COMPLX: a Verification Framework for Concurrent Imperative Programs

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

April 10, 2026

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"
  ⟨proof⟩
```

```
lemma snd_guaranteeStripPair: "snd (guaranteeStripPair f g) = g"
  ⟨proof⟩
```

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"
   ⟨proof⟩
```

```
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) body ⇒ ('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 ⊢(c₁,s) → (c₁',s')
       ⟹
       Γ ⊢(Seq c₁ c₂,s) → (Seq c₁' c₂, s')"
| SeqSkip: "Γ ⊢(Seq Skip c₂,s) → (c₂, s)"
| SeqThrow: "Γ ⊢(Seq Throw c₂,Normal s) → (Throw, Normal s)"

| CondTrue:  "s∈b ⟹ Γ ⊢(Cond b c₁ c₂,Normal s) → (c₁,Normal s)"
| CondFalse: "s∉b ⟹ Γ ⊢(Cond b c₁ c₂,Normal s) → (c₂,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 ⊢(c₁,s) → (c₁',s')⟧
         ⟹
         Γ ⊢(Catch c₁ c₂,s) → (Catch c₁' c₂,s')"

| CatchSkip: "Γ ⊢(Catch Skip c₂,s) → (Skip,s)"
| CatchThrow: "Γ ⊢(Catch Throw c₂,Normal s) → (c₂,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 c₁) c₂) (Normal s) = (s ∉ b)"
| "blocked Γ (Catch (Await b c₁) c₂) (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"
```

⟨*proof*⟩

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"
⟨*proof*⟩

lemma no_step_final':
  assumes step: "Γ ⊢cfg → cfg'"
  shows "final cfg ⟹ P"
⟨*proof*⟩

```
lemma no_steps_final:
"Γ ⊢ v →* w ⟹ final v ⟹ w = v"
  ⟨proof⟩

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

lemma step_Stuck:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "⋀f. s=Stuck ⟹ s'=Stuck"
⟨proof⟩

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')"
⟨proof⟩


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')"
⟨proof⟩


lemma steps_Fault: "Γ ⊢ (c, Fault f) →* (Skip, Fault f)"
⟨proof⟩

lemma steps_Stuck: "Γ ⊢ (c, Stuck) →* (Skip, Stuck)"
⟨proof⟩

lemma step_Fault_prop:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "⋀f. s=Fault f ⟹ s'=Fault f"
⟨proof⟩

lemma step_Stuck_prop:
  assumes step: "Γ ⊢ (c, s) → (c', s')"
  shows "s=Stuck ⟹ s'=Stuck"
⟨proof⟩

lemma steps_Fault_prop:
  assumes step: "Γ ⊢ (c, s) →* (c', s')"
  shows "s=Fault f ⟹ s'=Fault f"
⟨proof⟩
```

```
lemma steps_Stuck_prop:
  assumes step: "Γ ⊢ (c, s) →* (c', s')"
  shows "s=Stuck ⟹ s'=Stuck"
⟨proof⟩

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)"
   ⟨*proof*⟩

lemma "¬pre_par c ⟹ pre c ∈ pre_set c"
  ⟨*proof*⟩

lemma pre_set:
  "pre c = ⋂ (pre_set c)"
  ⟨*proof*⟩

lemma pre_imp_pre_set:
  "s ∈ pre c ⟹ a ∈ pre_set c ⟹ s ∈ a"
  ⟨*proof*⟩

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"
  ⟨proof⟩
```

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: Γ⊨/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"
```
⟨*proof*⟩

lemma `atomicsR_Throw[simp]:`
```
  "atomicsR Γ Θ a Throw r = False"
```
⟨*proof*⟩

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"

⟨*proof*⟩

lemma Normal_pre[rule_format, OF _ refl] :
"$\Gamma \vdash$ (p, s) $\rightarrow$ (p', s') $\Longrightarrow$
$\forall$u. s' = Normal u $\longrightarrow$ ($\exists$v. s = Normal v)"
⟨*proof*⟩

lemma Normal_pre_star[rule_format, OF _ refl] :
"$\Gamma \vdash$ cfg_1 $\rightarrow^*$ cfg_2 $\Longrightarrow$ $\forall$p' t. cfg_2 = (p', Normal t) $\longrightarrow$
($\exists$p s. cfg_1 = (p, Normal s))"
⟨*proof*⟩

lemma Seq_decomp_Skip[rule_format, OF _ refl] :
"$\Gamma \vdash$ (p, s) $\rightarrow$ (p', s') $\Longrightarrow$
$\forall p_2$. p = Seq Skip p_2 $\longrightarrow$ s' = s $\wedge$ p' = p_2"
⟨*proof*⟩

lemma Seq_decomp_Throw[rule_format, OF _ refl, OF _ refl] :
"$\Gamma \vdash$ (p, s) $\rightarrow$ (p', s') $\Longrightarrow$
$\forall p_2$ z. s = Normal z $\longrightarrow$ p = Seq Throw p_2 $\longrightarrow$ s' = s $\wedge$ p' = Throw"
⟨*proof*⟩

lemma Throw_star[rule_format, OF _ refl] :
"$\Gamma \vdash$ cfg_1 $\rightarrow^*$ cfg_2 $\Longrightarrow$ $\forall$s. cfg_1 = (Throw, Normal s) $\longrightarrow$
cfg_2 = cfg_1"
⟨*proof*⟩

lemma Seq_decomp[rule_format, OF _ refl] :
"$\Gamma \vdash$ (p, s) $\rightarrow$ (p', s') $\Longrightarrow$
$\forall p_1$ p_2. p = Seq p_1 p_2 $\longrightarrow$ $p_1 \neq$ Skip $\longrightarrow$ $p_1 \neq$ Throw $\longrightarrow$
($\exists p_1$'. $\Gamma \vdash$ (p_1, s) $\rightarrow$ (p_1', s') $\wedge$ p' = Seq p_1' p_2)"
⟨*proof*⟩

lemma Seq_decomp_relpow:
"$\Gamma \vdash$ (Seq p_1 p_2, Normal s) $\rightarrow^n$n (p', Normal s') $\Longrightarrow$
final (p', Normal s') $\Longrightarrow$
($\exists$n1<n. $\Gamma \vdash$ (p_1, Normal s) $\rightarrow^n$n1 (Throw, Normal s')) $\wedge$ p'=Throw $\vee$
($\exists$t n1 n2. $\Gamma \vdash$ (p_1, Normal s) $\rightarrow^n$n1 (Skip, Normal t) $\wedge$ n1 < n $\wedge$ n2 <
n $\wedge$ $\Gamma \vdash$ (p_2, Normal t) $\rightarrow^n$n2 (p', Normal s'))"
⟨*proof*⟩

lemma Seq_decomp_star:
"$\Gamma \vdash$ (Seq p_1 p_2, Normal s) $\rightarrow^*$ (p', Normal s') $\Longrightarrow$ final (p', Normal s')
$\Longrightarrow$
$\Gamma \vdash$ (p_1, Normal s) $\rightarrow^*$ (Throw, Normal s') $\wedge$ p'=Throw $\vee$
($\exists$t. $\Gamma \vdash$ (p_1, Normal s) $\rightarrow^*$ (Skip, Normal t) $\wedge$ $\Gamma \vdash$ (p_2, Normal t) $\rightarrow^*$

```
(p', Normal s'))"
  ⟨proof⟩
```

lemma `Seq_decomp_relpowp_Fault:`

```
"Γ ⊢ (Seq p₁ p₂, Normal s) →ⁿn (Skip, Fault f) ⟹
 (∃n1. Γ ⊢ (p₁, Normal s) →ⁿn1 (Skip, Fault f)) ∨
 (∃t n1 n2. Γ ⊢ (p₁, Normal s) →ⁿn1 (Skip, Normal t) ∧ n1 < n ∧ n2 <
n  ∧ Γ ⊢ (p₂, Normal t) →ⁿn2 (Skip, Fault f))"
  ⟨proof⟩
```

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)))"
  ⟨proof⟩
```

lemma `Seq_decomp_relpowp_Stuck:`

```
"Γ ⊢ (Seq p₁ p₂, Normal s) →ⁿn (Skip, Stuck) ⟹
 (∃n1. Γ ⊢ (p₁, Normal s) →ⁿn1 (Skip, Stuck)) ∨
 (∃t n1 n2. Γ ⊢ (p₁, Normal s) →ⁿn1 (Skip, Normal t) ∧ n1 < n ∧ n2 <
n  ∧ Γ ⊢ (p₂, Normal t) →ⁿn2 (Skip, Stuck))"
  ⟨proof⟩
```

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)))"
  ⟨proof⟩
```

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))"
  ⟨proof⟩
```

```
lemma Catch_decomp_Skip[rule_format, OF _ refl] :
"Γ ⊢ (p, s) → (p', s') ⟹
 ∀p₂. p = Catch Skip p₂ ⟶ s' = s ∧ p' = Skip"
  ⟨proof⟩

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)"
  ⟨proof⟩

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)))"
  ⟨proof⟩

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)))"
  ⟨proof⟩

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')"
 ⟨proof⟩

lemma oghoare_Skip[rule_format, OF _ refl]:
```

```
"Γ, Θ ⊢/F P c Q,A ⟹ c = Skip ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ Q)"
```
⟨*proof*⟩

lemma oghoare_Throw[rule_format, OF _ refl]:
```
"Γ, Θ ⊢/F P c Q,A ⟹ c = Throw ⟶
 (∃P'. P = AnnExpr P' ∧ P' ⊆ A)"
```
⟨*proof*⟩

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})"
```
⟨*proof*⟩

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)})"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma oghoare_Catch[rule_format, OF _ refl]:

```
"Γ, Θ ⊢/F P x Q,A ⟹ ∀p1 p2. x = Catch p1 p2 ⟶
 (∃ P1 P2 P' Q' A'. P = AnnComp P1 P2 ∧ Γ, Θ ⊢/F P1 p1 Q', P' ∧ P' ⊆
pre P2 ∧
         Γ, Θ ⊢/F P2 p2 Q',A' ∧
          Q' ⊆ Q ∧ A' ⊆ A)"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma ann_matches_weaken[OF _ refl]:
```
" ann_matches Γ Θ X c ⟹ X = (weaken_pre P P') ⟹ ann_matches Γ Θ P
c"
```
⟨*proof*⟩

lemma oghoare_seq_imp_ann_matches:
```
 " Γ,Θ⊩/F P a c Q,A ⟹ ann_matches Γ Θ a c"
```
⟨*proof*⟩

lemma oghoare_imp_ann_matches:
```
 " Γ,Θ⊢/F a c Q,A ⟹ ann_matches Γ Θ a c"
```
⟨*proof*⟩

lemma SkipRule: `"P ⊆ Q ⟹ Γ, Θ ⊢/F (AnnExpr P) Skip Q, A"`
⟨*proof*⟩

lemma ThrowRule: `"P ⊆ A ⟹ Γ, Θ ⊢/F (AnnExpr P) Throw Q, A"`
⟨*proof*⟩

lemma BasicRule:
```
"P ⊆ {s. (f s) ∈ Q} ⟹ Γ, Θ ⊢/F (AnnExpr P) (Basic
f) Q, A"
```
⟨*proof*⟩

lemma SpecRule:
```
  "P ⊆ {s. (∀t. (s, t) ∈ r ⟶ t ∈ Q) ∧ (∃t. (s, t) ∈ r)}
    ⟹ Γ, Θ ⊢/F (AnnExpr P) (Spec r) Q, A"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
  ⟨proof⟩
```

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"
 ⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

lemma Skip_normal_steps_end:

```
"Γ ⊢ (Skip, Normal s) →* (c, s') ⟹ c = Skip ∧ s' = Normal s"
  ⟨proof⟩
```

lemma Throw_normal_steps_end:

```
"Γ ⊢ (Throw, Normal s) →* (c, s') ⟹ c = Throw ∧ s' = Normal s"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

lemma valid_weaken_pre:

```
 "Γ⊨/F P c Q,A ⟹
  P' ⊆ P ⟹ Γ⊨/F P' c Q,A"
 ⟨proof⟩
```

lemma valid_strengthen_post:

```
"Γ⊨/F P c Q,A ⟹
  Q ⊆ Q' ⟹ Γ⊨/F P c Q',A"
⟨proof⟩
```

lemma valid_strengthen_abr:

```
"Γ⊨/F P c Q,A ⟹
  A ⊆ A' ⟹ Γ⊨/F P c Q,A'"
⟨proof⟩
```

lemma oghoare_while_valid:

```
"Γ⊨/F P c i,A ⟹
  i ∩ b ⊆ P ⟹
  i ∩ - b ⊆ Q ⟹
  Γ⊨/F i While b c Q,A"
  ⟨proof⟩
```

lemma oghoare_catch_valid:

```
"Γ⊨/F P1 c1 Q,P2 ⟹
  Γ⊨/F P2 c2 Q,A ⟹
    Γ⊨/F P1 Catch c1 c2 Q,A"
  ⟨proof⟩
```

lemma ann_matches_imp_assertionsR:

```
  "ann_matches Γ Θ a c ⟹ ¬ pre_par a ⟹
    assertionsR Γ Θ Q A a c (pre a)"
  ⟨proof⟩
```

lemma ann_matches_imp_assertionsR':

```
  "ann_matches Γ Θ a c ⟹ a' ∈ pre_set a ⟹
    assertionsR Γ Θ Q A a c a'"
 ⟨proof⟩
```

lemma rtranclp_conjD:

```
  "(λx1 x2. r1 x1 x2 ∧ r2 x1 x2)** x1 x2 ⟹
  r1** x1 x2 ∧ r2** x1 x2"
⟨proof⟩
```

lemma rtranclp_mono' :

```
"r** a b ⟹ r ≤ s ⟹ s** a b"
⟨proof⟩
```

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))"
```
⟨*proof*⟩

lemma oghoare_atom_com_sound:

```
  "Γ, Θ ⊩/F P a c Q,A ⟹ atom_com c ⟹ Γ ⊨/F P c Q, A"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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*⟩

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))"
```
⟨*proof*⟩

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)"
```

⟨*proof*⟩

```
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"
```
⟨*proof*⟩

```
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'"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

```
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'"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

```
lemma oghoare_soundness:
  "(Γ, Θ ⊢/F P c Q,A ⟶ Γ ⊨/F (pre P) c Q, A) ∧
   (Γ, Θ ⊩/FP' P c Q,A ⟶ Γ ⊨/F P' c Q, A)"
```
⟨*proof*⟩

```
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*⟩

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 P₁ c₁ Q,A;
    Γ, Θ ⊩/F C2 P₂ c₂ Q,A ⟧
   ⟹ Γ, Θ ⊩/F {s. (s∈b ⟶ s∈C1) ∧ (s∉b ⟶ s∈C2)} (AnnBin r P₁ P₂)
                 (Cond b c₁ c₂) Q,A"
```
  ⟨*proof*⟩

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"
```
  ⟨*proof*⟩

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"
```
  ⟨*proof*⟩

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

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"
  ⟨proof⟩
```

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 [] []"`

```
 ⟨proof⟩
```

lemma `MapAnnList:`

```
"⟦ Γ, Θ ⊢/F P c Q, A;
   Γ, Θ  [⊩]/F Ps Ts ⟧
 ⟹ Γ, Θ [⊩]/F ((P, Q, A)#Ps) (c#Ts)"
  ⟨proof⟩
```

```
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])"
 ⟨proof⟩

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))"
  ⟨proof⟩

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"
  ⟨proof⟩
```

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'"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma Skip_inter_right:
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) Skip q, q ⟹ interfree_aux_right Γ Θ
F (q, Skip, AnnExpr r)"
  ⟨proof⟩

lemma Throw_inter_right:
```

```
  "Γ, Θ ⊢/F (AnnExpr (q ∩ r)) Throw q, q ⟹ interfree_aux_right Γ Θ
F (q, Throw, AnnExpr r)"
  ⟨proof⟩
```

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)"
  ⟨proof⟩
```

lemma valid_Await:

```
 "atom_com c ⟹ Γ⊨/F (P ∩ b) c Q,A ⟹ Γ⊨/F P Await b c Q,A"
  ⟨proof⟩
```

lemma atomcom_imp_not_prepare:

```
 "ann_matches Γ Θ a c ⟹ atom_com c ⟹
   ¬ pre_par a"
 ⟨proof⟩
```

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)"
  ⟨proof⟩
```

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)"
  ⟨proof⟩
```

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)"
  ⟨proof⟩
```

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)"
 ⟨proof⟩
```

lemma Parallel_inter_right_empty:

```
  "interfree_aux_right Γ Θ F (q, Parallel [], AnnPar [])"
 ⟨proof⟩
```

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))"
  ⟨proof⟩
```

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]))"
  ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
 ⟨proof⟩
```

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

```
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)"
⟨proof⟩

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)"
  ⟨proof⟩

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)"
⟨proof⟩

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)"
 ⟨proof⟩

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)"
⟨proof⟩

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)"
⟨proof⟩
```

```
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')"
  ⟨proof⟩

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)"
  ⟨proof⟩

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)"
  ⟨proof⟩

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, [], [])"
⟨proof⟩

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)"
  ⟨proof⟩

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])"
⟨proof⟩

lemma interfree_Empty: "interfree Γ Θ F [] []"
⟨proof⟩

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)"
```

⟨*proof*⟩

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])"
```
⟨*proof*⟩

lemma list_lemmas:
```
"length []=0" "length (x#xs) = Suc(length xs)"
    "(x#xs) ! 0 = x" "(x#xs) ! Suc n = xs ! n"
```
⟨*proof*⟩

lemma le_Suc_eq_insert: `"{i. i <Suc n} = insert n {i. i< n}"`
⟨*proof*⟩

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*⟩

named_theorems proc_simp
named_theorems oghoare_simps

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

⟨*ML*⟩

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»"
```

⟨*ML*⟩

```
syntax
  "_fst" :: "'a × 'b ⇒ 'a"  (‹_,› [60] 61)
  "_snd" :: "'a × 'b ⇒ 'b"  (‹_.› [60] 61)
```

⟨*ML*⟩

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)
```

⟨*ML*⟩

```
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*⟩

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⦄"
```

⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma Example1':
notes Example1_proc_simp[proc_simp]
shows
```
 "Γ', Θ' |⊢/F Example1' {|∀i < 2. ´ex1_a i = 0|}, {|False|}"
```
⟨*proof*⟩

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)"
  ⟨proof⟩

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)"
 ⟨proof⟩

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⦄"
  ⟨proof⟩

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"
  ⟨proof⟩


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|}"
  ⟨proof⟩

lemma guard_example:
  "Γ, Θ |⊢/{42,66}
  {|True|} (42, {|´x=0|}),
   (66, {|´y=0|}) ⟼ {|´x = 0|}
   ´y := 0;;
   {|True|} ´x := 0
  {| ´x = 0|}, {|False|}"
  ⟨proof⟩
```

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⦄
"
  ⟨proof⟩
```

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⦄
 "
  ⟨proof⟩
```

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⦄"
```

⟨*proof*⟩

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"
```
  ⟨*proof*⟩

lemma `MAXSUM_le_plus:`
```
  "x < MAXSUM ⟹ MAXSUM ≤ MAXSUM + x"
```
  ⟨*proof*⟩

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)"
```
  ⟨*proof*⟩

lemma `local_sum_MAXSUM:`
```
  "k < length arr ⟹ MAXSUM ≤ arr ! k ⟹ local_sum arr = MAXSUM"
```
  ⟨*proof*⟩

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*⟩

lemma `word_min_0[simp]:`
```
 "min (x::'a::len word) 0 = 0"
 "min 0 (x::'a::len word) = 0"
```
 ⟨*proof*⟩

⟨*ML*⟩

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

⟨proof⟩

lemma map_of_prod_1[simp]:
```
  "i < n ⟹
    map_of (map (λi. ((p, i), g i)) [0..<n])
       (p, i) = Some (g i)"
```
  ⟨proof⟩

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)"
```
  ⟨proof⟩

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)"
```
 ⟨proof⟩

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⟩

lemma less_than_two_2[simp]:
```
  "i < 2 ⟹ Suc 0 - i < 2"
```
  ⟨proof⟩

lemma oghoare_call_sumarr:
notes sumarr_proc_simp[proc_simp add]
shows
  "i < 2 ⟹
  Γ, Θ |⊢$_{/F}$ call_sumarr i ⦃´local_postcond i⦄, ⦃False⦄"
  ⟨proof⟩

lemma less_than_two_inv[simp]:
```
  "i < 2 ⟹ j < 2 ⟹ i ≠ j ⟹ Suc 0 - i = j"
```
  ⟨proof⟩

```
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))"
  ⟨proof⟩

lemma pre_call_sumarr:
  "i < 2 ⟹ precond x ⟹ x ∈ pre (ann (call_sumarr i))"
  ⟨proof⟩

lemma post_call_sumarr:
  "local_postcond x 0 ⟹ local_postcond x 1 ⟹ postcond x"
  ⟨proof⟩

lemma sumarr_correct:
  "Γ, Θ |⊩/F ⦃´precond⦄
    COBEGIN
      SCHEME [0 ≤ m < 2]
      call_sumarr m
      ⦃´local_postcond m⦄,⦃False⦄
    COEND
   ⦃´postcond⦄, ⦃False⦄"
  ⟨proof⟩

end
```