sum_def array_nat_sum_def)
 apply (rule word_unat.Rep_inverse')
 apply (rule min_absorb1[symmetric])
 apply (subst (asm) word_le_nat_alt)
 apply (rule le_trans[rotated])
  apply (rule elem_le_sum_list)
  apply simp
 apply clarsimp
done

lemma local_sum_MAXSUM':
  ‹local_sum arr = MAXSUM›
 if ‹k < length arr›
   ‹MAXSUM ≤ local_sum (take k arr) + arr ! k›
   ‹local_sum (take k arr) ≤ MAXSUM›
   ‹arr ! k ≤ MAXSUM›
proof -
 define vs u ws where ‹vs = take k arr› ‹u = arr ! k› ‹ws = drop (Suc
k) arr›
 with ‹k < length arr› have *: ‹arr = vs @ u # ws›
   and **: ‹take k arr = vs› ‹arr ! k = u›
   by (simp_all add: id_take_nth_drop)
 from that show ?thesis
   apply (simp add: **)
   apply (simp add: *)
   apply (simp add: local_sum_def array_nat_sum_def ac_simps)
   apply (rule word_unat.Rep_inverse')
   apply (rule min_absorb1[symmetric])
   apply (subst (asm) word_le_nat_alt)
   apply (subst (asm) unat_plus_simple[THEN iffD1])
    apply (rule word_add_le_mono2[where i=0, simplified])
    apply (clarsimp simp: MAXSUM_def)
    apply unat_arith
   apply (rule le_trans, assumption)
   apply (rule add_mono)
    apply simp_all
   apply (subst le_unat_uoi)
   apply (rule min.cobounded1)
    apply simp
```

```
    done
qed

lemma word_min_0[simp]:
 "min (x::'a::len word) 0 = 0"
 "min 0 (x::'a::len word) = 0"
 by (simp add:min_def)+

ML ‹fun TRY' tac i = TRY (tac i)›


lemma imp_disjL_context':
  "((P ⟶ R) ∧ (Q ⟶ R)) = ((P ⟶ R) ∧ (¬P ∧ Q ⟶ R))"
by auto

lemma map_of_prod_1[simp]:
  "i < n ⟹
    map_of (map (λi. ((p, i), g i)) [0..<n])
       (p, i) = Some (g i)"
  apply (rule map_of_is_SomeI)
   apply (clarsimp simp: distinct_map o_def)
   apply (meson inj_onI prod.inject)
  apply clarsimp
  done

lemma map_of_prod_2[simp]:
  "i < n ⟹ p ≠ q ⟹
    (m ++
    map_of (map (λi. ((p, i), g i)) [0..<n]))
       (q, i) = m (q, i)"
  apply (rule map_add_dom_app_simps)
  apply (subst dom_map_of_conv_image_fst)
  apply clarsimp
  done

lemma sumarr_proc_simp[unfolded oghoare_simps]:
 "n < 2 ⟹ Γ (''sumarr'',n) = Some (com (sumarr n))"
 "n < 2 ⟹ Θ (''sumarr'',n) = Some ([ann (sumarr n)])"
 "n < 2 ⟹ Γ (''lock'',n) = Some (com (lock n))"
 "n < 2 ⟹ Θ (''lock'',n) = Some ([ann (lock n)])"
 "n < 2 ⟹ Γ (''unlock'',n) = Some (com (unlock n))"
 "n < 2 ⟹ Θ (''unlock'',n) = Some ([ann (unlock n)])"
 "[ann (sumarr n)]!0 = ann (sumarr n)"
 "[ann (lock n)]!0 = ann (lock n)"
 "[ann (unlock n)]!0 = ann (unlock n)"
 by (simp add: Γ_def Θ_def)+

lemmas sumarr_proc_simp_unfolded = sumarr_proc_simp[unfolded sumarr_def
unlock_def lock_def oghoare_simps]
```

```
lemma oghoare_sumarr:
  ‹Γ, Θ |⊢/F sumarr i {| ˝local_postcond i|}, {|False|}› if ‹i < 2›
proof -
  from that have ‹i = 0 ∨ i = 1› by auto
  note sumarr_proc_simp_unfolded[proc_simp add]
  show ?thesis
  using that apply -
  unfolding sumarr_def unlock_def lock_def
    ann_call_def call_def block_def
  apply simp
  apply oghoare
  unfolding tarr_inv_def array_length_def array_nth_def array_in_bound_def
    sumarr_in_lock1_def sumarr_in_lock2_def
  apply (tactic "PARALLEL_ALLGOALS ((TRY' o SOLVED')
          (clarsimp_tac (@{context} addsimps
                @{thms local_postcond_def global_sum_def ex_in_conv[symmetric]})

          THEN_ALL_NEW fast_force_tac
             (@{context} addSDs @{thms less_2_cases}
                         addIs @{thms local_sum_Suc unat_mono}
                        )
          ))")
  using ‹i = 0 ∨ i = 1› apply rule
      apply (clarsimp simp add: bit_simps even_or_iff)
      apply (clarsimp simp add: bit_simps even_or_iff)
   apply clarsimp
   apply (rule conjI)
    apply (fastforce intro!: local_sum_Suc unat_mono)
   apply (subst imp_disjL_context')
   apply (rule conjI)
    apply clarsimp
    apply (erule local_sum_MAXSUM[rotated])
    apply unat_arith
   apply (clarsimp simp: not_le)
   apply (erule (1) local_sum_MAXSUM'[rotated] ; unat_arith)
  apply clarsimp
    apply unat_arith
  apply (fact that)
 done
qed

lemma less_than_two_2[simp]:
  "i < 2 ⟹ Suc 0 - i < 2"
  by arith

lemma oghoare_call_sumarr:
notes sumarr_proc_simp[proc_simp add]
shows
```

```
  "i < 2 ⟹
  Γ, Θ ⊩/F call_sumarr i {´local_postcond i}, {False}"
  unfolding call_sumarr_def ann_call_def call_def block_def
    tarr_inv_def
  apply oghoare

  apply (clarsimp; fail | ((simp only: pre.simps)?, rule  oghoare_sumarr))+
  apply (clarsimp simp: sumarr_def tarr_inv_def)
  apply (clarsimp simp: local_postcond_def; fail)+
  done

lemma less_than_two_inv[simp]:
  "i < 2 ⟹ j < 2 ⟹ i ≠ j ⟹ Suc 0 - i = j"
  by simp

lemma inter_aux_call_sumarr [simplified]:
  notes sumarr_proc_simp_unfolded [proc_simp add]
     com.simps [oghoare_simps add]
     bit_simps [simp]
  shows
  "i < 2 ⟹ j < 2 ⟹ i ≠ j ⟹ interfree_aux Γ Θ
     F (com (call_sumarr i), (ann (call_sumarr i), {´local_postcond i},
{False}),
        com (call_sumarr j), ann (call_sumarr j))"
  unfolding call_sumarr_def ann_call_def call_def block_def
    tarr_inv_def sumarr_def lock_def unlock_def
  apply oghoare_interfree_aux
  unfolding
    tarr_inv_def local_postcond_def sumarr_in_lock1_def
    sumarr_in_lock2_def
  by (tactic "PARALLEL_ALLGOALS (
                 TRY' (remove_single_Bound_mem @{context}) THEN'
                 (TRY' o SOLVED')
                 (clarsimp_tac @{context} THEN_ALL_NEW
                   fast_force_tac (@{context} addSDs @{thms less_2_cases})
                 ))")

lemma pre_call_sumarr:
  "i < 2 ⟹ precond x ⟹ x ∈ pre (ann (call_sumarr i))"
  unfolding precond_def call_sumarr_def ann_call_def
  by (fastforce dest: less_2_cases simp: array_length_def)

lemma post_call_sumarr:
  "local_postcond x 0 ⟹ local_postcond x 1 ⟹ postcond x"
  unfolding postcond_def local_postcond_def
  by (fastforce dest: less_2_cases split: if_splits)

lemma sumarr_correct:
  "Γ, Θ ⊨/F {´precond}
```

```
    COBEGIN
      SCHEME [0 ≤ m < 2]
      call_sumarr m
      {| ´local_postcond m|},{|False|}
    COEND
   {| ´postcond|}, {|False|}"
  apply oghoare
      apply (fastforce simp: pre_call_sumarr)
     apply (rule oghoare_call_sumarr, simp)
    apply (clarsimp simp: post_call_sumarr)
   apply (simp add: inter_aux_call_sumarr)
  apply clarsimp
 done

end
```