Rely-Guarantee Extensions and Locks

Robert J. Colvin, Scott Heiner, Peter Höfner, Roger C. Su

January 9, 2026

Abstract

We enhance rely-guarantee verification in Isabelle/HOL by extending the 2003 built-in library with flexible syntax, data-invariant support, and new tactics. We demonstrate our enhanced library by applying it to the examples attached to the original library. We also apply our library to three queue locks: the Abstract Queue Lock, the Ticket Lock, and the Circular Buffer Lock.

Contents

1 Introduction

The content of this entry has been presented as [1]. The original built-in library is [2].

2 Rely-Guarantee (RG) Syntax Extensions

The core extensions to the built-in RG library: improved syntax of RG sentences in the quintuple- and keyword-styles, with data-invariants.

Also: subgoal-generating methods for RG inference-rules that work with the structured proof-language, Isar.

```
theory RG_Syntax_Extensions

imports
  "HOL-Hoare_Parallel.RG_Syntax"
  "HOL-Eisbach.Eisbach"

begin
```

We begin with some basic notions that are used later on.

Notation for forward function-composition: defined in the built-in `Fun.thy` but disabled at the end of that theory. This operator is useful for modelling atomic primitives such as Swap and Fetch-And-Increment, and also useful when coupling concrete- and auxiliary-variable instructions.

```
notation fcomp (infixl "∘>" 60)

lemmas definitions [simp] =
  stable_def Pre_def Rely_def Guar_def Post_def Com_def
```

In applications, guarantee-relations often stipulates that Thread `i` should "preserve the rely-relations of all other threads". This pattern is supported by the following higher-order function, where `j` ranges through all the threads that are not `i`.

```
abbreviation for_others :: "('index ⇒ 'state rel) ⇒ 'index ⇒ 'state rel" where
  "for_others R i ≡ ⋂ j ∈ -{i}. R j"
```

Relies and guarantees often state that certain variables remain unchanged. We support this pattern with the following syntactic sugars.

```
abbreviation record_id :: "('record ⇒ 'field) ⇒ 'record rel"
  ("id'(_')" [75] 74) where
  "id(c) ≡ ⦃ ªc = ºc ⦄"

abbreviation record_ids :: "('record ⇒ 'field) set ⇒ 'record rel"
  ("ids'(_')" [75] 74) where
  "ids(cs) ≡ ⋂ c ∈ cs. id(c)"

abbreviation record_id_indexed ::
  "('record ⇒ 'index ⇒ 'field) ⇒ 'index ⇒ 'record rel"
  ("id'( _ @ _ ')") where
  "id(c @ self) ≡ ⦃ ºc self = ªc self ⦄"

abbreviation record_ids_indexed ::
  "('record ⇒ 'index ⇒ 'field) set ⇒ 'index ⇒ 'record rel"
  ("ids'( _ @ _ ')") where
  "ids(cs @ self) ≡ ⋂ c ∈ cs. id(c @ self)"
```

The following simple method performs an optional simplification-step, and then tries to apply one of the RG rules, before attempting to discharge each subgoal using `force`. This method works well on simple RG sentences.

```
method method_rg_try_each =
  (clarsimp | simp)?,
  ( rule Basic | rule Seq | rule Cond | rule While
  | rule Await | rule Conseq | rule Parallel);
   force+
```

2.1 Lifting of Invariants

There are different ways to combine the invariant with the rely or guarantee, as long as the invariant is preserved. Here, a rely- or guarantee-relation `R` is combined with the invariant `I` into `{(s, s'). (s ∈ I ⟶ s' ∈ I) ∧ R}`.

```
definition pred_to_rel :: "'a set ⇒ 'a rel" where
  "pred_to_rel P ≡ {(s,s') . s ∈ P ⟶ s' ∈ P}"

definition invar_and_guar :: "'a set ⇒ 'a rel ⇒ 'a rel" where
  "invar_and_guar I G ≡ G ∩ pred_to_rel I"

lemmas simp_defs [simp] = pred_to_rel_def invar_and_guar_def
```

2.2 RG Sentences

The quintuple-style of RG sentences.

```
abbreviation rg_quint ::
  "'a set ⇒ 'a rel ⇒ 'a com ⇒ 'a rel ⇒ 'a set ⇒ bool"
  ("{_,_} _ {_,_}") where
  "{P, R} C {G, Q} ≡ ⊢ C sat [P, R, G, Q]"
```

Quintuples with invariants.

```
abbreviation rg_quint_invar ::
```

```
  "'a set ⇒ 'a rel ⇒ 'a com ⇒ 'a set ⇒ 'a rel ⇒ 'a set ⇒ bool"
  ("{_,_} _ // _ {_,_}") where
  "{P, R} C // I {G, Q} ≡ ⊢ C sat [
    P ∩ I,
    R ∩ pred_to_rel I,
    invar_and_guar I G,
    Q ∩ I]"
```

The keyword-style of RG sentences.

```
abbreviation rg_keyword ::
  "'a rel ⇒ 'a rel ⇒ 'a set ⇒ 'a com ⇒ 'a set ⇒ bool"
  ("rely:_ guar:_ code: {_} _ {_}") where
  "rg_keyword R G P C Q ≡ ⊢ C sat [P, R, G, Q]"
```

Keyword-style RG sentences with invariants.

```
abbreviation rg_keyword_invar ::
  "'a rel ⇒ 'a rel ⇒ 'a set ⇒ 'a set ⇒ 'a com ⇒ 'a set ⇒ bool"
  ("rely:_ guar:_ inv:_ code: {_} _ {_}") where
  "rg_keyword_invar R G I P C Q ≡ ⊢ C sat [
    P ∩ I,
    R ∩ pred_to_rel I,
    invar_and_guar I G,
    Q ∩ I]"
```

2.3 RG Subgoal-Generating Methods

As in Floyd-Hoare logic, in RG we can strengthen (make smaller) the precondition and weaken (make larger) the postcondition without affecting the validity of an RG sentence.

```
theorem strengthen_pre:
  assumes "P' ⊆ P"
      and "⊢ c sat [P,  R, G, Q]"
    shows "⊢ c sat [P', R, G, Q]"
  using assms Conseq by blast

theorem weaken_post:
  assumes "Q ⊆ Q'"
      and "⊢ c sat [P, R, G, Q ]"
    shows "⊢ c sat [P, R, G, Q']"
  using assms Conseq by blast
```

We then develop subgoal-generating methods for various instruction types and patterns, to be used in conjunction with the Isar proof-language.

2.3.1 Basic

A `Basic` instruction wraps a state-transformation function.

```
theorem rg_basic_named[intro]:
  assumes "stable P R"
      and "stable Q R"
      and "∀s. s ∈ P ⟶ (s, s) ∈ G"
      and "∀s. s ∈ P ⟶ (s, f s) ∈ G"
      and "P ⊆ {| ´f ∈ Q |}"
    shows "{P, R} Basic f {G, Q}"
  using assms apply -
  by (rule Basic; fastforce)
```

```
method method_basic =
  rule rg_basic_named,
  goal_cases stable_pre stable_post guar_id establish_guar establish_post
```

The *skip* command is a `Basic` instruction whose function is the identity.

```
theorem rg_skip_named:
  assumes "stable P R"
      and "stable Q R"
      and "Id ⊆ G"
      and "P ⊆ Q"
    shows "{P, R} SKIP {G, Q}"
  using assms by force

method method_skip =
  rule rg_skip_named,
  goal_cases stab_pre stab_post guar_id est_post
```

An alternative version with an invariant subgoal.

```
theorem rg_basic_inv[intro]:
  assumes "stable (P ∩ I) (R ∩ pred_to_rel I)"
      and "stable (Q ∩ I) (R ∩ pred_to_rel I)"
      and "∀s. s ∈ P ∩ I ⟶ (s, s) ∈ G"
      and "∀s. s ∈ P ∩ I ⟶ f s ∈ I"
      and "∀s. s ∈ P ∩ I ⟶ f s ∈ Q"
      and "∀s. s ∈ P ∩ I ⟶ (s, f s) ∈ G"
    shows "⊢ (Basic f) sat [
      P ∩ I,
      R ∩ pred_to_rel I,
      invar_and_guar I G,
      Q ∩ I]"
  using assms apply -
  by (method_basic; fastforce)

method method_basic_inv = rule rg_basic_inv,
  goal_cases stab_pre stab_post id_guar est_inv est_post est_guar
```

2.3.2 Looping constructs

```
theorem rg_general_loop_named[intro]:
  assumes "stable P R"
      and "stable Q R"
      and "Id ⊆ G"
      and "P ∩ -b ⊆ Q"
      and "{P ∩ b, R} c {G, P}"
    shows "{P, R} While b c {G, Q}"
  using assms apply -
  by (rule While; fastforce)

method method_loop =
  rule rg_general_loop_named,
  goal_cases stable_pre stable_post id_guar loop_exit loop_body
```

A similar version but with the `loop_body` subgoal having a weakend precondition.

```
theorem rg_general_loop_no_guard[intro]:
  assumes "stable P R"
      and "stable Q R"
      and "Id ⊆ G"
      and "P ∩ -b ⊆ Q"
```

```
    and "{P, R} c {G, P}"
  shows "{P, R} While b c {G, Q}"
  apply(rule rg_general_loop_named)
  by (fastforce intro!: assms Int_lower1 intro: strengthen_pre)+

method method_loop_no_guard =
  rule rg_general_loop_no_guard,
  goal_cases stab_pre stab_post guar_id loop_exit loop_body
```

A *spinloop* is a loop with an empty body. Such a loop repeatedly checks a property, and is a key construct in mutual exclusion algorithms.

```
theorem rg_spinloop_named[intro]:
  assumes "stable P R"
      and "stable Q R"
      and "Id ⊆ G"
      and "P ∩ -b ⊆ Q"
    shows "{P, R} While b SKIP {G, Q}"
  using assms
  by (fastforce simp: rg_general_loop_no_guard rg_skip_named)


method method_spinloop =
  rule rg_spinloop_named,
  goal_cases stable_pre stable_post guar_id est_post

theorem rg_infinite_loop:
  assumes "stable P R"
      and "Id ⊆ G"
      and "{P, R} C {G, P}"
    shows "{P, R} While UNIV C {G, Q}"
proof -
  have "{P, R} While UNIV C {G, {}}"
    using assms by (fastforce simp: rg_general_loop_no_guard)
  thus ?thesis
    using weaken_post by fastforce
qed

method method_infinite_loop =
  rule rg_infinite_loop,
  goal_cases stable_pre guar_id loop_body,
  clarsimp+

theorem rg_infinite_loop_syntax:
  assumes "stable P R"
      and "Id ⊆ G"
      and "{P, R} C {G, P}"
    shows "{P, R} WHILE True DO C OD {G, Q}"
  using assms by (fastforce simp: rg_infinite_loop)

method method_infinite_loop_syntax =
  rule rg_infinite_loop_syntax,
  goal_cases stable_pre guar_id loop_body
```

A *repeat-loop* encodes the pattern where the loop body is executed before the first evaluation of the guard.

```
theorem rg_repeat_loop[intro]:
  assumes "stable P R"
      and "stable Q R"
```

```
      and "Id ⊆ G"
      and "P ∩ b ⊆ Q"
      and loop_body: "{P, R} C {G, P}"
    shows "{P, R} C ;; While (-b) C {G, Q}"
  using assms apply -
  apply (rule Seq)
   apply (force intro: loop_body)
  by (method_loop_no_guard; fastforce)

method method_repeat_loop =
  rule rg_repeat_loop,
  goal_cases stab_pre stab_post guar_id loop_exit loop_body
```

When reasoning about repeat-loops, we may need information from `P` to determine whether we reach the postcondition. In this case we can use the following form, which introduces a mid-state.

```
theorem rg_repeat_loop_mid[intro]:
  assumes stab_pre:  "stable (P ∩ M) R"
      and stab_post: "stable Q R"
      and guar_id:   "Id ⊆ G"
      and loop_exit: "P ∩ M ∩ b ⊆ Q"
      and loop_body: "{P, R} C {G, P ∩ M}"
    shows "{P, R} C ;; While (-b) C {G, Q}"
 using assms apply -
  apply (rule Seq)
   apply (fast intro: loop_body)
  by (method_loop_no_guard; fast intro: loop_body strengthen_pre)

method method_repeat_loop_mid =
  rule rg_repeat_loop_mid,
  goal_cases stab_pre stab_post guar_id loop_exit loop_body
```

We define dedicated syntax for the repeat-loop pattern.

```
definition Repeat :: "'a com ⇒ 'a bexp ⇒ 'a com" where
  "Repeat c b ≡ c ;; While (-b) c"

syntax "_Repeat" :: "'a com ⇒ 'a bexp ⇒ 'a com"  ("(0REPEAT _ /UNTIL _ /END)"  [0, 0]
61)
translations "REPEAT c UNTIL b END" ⇀ "CONST Repeat c {|b|}"

theorem rg_repeat_loop_def[intro]:
  assumes stab_pre:  "stable P R"
      and stab_post: "stable Q R"
      and guar_id:   "Id ⊆ G"
      and loop_exit: "P ∩ b ⊆ Q"
      and loop_body: "{P, R} C {G, P}"
    shows "{P, R} Repeat C b {G, Q}"
  using assms
  by (fastforce simp: Repeat_def rg_repeat_loop)

method method_repeat_loop_def =
  rule rg_repeat_loop_def,
  goal_cases stab_pre stab_post guar_id loop_exit loop_body
```

2.3.3 Conditionals

We first cover conditional-statements with or without the else-branch.

```
theorem rg_cond_named[intro]:
  assumes stab_pre:  "stable P R"
      and stab_post: "stable Q R"
      and guar_id:   "Id ⊆ G"
      and then_br:   "{P ∩  b, R} c1 {G, Q}"
      and else_br:   "{P ∩ -b, R} c2 {G, Q}"
    shows "{P, R} Cond b c1 c2 {G, Q}"
  using assms apply -
  by (rule Cond; fastforce)

theorem rg_cond2_named[intro]:
  assumes stab_pre:  "stable P R"
      and stab_post: "stable Q R"
      and guar_id:   "Id ⊆ G"
      and then_br:   "{P ∩  b, R} c1 {G, Q}"
      and else_br:   "P ∩ -b ⊆ Q"
    shows "{P, R} Cond b c1 SKIP {G, Q}"
  using assms apply -
  by (rule rg_cond_named; fastforce simp: rg_skip_named strengthen_pre)

method method_cond =
  (rule rg_cond2_named | rule rg_cond_named),
  goal_cases stab_pre stab_post guar_id then_br else_br
```

Variants without the stable-post subgoal.

```
theorem rg_cond_no_post[intro]:
  assumes stable_pre: "stable P R"
      and guar_id: "Id ⊆ G"
      and then_br: "{P ∩ b, R} c1 {G, Q}"
      and else_br: "{P ∩ -b, R} c2 {G, Q}"
    shows "{P, R} Cond b c1 c2 {G, Q}"
  using assms by (fastforce simp: Cond subset_iff)

theorem rg_cond_no_guard_no_post[intro]:
  assumes stable_pre: "stable P R"
      and guar_id: "Id ⊆ G"
      and then_br: "{P, R} c1 {G, Q}"
      and else_br: "{P, R} c2 {G, Q}"
    shows "{P, R} Cond b c1 c2 {G, Q}"
  using assms apply -
  by (rule Cond; fastforce intro: strengthen_pre)

method method_cond_no_post =
  (rule rg_cond_no_post | rule rg_cond_no_guard_no_post),
  goal_cases stab_pre guar_id then_br else_br
```

2.4 Parallel Compositions

We now turn to the parallel composition, and cover several variants, from the *binary* parallel composition of two commands, to the *multi-parallel* composition of an indexed list of commands. For each variant, we define the syntax and devise the subgoal-generating methods.

2.4.1 Binary Parallel

The syntax of binary parallel composition, without and with invariant.

```
abbreviation binary_parallel ::
```

```
  "'a set ⇒ 'a rel ⇒ 'a com ⇒ 'a com ⇒ 'a rel ⇒ 'a set ⇒ bool"
 ("{_, _}  _ ∥ _  {_, _}") where
  "{P, R} C1 ∥ C2 {G, Q} ≡
    ∃ P1 P2 R1 R2 G1 G2 Q1 Q2.
      ⊢ COBEGIN
          (C1, P1, R1, G1, Q1)
          ∥
          (C2, P2, R2, G2, Q2)
      COEND SAT [P, R, G, Q]"

abbreviation binary_parallel_invar ::
  "'a set ⇒ 'a rel ⇒ 'a com ⇒ 'a com ⇒ 'a set ⇒ 'a rel ⇒ 'a set ⇒ bool"
 ("{_, _}  _ ∥  _ // _ {_, _}") where
  "{P, R} C1 ∥ C2 // I {G, Q} ≡
    ∃ P1 P2 R1 R2 G1 G2 Q1 Q2.
      ⊢ COBEGIN
          (C1, P1, R1, G1, Q1)
          ∥
          (C2, P2, R2, G2, Q2)
      COEND SAT [P ∩ I, R ∩ pred_to_rel I, invar_and_guar I G, Q ∩ I]"
```

Some helper lemmas for later.

```
lemma simp_all_2:
  "(∀ i < Suc (Suc 0). P i) ⟷ P 0 ∧ P 1"
  by (fastforce simp: less_Suc_eq)

lemma simp_gen_Un_2:
  "(⋃ x ∈ {| ´(<) (Suc (Suc 0)) |}. S x) = S 0 ∪ S 1"
  by (fastforce simp: less_Suc_eq)

lemma simp_gen_Un_2_not0:
  "(⋃ x ∈ {| ´(<) (Suc (Suc 0)) ∧  ´(≠) (Suc 0) |}. S x) = S 0"
  by (fastforce simp: less_Suc_eq)

lemma simp_gen_Int_2:
  "(⋂ x ∈ {| ´(<) (Suc (Suc 0)) |}. S x) = S 0 ∩ S 1"
  by (fastforce simp: less_Suc_eq)

theorem rg_binary_parallel:
  assumes "{P1, R1} (C1::'a com) {G1, Q1}"
      and "{P2, R2} (C2::'a com) {G2, Q2}"
      and "G1 ⊆ R2"
      and "G2 ⊆ R1"
      and "P ⊆ P1 ∩ P2"
      and "R ⊆ R1 ∩ R2"
      and "G1 ∪ G2 ⊆ G"
      and "Q1 ∩ Q2 ⊆ Q"
    shows "⊢ COBEGIN
      (C1, P1, R1, G1, Q1)
       ∥
      (C2, P2, R2, G2, Q2)
      COEND SAT [P, R, G, Q]"
  using assms apply -
  apply (rule Parallel)
  by (simp_all add: simp_all_2 simp_gen_Un_2 simp_gen_Int_2 simp_gen_Un_2_not0)

theorem rg_binary_parallel_exists:
  assumes "{P1, R1} (C1::'a com) {G1, Q1}"
```

```
    and "{P2, R2} (C2::'a com) {G2, Q2}"
    and "G1 ⊆ R2"
    and "G2 ⊆ R1"
    and "P ⊆ P1 ∩ P2"
    and "R ⊆ R1 ∩ R2"
    and "G1 ∪ G2 ⊆ G"
    and "Q1 ∩ Q2 ⊆ Q"
  shows "{P, R} C1 ∥ C2 {G, Q}"
  by (metis assms rg_binary_parallel)

theorem rg_binary_parallel_invar_conseq:
  assumes C1: "{P1, R1} (C1::'a com) // I {G1, Q1}"
    and C2: "{P2, R2} (C2::'a com) // I {G2, Q2}"
    and "G1 ⊆ R2"
    and "G2 ⊆ R1"
    and "P ⊆ P1 ∩ P2"
    and "R ⊆ R1 ∩ R2"
    and "Q1 ∩ Q2 ⊆ Q"
    and "G1 ∪ G2 ⊆ G"
  shows "{P, R} C1 ∥ C2 // I {G, Q}"
  using assms apply -
  apply (rule rg_binary_parallel_exists)
  by force+
```

2.4.2 Multi-Parallel

The syntax of multi-parallel, without and with invariants.

```
syntax multi_parallel ::
  "'a set ⇒ 'a rel ⇒ idt ⇒ nat ⇒
   (nat ⇒ 'a set) ⇒ (nat ⇒ 'a rel) ⇒
   (nat ⇒ 'a com) ⇒
   (nat ⇒ 'a rel) ⇒ (nat ⇒ 'a set) ⇒
   'a rel ⇒ 'a set ⇒ bool"
  ("global'_init: _ global'_rely: _ ∥  _ < _ @ {_,_} _ {_,_} global'_guar: _ global'_post:
_")

translations
  "global_init: Init global_rely: RR ∥ i < N @
   {P,R} c {G,Q} global_guar: GG global_post: QQ"
  ⇀ "⊢ COBEGIN SCHEME [0 ≤ i < N] (c, P, R, G, Q) COEND
     SAT [Init, RR , GG, QQ]"

syntax multi_parallel_inv ::
  "'a set ⇒ 'a rel ⇒ idt ⇒ nat ⇒
   (nat ⇒ 'a set) ⇒ (nat ⇒ 'a rel) ⇒
   (nat ⇒ 'a com) ⇒ (nat ⇒ 'a set) ⇒
   (nat ⇒ 'a rel) ⇒ (nat ⇒ 'a set) ⇒
   'a rel ⇒ 'a set ⇒ bool"
  ("global'_init: _ global'_rely: _ ∥  _ < _ @ {_,_} _ // _ {_,_} global'_guar: _ global'_post:
_")

translations
  "global_init: Init global_rely: RR ∥ i < N @
   {P, R} c // I {G, Q} global_guar: GG global_post: QQ"
  ⇀ "⊢ COBEGIN SCHEME [0 ≤ i < N] (c,
         P ∩ I,
         R ∩ CONST pred_to_rel I,
         CONST invar_and_guar I G,
```

```
        Q ∩ I
      ) COEND
   SAT [Init, RR , GG, QQ]"
```

The subgoal-generating method for multi-parallel.

```
theorem rg_multi_parallel_subgoals:
  assumes assm_guar_rely: "∀ i j. i ≠ j ⟶ i < N ⟶ j < N ⟶ G j ⊆ R i"
      and assm_pre:  "∀ i < N. P' ⊆ P i"
      and assm_rely: "∀ i < N. R' ⊆ R i"
      and assm_guar: "∀ i < N. G i ⊆ G'"
      and assm_post: "(⋂ i ∈ { i. i < N }. Q i) ⊆ Q'"
      and assm_local: "∀ i<N. ⊢ C i sat [P i, R i, G i, Q i]"
    shows "⊢ COBEGIN SCHEME [0 ≤ i < (N::nat)]
           (C i, P i, R i, G i, Q i)
           COEND SAT [P', R', G', Q']"
proof (rule Parallel, goal_cases)
  case 1 show ?case using assm_rely assm_guar_rely by (simp add: SUP_le_iff)
  case 2 show ?case using assm_guar  by force
  case 3 show ?case using assm_pre   by force
  case 4 show ?case using assm_post  by force
  case 5 show ?case using assm_local by force
qed

method method_multi_parallel = rule rg_multi_parallel_subgoals,
  goal_cases guar_rely pre rely guar post body

theorem rg_multi_parallel_nobound_subgoals:
  assumes assm_guar_rely: "∀ i j. i ≠ j ⟶ G j ⊆ R i"
      and assm_pre:  "∀ i. P' ⊆ P i"
      and assm_rely: "∀ i. R' ⊆ R i"
      and assm_guar: "∀ i. G i ⊆ G'"
      and assm_post: "(⋂ i ∈ { i. i < N }. Q i) ⊆ Q'"
      and assm_local: "∀ i. ⊢ C i sat [P i, R i, G i, Q i]"
    shows "⊢ COBEGIN SCHEME [0 ≤ i < (N::nat)]
           (C i, P i, R i, G i, Q i)
           COEND SAT [P', R', G', Q']"
  using assms apply -
  apply (rule Parallel)
  by (simp_all add: SUP_le_iff INT_greatest)

method method_multi_parallel_nobound =
  rule rg_multi_parallel_nobound_subgoals,
  goal_cases guar_rely pre rely guar post body
```

2.5 Syntax of Record-Updates

This section contains syntactic sugars for updating a field of a record. As we use records to model the states of a program, these record-update operations correspond to the variable-assignments.

The type `idt` denotes a field of a record. The first syntactic sugar expresses a Basic command (of type `‹'a com›`) that updates a record-field `x` that is a function; often `x` models an array. After the update, the new value of `‹x i›` becomes `a`.

```
syntax "_record_array_assign" ::
  "idt ⇒ 'index ⇒ 'expr ⇒ 'state com" ("(´_[_] :=/ _)" [70, 65, 64] 61)
translations "´x[i] := a"
  ⇀ "CONST Basic «´(_update_name x (λ_. ´x(i:= a)))»"
```

The next two syntactic sugars express a state-transformation function (rather than a command) that updates record-fields. The first one simply updates an entire variable x, while the second updates an array ‹x i›.

```
syntax "_record_update_field" ::
  "idt ⇒ 'expr ⇒ ('a ⇒ 'a)" ("´_ ←/ _" [70] 61)
translations "´x ← a"
  ⇌ "«´(_update_name x (λ_. a))»"

syntax "_record_update_array" ::
  "idt ⇒ 'expr ⇒ 'expr ⇒ ('a ⇒ 'a)" ("´_[_] ←/ _" [70, 71] 61)
translations "´x[i] ← a"
  ⇀ "«´(_update_name x (λ_. ´x(i:= a)))»"
```

Syntactic sugars for incrementing variables.

```
syntax "_inc_fn" :: "idt ⇒ 'c ⇒ 'c"  ("(´_.++)" 61)
translations "´x.++ " ⇀
  " «´(_update_name x (λ_. ´x + 1))»"

syntax "_inc" :: "idt ⇒ 'c com"  ("(´_++)" 61)
translations "´x++ " ⇀
  "CONST Basic (´x.++)"

end
```

3 Annotated Commands

```
theory RG_Annotated_Commands

imports RG_Syntax_Extensions "HOL-Hoare.Hoare_Tac"

begin

datatype 'a anncom =
    NoAnno     "'a com"
  | BasicAnno  "'a ⇒ 'a"
  | WeakPre    "'a set"    "'a anncom"             ("{_} _" [65,61] 61)
  | StrongPost "'a anncom" "'a set"                ("_ {_}" [61,65] 61)
  | SeqAnno    "'a anncom" "'a set"    "'a anncom"
  | CondAnno   "'a bexp"   "'a anncom" "'a anncom"
  | WhileAnno  "'a bexp"   "'a set"    "'a anncom"
  | AwaitAnno  "'a bexp"   "'a anncom"

fun anncom_to_com :: "'a anncom ⇒ 'a com" where
    "anncom_to_com (NoAnno c)    = c"
  | "anncom_to_com (BasicAnno f) = Basic f"
  | "anncom_to_com (WeakPre b c)    = anncom_to_com c"
  | "anncom_to_com (StrongPost c b) = anncom_to_com c"
  | "anncom_to_com (SeqAnno c1 mid c2) = Seq     (anncom_to_com c1) (anncom_to_com c2)"
  | "anncom_to_com (CondAnno b c1 c2)  = Cond  b (anncom_to_com c1) (anncom_to_com c2)"
  | "anncom_to_com (WhileAnno b b' c)  = While b (anncom_to_com c)"
  | "anncom_to_com (AwaitAnno b c)     = Await b (anncom_to_com c)"

fun add_invar :: "'a set ⇒ 'a anncom ⇒ 'a anncom" where
    "add_invar I (NoAnno c)          = NoAnno c"
  | "add_invar I (BasicAnno f)       = BasicAnno f"
  | "add_invar I (WeakPre b c)       = WeakPre (b ∩ I) (add_invar I c)"
  | "add_invar I (StrongPost c b)    = StrongPost      (add_invar I c)  (b ∩ I)"
```

```
  | "add_invar I (SeqAnno c1 mid c2) = SeqAnno         (add_invar I c1) (mid ∩ I) (add_invar
I c2)"
  | "add_invar I (CondAnno b c1 c2)  = CondAnno  b     (add_invar I c1) (add_invar I c2)"
  | "add_invar I (WhileAnno b b' c)  = WhileAnno b b'  (add_invar I c)"
  | "add_invar I (AwaitAnno b c)     = AwaitAnno b     (add_invar I c)"
```

syntax

```
  "_CondAnno" :: "'a bexp ⇒ 'a anncom ⇒ 'a anncom ⇒ 'a anncom"
    ("(0IFa _/ THEN _/ ELSE _/FI)" [0, 0, 0] 61)
  "_Cond2Anno" :: "'a bexp ⇒ 'a anncom ⇒ 'a anncom"
    ("(0IFa _ THEN _ FI)" [0,0] 56)
  "_WhileAnno" :: "'a bexp ⇒ 'a set ⇒ 'a anncom ⇒ 'a anncom"
    ("(0WHILEa _ /DO {stable'_guard: _ } _ /OD)"  [0, 0] 61)
  "_WhileAnno_simple_b" :: "'a bexp ⇒ 'a anncom ⇒ 'a anncom"
    ("(0WHILEa _ /DO  _ /OD)"  [0, 0] 61)
  "_AwaitAnno" :: "'a bexp ⇒ 'a anncom ⇒ 'a anncom"
    ("(0AWAITa _ /THEN /_ /END)"  [0,0] 61)
  "_AtomAnno" :: "'a com ⇒ 'a anncom"
    ("(⟨_⟩a)" 61)
  "_WaitAnno" :: "'a bexp ⇒ 'a anncom"
    ("(0WAITa _ END)" 61)
  "_CondAnno_NoAnnoions" :: "'a bexp ⇒ 'a com ⇒ 'a com ⇒ 'a anncom"
    ("(0IF. _/ THEN _/ ELSE _/FI)" [0, 0, 0] 61)
```

translations

```
  "IFa b THEN c1 ELSE c2 FI" ⇀ "CONST CondAnno ⦃b⦄ c1 c2"
  "IFa b THEN c FI" ⇌ "IFa b THEN c ELSE SKIP FI"
  "IF. b THEN c1 ELSE c2 FI" ⇀ "CONST CondAnno ⦃b⦄ (CONST NoAnno c1) (CONST NoAnno c2)"

  "WHILEa b DO {stable_guard: b'} c OD" ⇀ "CONST WhileAnno ⦃b⦄ b' c"
  "WHILEa b DO  c OD" ⇀ "CONST WhileAnno ⦃b⦄ ⦃b⦄ c"

  "AWAITa b THEN c END" ⇌ "CONST AwaitAnno ⦃b⦄ c"
  "⟨c⟩a" ⇌ "AWAITa CONST True THEN c END"
  "WAITa b END" ⇌ "AWAITa b THEN SKIP END"
```

abbreviation `no_assertions_semicolon ::`

```
  "'a anncom ⇒ 'a set ⇒ 'a anncom ⇒ 'a anncom"
 ("_ .; {_} _" [60,60,61] 60) where
  "c1 .; {m} c2 ≡ SeqAnno c1 m c2"
```

Below is a special syntax for Basic commands (type "com") encoded inside NoAnno annotated commands (type "anncom").

This allows us to keep our syntactic sugars for Basic commands, which are mostly assignments (":="), without having to redo them all for BasicAnno annotated commands.

Hence, we wrap Basic commands with this helper function, which is only defined for Basic commands.

fun `basic_to_basic_anno_syntax:: "'a com ⇒ 'a anncom" ("'(_')-")` **where**

```
  "basic_to_basic_anno_syntax (Basic f) = BasicAnno f"
| "basic_to_basic_anno_syntax c = NoAnno c"
```

The following function defines what it means for an annotated command to satisfy the given specification components. The soundness of this definition will be proved later.

fun `anncom_spec_valid :: "'a set ⇒ 'a rel ⇒ 'a rel ⇒ 'a set ⇒ 'a anncom ⇒ bool"` **where**

```
    "anncom_spec_valid pre rely guar post (NoAnno c)
    = (⊢ c sat [pre, rely, guar, post])"
```

```
  | "anncom_spec_valid pre rely guar post (BasicAnno f)
    = (stable pre rely ∧
       stable post rely ∧
       (∀s. s ∈ pre ⟶ (s, s) ∈ guar) ∧
       (∀s. s ∈ pre ⟶ (s, f s) ∈ guar) ∧
       pre ⊆ ⦃ ´f ∈ post ⦄)"

  | "anncom_spec_valid pre rely guar post (WeakPre p' ac)
    = ((pre ⊆ p') ∧
       (anncom_spec_valid p' rely guar post ac))"

  | "anncom_spec_valid pre rely guar post (StrongPost ac q')
    = ((q' ⊆ post) ∧
       (anncom_spec_valid pre rely guar q' ac))"

  | "anncom_spec_valid pre rely guar post (SeqAnno ac1 mid ac2)
    = ((anncom_spec_valid pre rely guar mid  ac1) ∧
       (anncom_spec_valid mid rely guar post ac2))"

  | "anncom_spec_valid pre rely guar post (CondAnno b ac1 ac2)
    = ((stable pre rely) ∧
       (Id ⊆ guar) ∧
       (anncom_spec_valid (pre ∩  b) rely guar post ac1) ∧
       (anncom_spec_valid (pre ∩ -b) rely guar post ac2))"

  | "anncom_spec_valid pre rely guar post (WhileAnno b b' ac)
    = ((stable pre rely) ∧
       (stable post rely) ∧
       (Id ⊆ guar) ∧
       (pre ∩ -b ⊆ post) ∧
       (pre ∩  b ⊆ b') ∧
       (anncom_spec_valid (pre ∩ b') rely guar pre ac))"

  | "anncom_spec_valid pre rely guar post (AwaitAnno b ac)
    = ((stable pre rely) ∧
       (stable post rely) ∧
       (∀ s. anncom_spec_valid (pre ∩ b ∩ {s}) Id UNIV ({s'. (s, s') ∈ guar} ∩ post) ac))"
```

The following theorem establishes the soundness of the definition above.

```
theorem anncom_spec_valid_sound:
  "anncom_spec_valid pre rely guar post ac ⟹ ⊢ anncom_to_com ac sat [pre, rely, guar,
post]"
proof (induction ac arbitrary: pre rely guar post)
  case (NoAnno x)
  thus ?case
    by (cases x; fastforce)
next
  case (BasicAnno x)
  thus ?case
    by (fastforce simp: rg_basic_named)
next
  case (WeakPre pre' ac)
  thus ?case
    by (fastforce dest: Conseq)
next
  case (StrongPost ac x2)
  thus ?case
```

```
    by (fastforce simp: weaken_post)
next
  case (SeqAnno ac1 x2 ac2)
  thus ?case
    by (fastforce simp: Seq)
next
  case (CondAnno x1a ac1 ac2)
  thus ?case
    by (fastforce simp: Cond subset_iff)
next
  case (WhileAnno x1a x2 ac)
  thus ?case
    apply clarsimp
    apply (rule While,simp_all)
      apply (metis le_inf_iff Int_lower1 strengthen_pre)
    by fastforce
next
  case (AwaitAnno x1a ac)
  thus ?case
    apply clarsimp
    apply (rule Await, simp_all)
   by (smt (verit, best) Conseq IdI case_prodE mem_Collect_eq subset_iff)
qed
```

3.1 Annotated Quintuples

For convenience, we define the following datatype, which collects an annotated command with its specification components.

```
datatype 'a annquin = AnnQuin "'a set" "'a rel" "'a anncom" "'a rel" "'a set"
  ("{_,_} _ {_,_}" )

abbreviation annquin_invar ::
  "'a set ⇒ 'a rel ⇒ 'a anncom ⇒ 'a set ⇒ 'a rel ⇒ 'a set ⇒ 'a annquin"
  ("{_,_} _ // _ {_,_}") where
  "annquin_invar pre rely ac I guar post ≡ AnnQuin
    (pre ∩ I) (rely ∩ pred_to_rel I)
    (add_invar I ac)
    (invar_and_guar I guar) (post ∩ I)"
```

Helper functions for extracting the individual components of an ‹'a annquin›.

```
fun  preOf :: "'a annquin ⇒ 'a set"
  where "preOf (AnnQuin pre rely ac guar post) = pre"

fun relyOf :: "'a annquin ⇒ 'a rel"
  where "relyOf (AnnQuin pre rely ac guar post) = rely"

fun  cmdOf :: "'a annquin ⇒ 'a anncom"
  where "cmdOf (AnnQuin pre rely ac guar post) = ac"

fun guarOf :: "'a annquin ⇒ 'a rel"
  where "guarOf (AnnQuin pre rely ac guar post) = guar"

fun postOf :: "'a annquin ⇒ 'a set"
  where "postOf (AnnQuin pre rely ac guar post) = post"
```

Validity of ‹'a annquin› is the same as the validity of the “quintuples” when written out separately.

```
abbreviation annquin_valid :: "'a annquin ⇒ bool" where
  "annquin_valid rgac ≡ case rgac of (AnnQuin pre rely ac guar post) ⇒
   anncom_spec_valid pre rely guar post ac"

lemma annquin_simp[simp]:
  "annquin_valid (AnnQuin p r c g q) = anncom_spec_valid p r g q c"
  by fastforce
```

Syntax for expressing a valid ‹'a annquin› in terms of its components.

```
syntax
  "_valid_annquin"
    :: "'a rel ⇒ 'a rel ⇒ 'a set ⇒ 'a anncom ⇒ 'a set ⇒ bool"
    ("rely:_ guar:_ anno'_code: {_} _ {_}")
  "_valid_annquin_invar"
    :: "'a rel ⇒ 'a rel ⇒ 'a set ⇒ 'a set ⇒ 'a anncom ⇒ 'a set ⇒ bool"
    ("rely:_ guar:_ inv:_ anno'_code: {_} _ {_}")

translations
  "rely: R guar: G anno_code: {P} ac {Q}"
  ⇀ "CONST annquin_valid (CONST AnnQuin P R ac G Q)"
  "rely: R guar: G inv: I anno_code: {P} ac {Q}"
  ⇀ "CONST annquin_valid (CONST AnnQuin
        (P ∩ I) (R ∩ CONST pred_to_rel I)
        (CONST add_invar I ac)
        (CONST invar_and_guar I G) (Q ∩ I))"
```

3.2 Structured Tactics for Annotated Commands

```
lemma anncom_subgoals_no:
  "⊢ c sat [pre, rely, guar, post] ⟹ anncom_spec_valid pre rely guar post (NoAnno c)"
  by fastforce

lemma anncom_subgoals_invar_no:
  assumes "⊢ c sat [pre ∩ I, rely ∩ pred_to_rel I, invar_and_guar I guar, post ∩ I]"
  shows "anncom_spec_valid (pre ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                           (post ∩ I) (add_invar I (NoAnno c))"
  using assms by fastforce

lemma anncom_subgoals_basicanno_invar:
  assumes stable_pre:     "stable (pre ∩ I) (rely ∩ pred_to_rel I)"
      and stable_post:    "stable (post ∩ I) (rely ∩ pred_to_rel I)"
      and guar_id:        "∀s. s ∈ (pre ∩ I) ⟶ (s, s) ∈ (invar_and_guar I guar)"
      and establish_guar: "∀s. s ∈ (pre ∩ I) ⟶ (s, f s) ∈ (invar_and_guar I guar)"
      and establish_post: "(pre ∩ I) ⊆ ⦃ ´f ∈ (post ∩ I) ⦄"
  shows "rely: rely guar: guar inv: I anno_code: {pre} (BasicAnno f) {post}"
  using assms by fastforce

method method_annquin_basicanno =
  rule anncom_subgoals_basicanno_invar,
  goal_cases stable_pre stable_post guar_id est_guar est_post

lemma anncom_subgoals_seq:
  assumes "anncom_spec_valid pre rely guar mid  c1"
      and "anncom_spec_valid mid rely guar post c2"
    shows "anncom_spec_valid pre rely guar post (SeqAnno c1 mid c2)"
  using assms by fastforce

lemma anncom_subgoals_invar_seq:
```

```
  assumes "anncom_spec_valid (pre ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                             (mid  ∩ I) (add_invar I c1)"
      and "anncom_spec_valid (mid ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                             (post ∩ I) (add_invar I c2)"
    shows "anncom_spec_valid (pre ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                             (post ∩ I) (add_invar I (SeqAnno c1 mid c2))"
  using Seq assms by fastforce

lemma anncom_subgoals_invar_seq_abbrev:
  assumes "anncom_spec_valid (pre ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                             (mid  ∩ I) (add_invar I c1)"
      and "anncom_spec_valid (mid ∩ I) (rely ∩ pred_to_rel I) (invar_and_guar I guar)
                             (post ∩ I) (add_invar I c2)"
    shows "rely: (rely) guar: guar inv: I anno_code: {pre} (c1 .; {mid} c2) {post}"
  using Seq assms by fastforce

method method_annquin_seq =
  (rule anncom_subgoals_invar_seq | rule anncom_subgoals_invar_seq_abbrev),
  goal_cases c1 c2

lemma anncom_subgoals_while:
  assumes "stable pre rely"
      and "stable post rely"
      and "Id ⊆ guar"
      and "pre ∩ -b ⊆ post"
      and "pre ∩  b ⊆ b'"
      and "anncom_spec_valid (pre ∩ b') rely guar pre ac"
    shows "anncom_spec_valid pre rely guar post (WhileAnno b b' ac)"
  using assms by fastforce

lemma add_invar_while:
  assumes "anncom_spec_valid (p ∩ I) (R ∩ pred_to_rel I) (invar_and_guar I G)
                             (q ∩ I) (WhileAnno b b' (add_invar I ac))"
    shows "anncom_spec_valid (p ∩ I) (R ∩ pred_to_rel I) (invar_and_guar I G)
                             (q ∩ I) (add_invar I (WhileAnno b b' ac))"
  using assms by fastforce

lemma anncom_subgoals_invar_while_abbrev:
  assumes "anncom_spec_valid (p ∩ I) (R ∩ pred_to_rel I) (invar_and_guar I G)
                             (q ∩ I) (add_invar I (WhileAnno b b' ac))"
    shows "rely: R guar: G inv: I anno_code: {p} (WhileAnno b b' ac) {q}"
  using assms by fastforce

method method_annquin_while =
  rule anncom_subgoals_invar_while_abbrev,
  rule add_invar_while,
  rule anncom_subgoals_while,
  goal_cases stable_pre stable_post guar_id neg_guard guard body
```

3.3 Binary Parallel

This section contains inference rules for two annotated commands running in parallel. For convenience, we first define a datatype that encapsulates the components.

```
datatype 'a binary_par_quin = ParCode
  "'a set" "'a rel" "'a annquin" "'a annquin" "'a rel" "'a set"
  ("{_,_} _ ∥a _ {_,_}")
```

The next function sets out the proof obligations of binary parallel, using the datatype ‹'a binary_par_quin› above. It is then followed by the theorem that establishes the soundness of the inference rule encoded by the function binary_parallel_valid.

```
fun binary_parallel_valid:: "'a binary_par_quin ⇒ bool" where
 "binary_parallel_valid (ParCode init gr (AnnQuin p1 r1 c1 g1 q1) (AnnQuin p2 r2 c2 g2 q2)
gg final)
  = ( annquin_valid (AnnQuin p1 r1 c1 g1 q1)
    ∧ annquin_valid (AnnQuin p2 r2 c2 g2 q2)
    ∧    init ⊆ p1 ∩ p2
    ∧      gr ⊆ r1 ∩ r2
    ∧      g1 ⊆ r2
    ∧      g2 ⊆ r1
    ∧ g1 ∪ g2 ⊆ gg
    ∧ q1 ∩ q2 ⊆ final)"

theorem valid_binary_parallel:
 "binary_parallel_valid (ParCode init gr (AnnQuin p1 r1 c1 g1 q1) (AnnQuin p2 r2 c2 g2 q2)
gg final)
 ⟹ ⊢ COBEGIN (anncom_to_com c1, p1, r1, g1, q1) ∥ (anncom_to_com c2, p2, r2, g2, q2)
       COEND SAT [init, gr, gg, final]"
  by (rule Parallel; force intro:anncom_spec_valid_sound simp: less_Suc_eq)
```

Variants of the theorem above.

```
theorem valid_binary_parallel_exists:
 "binary_parallel_valid (ParCode init gr (AnnQuin p1 r1 c1 g1 q1) (AnnQuin p2 r2 c2 g2 q2)
gg final)
 ⟹ {init, gr} anncom_to_com c1 ∥ anncom_to_com c2 {gg, final}"
  by (fast dest: valid_binary_parallel)

theorem valid_binary_parallel_exists_annotated:
  assumes "binary_parallel_valid (ParCode
          init gr
          (AnnQuin p1 r1 c1' g1 q1) (AnnQuin p2 r2 c2' g2 q2)
          gg final)"
      and "anncom_to_com c1' = c1"
      and "anncom_to_com c2' = c2"
    shows "{init, gr} c1 ∥ c2 {gg, final}"
  using assms
  by (fast dest: valid_binary_parallel)
```

3.4 Helpers: Index Offsets

Before moving on to multi-parallel programs, we first prepare some lemmas that help reason about offsets and indices.

```
abbreviation nat_range_set_neq_i :: "nat ⇒ nat ⇒ nat ⇒ nat set"
  ("{_..<_≠_}") where
  "nat_range_set_neq_i lo hi x ≡ {lo..<hi} - {x}"

lemma all_set_range_to_offset:
  "(∀i∈{lo..<hi::nat}. P (f i)) ⟷ (∀i<(hi-lo). P (f (lo + i)))"
  by (metis add.commute atLeastLessThan_iff less_diff_conv nat_le_iff_add)

lemma Int_set_range_to_offset:
  "(⋂i∈{lo..<hi::nat}.  f i) = (⋂i<(hi-lo).  f (lo + i))"
  by (fastforce simp: le_iff_add)

lemma Un_set_range_to_offset:
  "(⋃i∈{lo..<hi::nat}.  g (f i)) = (⋃i<(hi-lo).  g (f (lo + i)))"
```

```
  apply standard
   apply clarsimp
   apply (metis add_diff_cancel_left' diff_less_mono lessThan_iff nat_le_iff_add)
  by fastforce

lemma Int_set_range_neq_to_offset: "i = lo + ii
  ⟹ (⋂j∈{lo..<hi≠i}.  f j) = (⋂j∈{0..<(hi-lo)≠ ii}.  f (lo + j))"
  unfolding Ball_def Bex_def image_def
  apply clarsimp
  by (metis (lifting) diff_add_inverse diff_less_mono less_diff_conv nat_le_iff_add)

lemma Int_set_range_neq_to_offset2: "ii< (hi - lo)
  ⟹ (⋂j∈{lo..<hi≠(lo + ii)}.  f j) = (⋂j∈{0..<(hi-lo)≠ ii}.  f (lo + j))"
  unfolding Ball_def Bex_def image_def
  apply clarsimp
  by (metis (lifting) add.commute add_left_imp_eq less_diff_conv nat_le_iff_add)

lemma forall_range_to_offset:
  "(∀i∈{lo..<(hi::nat)}. P i) ⟷ (∀i∈{0..<(hi - lo)}. P (lo + i))"
  unfolding Ball_def
  apply clarsimp
  by (metis add.commute le_add1 le_add_diff_inverse less_diff_conv)

lemma SCHEME_map_domain:
  "map (λi. rgac i) [lo ..< (N::nat)] = map (λi. rgac (lo + i)) [0..<(N-lo)]"
  by (induct N arbitrary: lo; simp add: Suc_diff_le)

lemma offset_P: "(∀i. lo ≤ i ∧ i < (N::nat) ⟶ P i) ⟹ lo ≤ N ⟹ (∀i. i <(N-lo) ⟶
P (lo + i))"
  by fastforce

lemma INTER_offset:
  shows "(⋂x<((N::nat) - lo). p (lo + x)) = (⋂x∈{lo..<N}. p x)"
  by (simp add: Ball_def Int_set_range_to_offset)

lemma LT_offset: "(∀i. lo ≤ i ∧ i < (N::nat) ⟶ P i) ⟷ (∀i<N - lo. P (lo + i))"
  by (metis add.commute le_add1 le_add_diff_inverse2 less_diff_conv)
```

3.5 Multi-Parallel

This section contains inference rules for multiple annotated commands running in parallel. Again, for convenience we first define a datatype that encapsulates the components:

1. Global precondition
2. Global rely
3. The lower index
4. The upper index
5. Sequential programs (each an annotated quintuple), indexed by the natural numbers
6. Global guarantee
7. Global postcondition

```
datatype 'a multi_par_quin = MultiParCode
```

```
  "'a set"
  "'a rel"
   nat nat
  "nat ⇒ 'a annquin"
  "'a rel"
  "'a set"
```

Using the datatype above, the inference rules are set out as the following two functions.

```
fun multipar_valid :: "'a multi_par_quin ⇒ bool" where
  "multipar_valid (MultiParCode init RR lo N iac gg final) =
   ( (∀i∈{lo..<N}. annquin_valid (iac i)) ∧
      init ⊆ (⋂i∈{lo..<N}.  preOf (iac i)) ∧
      RR ⊆ (⋂i∈{lo..<N}. relyOf (iac i)) ∧
      (∀i∈{lo..<N}. guarOf (iac i) ⊆ (⋂j∈{lo..<N≠i}. relyOf (iac j))) ∧
      (⋃i∈{lo..<N}. guarOf (iac i)) ⊆ gg ∧
      (⋂i∈{lo..<N}. postOf (iac i)) ⊆ final )"

fun multipar_valid_offset:: "'a multi_par_quin ⇒ bool" where
  "multipar_valid_offset (MultiParCode init RR lo N iac gg final) =
   ( (∀i<(N-lo). annquin_valid (iac (lo + i))) ∧
      init ⊆ (⋂i<(N-lo).  preOf (iac (lo + i))) ∧
      RR ⊆ (⋂i<(N-lo). relyOf (iac (lo + i))) ∧
      (∀i<(N-lo). guarOf (iac (lo + i)) ⊆ (⋂j∈{0..<(N-lo)≠i}. relyOf (iac (lo + j))))
∧
      (⋃i<(N-lo). guarOf (iac (lo + i))) ⊆ gg ∧
      (⋂i<(N-lo). postOf (iac (lo + i))) ⊆ final )"
```

Alternative syntax that encodes the validity of multi-parallel statements.

```
syntax
  "_multi_parallel_anno"
    :: "'a set ⇒ 'a rel ⇒ idt ⇒ nat ⇒ 'a annquin ⇒ 'a rel ⇒ 'a set ⇒ bool"
    ("annotated global'_init: _ global'_rely: _ ∥  _ < _ @ _ global'_guar: _ global'_post:
_")
  "_multi_parallel_anno_lo_hi"
    :: "'a set ⇒ 'a rel ⇒ nat ⇒ idt ⇒ nat ⇒ 'a annquin ⇒ 'a rel ⇒ 'a set ⇒ bool"
    ("annotated global'_init: _ global'_rely: _ ∥  _ ≤ _ < _ @ _ global'_guar: _ global'_post:
_")

translations
  "annotated global_init: Init global_rely: RR ∥ i < N @ rgac global_guar: GG global_post:
QQ"
  ⇀ "CONST multipar_valid (CONST MultiParCode Init RR 0 N (λi. rgac) GG QQ)"
  "annotated global_init: Init global_rely: RR ∥ lo ≤ i < hi @ rgac global_guar: GG global_post:
QQ"
  ⇀ "CONST multipar_valid_offset (CONST MultiParCode Init RR lo hi (λi. rgac) GG QQ)"
```

The soundness of the inference rules, in multiple variants.

```
lemma multipar_valid_offset_equiv:
  "multipar_valid        (MultiParCode init RR lo hi iac gg final) ⟷
   multipar_valid_offset (MultiParCode init RR lo hi iac gg final)"
  apply clarsimp
  apply (intro conjI iffI)
            apply fastforce
           apply fastforce
          apply fastforce
         apply fastforce
        apply fastforce
```

```
        apply (fastforce simp: Int_set_range_to_offset)
       apply (metis (lifting) atLeastLessThan_iff diff_less_mono le_add_diff_inverse)
      apply (metis (no_types, lifting) ext INTER_offset)
      apply (metis (no_types, lifting) ext INTER_offset)
    apply (simp add: Ball_def)
    apply (smt (verit, ccfv_threshold) Int_set_range_neq_to_offset Sup.SUP_cong le_add_diff_inverse

                                   le_add_diff_inverse2 less_diff_conv)
   apply (metis Un_set_range_to_offset)
  by (fastforce simp: Int_set_range_to_offset)

theorem valid_multipar:
  "multipar_valid (MultiParCode Init RR lo N rgac GG QQ) ⟹
  ⊢ COBEGIN SCHEME [lo ≤ i < N] (
         CONST anncom_to_com (cmdOf (rgac i)),
          preOf (rgac i),
          relyOf (rgac i),
          guarOf (rgac i) ,
          postOf (rgac i)
        ) COEND
     SAT [Init, RR , GG, QQ]"
  apply (rule Parallel)
       apply (simp add: subset_iff)
      apply (metis Diff_iff add.commute add_left_cancel atLeastLessThan_iff empty_iff insert_iff

                  le_add1 less_diff_conv nth_upt)
     apply (fastforce intro: Ball_def simp: subset_iff)
    apply (fastforce simp: subset_iff)
   apply (simp add: subset_iff )
   apply (metis atLeastLessThan_iff diff_less_mono le_add_diff_inverse)
  apply (simp add: subset_iff)
  by (metis (lifting) ext add.commute anncom_spec_valid_sound annquin_simp atLeastLessThan_iff

                     cmdOf.simps guarOf.simps le_add1 less_diff_conv postOf.elims preOf.simps

                     relyOf.simps)


theorem valid_multipar_with_internal_rg:
  "multipar_valid (MultiParCode Init RR lo N (λi. AnnQuin (p i) (r i) (ac i) (g i) (q i))
GG QQ) ⟹
  (∀i. anncom_to_com (ac i) = c i) ⟹
  ⊢ COBEGIN SCHEME [lo ≤ i < N] ((c i), p i, r i, g i, q i) COEND
    SAT [Init, RR , GG, QQ]"
  unfolding  Ball_def
  apply (rule Parallel, simp_all add: subset_iff)
     apply (metis Diff_iff add.commute atLeastLessThan_iff diff_add_inverse less_diff_conv

                 nat_le_iff_add nth_upt singletonD)
    apply (metis add.commute atLeastLessThan_iff le_add1 less_diff_conv)
   apply (metis add.commute atLeastLessThan_iff less_diff_conv nat_le_iff_add)
  by (metis add.commute anncom_spec_valid_sound atLeastLessThan_iff le_add1 less_diff_conv)

theorem valid_multipar_explicit:
  assumes
    local_sat: "⋀i. lo ≤ i ∧ i < N ⟹ annquin_valid (iac i)" and
    pre:  "⋀i. lo ≤ i ∧ i < N ⟹ init ⊆  preOf (iac i)" and
    rely: "⋀i. lo ≤ i ∧ i < N ⟹ RR ⊆ relyOf (iac i)" and
```

```
    guar_imp_rely: "⋀i j. lo ≤ i ∧ i<N ⟹ lo ≤ j ∧ j < N ⟹ i ≠ j
                    ⟹ guarOf (iac i) ⊆ relyOf (iac j)" and
    guar: "⋀i. lo ≤ i ∧ i < N ⟹ guarOf (iac i) ⊆ gg" and
    post: "(⋂i∈{lo..<N}. postOf (iac i)) ⊆ final"
  shows "multipar_valid (MultiParCode init RR lo N iac gg final)"
  using assms by fastforce

theorem valid_multipar_offset_explicit:
  assumes
    local_sat: "⋀i. lo ≤ i ∧ i < N ⟹ annquin_valid (iac i)" and
    pre:  "⋀i. lo ≤ i ∧ i < N ⟹ init ⊆  preOf (iac i)" and
    rely: "⋀i. lo ≤ i ∧ i < N ⟹ RR ⊆ relyOf (iac i)" and
    guar_imp_rely: "⋀i j. lo ≤ i ∧ i<N ⟹ lo ≤ j ∧ j < N ⟹ i ≠ j
                    ⟹ guarOf (iac i) ⊆ relyOf (iac j)" and
    guar: "⋀i. lo ≤ i ∧ i < N ⟹ guarOf (iac i) ⊆ gg" and
    post: "(⋂i∈{lo..<N}. postOf (iac i)) ⊆ final"
  shows "multipar_valid_offset (MultiParCode init RR lo N iac gg final)"
  apply clarsimp
  apply (intro conjI, simp_all add: le_INF_iff assms)
    apply (simp add: guar_imp_rely less_diff_conv)
   apply (simp add: SUP_le_iff guar)
  using post by (fastforce simp: nat_le_iff_add SUP_le_iff guar)

theorem valid_multipar_explicit2:
  assumes
    local_sat: "⋀i. lo ≤ i ∧ i < N ⟹ annquin_valid {p i,r i} c i {g i ,q i}" and
    pre:  "⋀i. lo ≤ i ∧ i < N ⟹ init ⊆  p i" and
    rely: "⋀i. lo ≤ i ∧ i < N ⟹ RR ⊆ r i" and
    guar_imp_rely: "⋀i j. lo ≤ i ∧ i<N ⟹ lo ≤ j ∧ j < N ⟹ i ≠ j ⟹ g i ⊆ r j" and

    guar: "⋀i. lo ≤ i ∧ i < N ⟹ g i ⊆ gg" and
    post: "(⋂i∈{lo..<N}. q i) ⊆ final"
    shows "multipar_valid (MultiParCode init RR lo N (λi. {p i,r i} c i {g i ,q i}) gg
final)"
  using assms
  by (force simp: subset_iff)

theorem valid_multipar_explicit_with_invariant:
  assumes
    local_sat: "⋀i. i < N ⟹ annquin_valid {p i,r i} c i // Inv {g i ,q i}" and
    pre:  "⋀i. i < N ⟹ init ⊆  p i ∩ Inv" and
    rely: "⋀i. i < N ⟹ RR ⊆ r i ∩ pred_to_rel Inv" and
    guar_imp_rely: "⋀i j. i<N ⟹ j < N ⟹ i ≠ j
                    ⟹ invar_and_guar Inv (g i) ⊆ r j ∩ pred_to_rel Inv" and
    guar: "⋀i. i < N ⟹ invar_and_guar Inv (g i) ⊆ gg" and
    post: "(⋂i<N. q i ∩ Inv) ⊆ final"
  shows "multipar_valid (MultiParCode init RR 0 N (λi. {p i,r i} c i // Inv {g i ,q i})
gg final)"
  apply (rule valid_multipar_explicit2)
  using lessThan_atLeast0 assms by presburger+

method method_annquin_multi_parallel =
  rule valid_multipar_explicit2,
  goal_cases local_sat pre rely guar_imp_rely guar post
```

3.6 The Main Tactics

```
lemmas rg_syntax_simps_collection =
```

```
  multipar_valid.simps
  multipar_valid_offset.simps
  add_invar.simps
  basic_to_basic_anno_syntax.simps
  postOf.simps preOf.simps relyOf.simps guarOf.simps
  annquin_simp
  anncom_spec_valid.simps
```

method `rg_proof_expand = (auto simp only: rg_syntax_simps_collection ; simp?)`

```
method method_anno_ultimate =
    method_annquin_basicanno
  | method_annquin_seq+
  | method_annquin_while
  | method_annquin_multi_parallel
  | rg_proof_expand
```

end

4 Examples Reworked

The examples in the original library [2], expressed using our new syntax, and proved using our new tactics.

theory `RG_Examples_Reworked`

imports `RG_Annotated_Commands`

begin
declare `[[syntax_ambiguity_warning = false]]`

4.1 Setting Elements of an Array to Zero

```
record Example1 =
  A :: "nat list"
```

```
theorem Example1:
  "global_init: ⦃ n < length ´A ⦄
   global_rely: id(A)
    ∥ i < n @
     { ⦃ i < length ´A ⦄,
       ⦃ length ºA = length ªA ∧ ºA ! i = ªA ! i ⦄ }
    ´A := ´A[i := 0]
     { ⦃ length ºA = length ªA ∧ (∀j < n. i ≠ j ⟶ ºA ! j = ªA ! j) ⦄,
       ⦃ ´A ! i = 0 ⦄ }
   global_guar: ⦃ True ⦄
   global_post: ⦃ ∀i < n. ´A ! i = 0 ⦄"
  by method_rg_try_each
```

```
theorem Example1'':
  "annotated global_init: ⦃ length ´A = N ⦄ global_rely: ⦃ ªA = ºA ⦄
   ∥ i < N @
   { ⦃ True ⦄,
     ⦃  length ªA = length ºA ∧ ªA ! i = ºA ! i⦄ }
     (´A := ´A [i := f i])- // ⦃length ´A = N ⦄
   { ⦃ length ªA = length ºA ∧ (∀j. i ≠ j ⟶   ªA ! j = ºA ! j ) ⦄,
     ⦃ ´A ! i = f i ⦄ }
   global_guar: ⦃ length ªA = length ºA⦄
```

```
  global_post: ⦃ take N ´A = map f [0 ..< N] ⦄"
  apply rg_proof_expand
  by (fastforce split: if_splits simp: map_upt_eqI)
```

4.2 Incrementing a Variable in Parallel

Two Components

```
record Example2 =
  x   :: nat
  c_0 :: nat
  c_1 :: nat

lemma ex2_leftside:
  "{ ⦃ ´c_0 = 0 ⦄, id(c_0) }
     Basic ((´x ← ´x + 1) ∘> (´c_0 ← 1))
   // ⦃ ´x = ´c_0 + ´c_1 ⦄
   { id(c_1), ⦃ ´c_0 = 1 ⦄}"
  by method_rg_try_each

lemma ex2_rightside:
  "{ ⦃ ´c_1 = 0 ⦄, id(c_1) }
     Basic ((´x ← ´x + 1) ∘> (´c_1 ← 1))
   // ⦃ ´x = ´c_0 + ´c_1 ⦄
   { id(c_0), ⦃ ´c_1 = 1 ⦄}"
  by method_rg_try_each

theorem Example2b:
  "{ ⦃ ´c_0 = 0 ∧ ´c_1 = 0 ⦄, ids({c_0, c_1}) }
   (Basic ((´x ← ´x + 1) ∘> (´c_0 ← 1))) ‖ (Basic ((´x ← ´x + 1) ∘> (´c_1 ← 1)))
   // ⦃ ´x = ´c_0 + ´c_1 ⦄
   { UNIV, ⦃ True ⦄ }"
  using ex2_leftside ex2_rightside
  by (rule rg_binary_parallel_invar_conseq; blast)
```

Parameterised

```
lemma sum_split:
  "(j::nat) < (n::nat)
  ⟹ sum a {0..<n} = sum a {0..<j} + a j + sum a {j+1..<n}"
  by (metis Suc_eq_plus1 bot_nat_0.extremum group_cancel.add1 le_eq_less_or_eq sum.atLeastLessThan_c
sum.atLeast_Suc_lessThan)
```

Intuition of the lemma above: Consider the sum of a function `b k` with `k` ranging from 0 to `n - 1`. Let `j` be an index in this range, and assume `b j = 0`. Then, replacing `b j` with 1 in the sum, the result is the same as adding 1 to the original sum.

```
lemma Example2_lemma2_replace:
  assumes "(j::nat) < n"
      and "b' = b(j:=xx::nat)"
    shows "(∑ i = 0 ..< n. b' i) = (∑ i = 0 ..< n. b i) - b j + xx"
  apply (subst sum_split, rule assms(1))
  apply (subst sum_split, rule assms(1))
  using assms(2) by clarsimp

lemma Example2_lemma2_Suc0[simp]:
  assumes "(j::nat) < n"
      and "b j = 0"
      and "b' = b(j:=1)"
    shows "Suc (∑ i::nat = 0 ..< n. b i) = (∑ i = 0 ..< n. b' i)"
```

```
  using assms  Example2_lemma2_replace
  by fastforce

record Example2_param =
  y ::  nat
  C :: "nat ⇒ nat"

lemma Example2_local:
  "i < n ⟹
  { ⦃ ´C i = 0 ⦄,
     id(C @ i) }

    Basic ((´y ← ´y + 1) ∘> (´C ← ´C(i:=1)))
    // ⦃ ´y = (∑ k::nat = 0 ..< n. ´C k) ⦄

   { ⦃ ∀ j < n. i ≠ j ⟶ ºC j = ªC j ⦄,
     ⦃ ´C i = 1 ⦄ }"
  by method_rg_try_each

theorem Example2_param:
  assumes "0 < n" shows
  "global_init: ⦃ ´y = 0 ∧ sum ´C {0 ..< n} = 0 ⦄
   global_rely: id(C) ∩ id(y)
    ∥ i < n @
   { ⦃ ´C i = 0 ⦄,
     id(C @ i) }
     Basic ((´y ← ´y + 1) ∘> (´C ← ´C(i:=1)))
     // ⦃ ´y = sum ´C {0 ..< n} ⦄
   { ⦃ ∀ j < n. i ≠ j ⟶ ºC j = ªC j ⦄,
     ⦃ ´C i = 1 ⦄ }
   global_guar: ⦃ True ⦄
   global_post: ⦃ ´y = n ⦄"
proof method_multi_parallel
  case post
  then show ?case
    using assms by (clarsimp, fastforce)
next
  case body
  then show ?case
    by method_rg_try_each
qed (fastforce+)
```

As above, but using an explicit annotation and a different method.

```
theorem Example2_param_with_expansion:
  assumes "0 < n" shows "annotated
  global_init: ⦃ ´y = 0 ∧ sum ´C {0 ..< n} = 0 ⦄
  global_rely: id(C) ∩ id(y)
    ∥ i < n @
  { ⦃ ´C i = 0 ⦄,
    id(C @ i) }
    (Basic ((´y ← ´y + 1) ∘> (´C ← ´C(i:=1))))-
    // ⦃ ´y = sum ´C {0 ..< n} ⦄
  { ⦃ ∀ j < n. i ≠ j ⟶ ºC j = ªC j ⦄,
    ⦃ ´C i = 1 ⦄ }
  global_guar: ⦃ True ⦄
  global_post: ⦃ ´y = n ⦄"
  apply rg_proof_expand
  using assms by (fastforce split: if_splits)
```

4.3 FindP

Titled “Find Least Element” in the original [2], the "findP" problem assumes that `n` divides `m`, and runs `n` threads in parallel to search through a length-`m` array `B` for an element that satisfies a predicate `P`. The indices of the array `B` are partitioned into the congruence-classes modulo `n`, where Thread `i` searches through the indices that are congruent to `i` mod `n`.

In the program, `X i` is the next index to be checked by Thread `i`. Meanwhile, `Y i` is either the out-of-bound default `m + i` if Thread `i` has not found a `P`-element, or the index of the first `P`-element found by Thread `i`.

The first helper lemma: an equivalent version of `mod_aux` found in the original.

```
lemma mod_aux :
  "a mod (n::nat) = i ⟹  a < j ∧ j < a + n ⟹ j mod n ≠ i"
  using mod_eq_dvd_iff_nat nat_dvd_not_less by force

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

lemma Example3:
  assumes "m mod n=0" shows "annotated
  global_init: ⦃∀i < n. ´X i = i ∧ ´Y i = m + i ⦄
  global_rely: ⦃ ºX = ªX ∧ ºY = ªY⦄
    ‖ i < n @

  { ⦃(´X i) mod n=i ∧ (∀j<´X i. j mod n=i ⟶ ¬P(B!j)) ∧ (´Y i<m ⟶ P(B!(´Y i)) ∧ ´Y
i≤ m+i)⦄,
    ⦃(∀j<n. i≠j ⟶ ªY j ≤ ºY j) ∧ ºX i = ªX i ∧ ºY i = ªY i⦄ }

    WHILEa (∀ j < n. ´X i < ´Y j) DO
      {stable_guard: ⦃´X i < ´Y i⦄}
      IFa P(B!(´X i)) THEN
        (´Y[i] := ´X i)-
      ELSE
        (´X[i] := ´X i + n)-
      FI
    OD

  { ⦃(∀j<n. i≠j ⟶ ºX j = ªX j ∧ ºY j = ªY j) ∧ ªY i ≤ ºY i⦄,
    ⦃ (´X i) mod n = i ∧ (∀j<´X i. j mod n=i ⟶ ¬P(B!j))
    ∧ (´Y i<m ⟶ P(B!(´Y i)) ∧ ´Y i≤ m+i)
    ∧ (∃j<n. ´Y j ≤ ´X i) ⦄ }

  global_guar: ⦃True⦄
  global_post: ⦃ ∀ i < n. (´X i) mod n=i
               ∧ (∀j<´X i. j mod n=i ⟶ ¬P(B!j))
               ∧ (´Y i<m ⟶ P(B!(´Y i)) ∧´Y i≤ m+i)
               ∧ (∃j<n. ´Y j ≤ ´X i) ⦄"
  apply rg_proof_expand
          apply fastforce+
    apply (metis linorder_neqE_nat mod_aux)
   apply (metis antisym_conv3 mod_aux)
  by (metis leD mod_less_eq_dividend)
```

Below is the original version of the theorem, and is immediately derivable from the above. We include some formatting changes (such as line breaks) for better readability.

```
lemma Example3_original: "m mod n=0 ⟹
```

```
⊢ COBEGIN SCHEME [0≤i<n]

  (WHILE (∀ j < n. ´X i < ´Y j) DO
     IF P(B!(´X i)) THEN ´Y:=´Y (i:=´X i) ELSE ´X:= ´X (i:=(´X i)+ n) FI
   OD,

 ⦃(´X i) mod n=i ∧ (∀j<´X i. j mod n=i ⟶ ¬P(B!j)) ∧ (´Y i<m ⟶ P(B!(´Y i)) ∧ ´Y i≤
m+i)⦄,

 ⦃(∀j<n. i≠j ⟶ ªY j ≤ ºY j) ∧ ºX i = ªX i ∧ ºY i = ªY i⦄,

 ⦃(∀j<n. i≠j ⟶ ºX j = ªX j ∧ ºY j = ªY j) ∧ ªY i ≤ ºY i⦄,

 ⦃(´X i) mod n=i ∧ (∀j<´X i. j mod n=i ⟶ ¬P(B!j)) ∧ (´Y i<m ⟶ P(B!(´Y i)) ∧ ´Y i≤
m+i) ∧ (∃j<n. ´Y j ≤ ´X i) ⦄)

 COEND
 SAT [
   ⦃ ∀ i < n. ´X i = i ∧ ´Y i = m+i ⦄,

   ⦃ºX=ªX ∧ ºY=ªY⦄,

   ⦃True⦄,

   ⦃∀ i < n. (´X i) mod n=i ∧
            (∀j<´X i. j mod n=i ⟶ ¬P(B!j)) ∧
            (´Y i<m ⟶ P(B!(´Y i)) ∧´Y i≤ m+i) ∧
            (∃j<n. ´Y j ≤ ´X i)⦄]"
  by (rule valid_multipar_with_internal_rg[OF Example3]; simp)

end
```

5 Abstract Queue Lock

theory Lock_Abstract_Queue

imports
  RG_Annotated_Commands

begin

We identify each thread by a natural number.

type_synonym thread_id = nat

The state of the Abstract Queue Lock consists of one single field, which is the list of threads.

record queue_lock = queue :: "thread_id list"

The following abbreviation describes when an object is at the head of a list. Note that both clauses are needed to characterise the predicate faithfully, because the term x = hd xs (i.e. x is the head of xs) does not imply that x ∈ set xs.

abbreviation at_head :: "'a ⇒ 'a list ⇒ bool" **where**
  "at_head x xs ≡ xs ≠ [] ∧ x = hd xs"

The contract of the Abstract Queue Lock consists of two clauses. The first states that a thread cannot be added to or removed from the queue by its environment. The second states that the head of the queue remains at the head after any environment-step.

```
abbreviation queue_contract :: "thread_id ⇒ queue_lock rel" where
  "queue_contract i ≡ {|
    (i ∈ set ºqueue ⟷ i ∈ set ªqueue) ∧
    (at_head i ºqueue ⟶ at_head i ªqueue) |}"
```

The RG sentence of the Release procedure is made into a separate lemma below.

```
lemma qlock_rel:
 "rely: queue_contract t     guar: for_others queue_contract t
  inv:  {| distinct ´queue |}  code:
    { {| at_head t ´queue |} }
  ´queue := tl ´queue
    { {| t ∉ set ´queue |} }"
proof method_basic_inv
  case est_guar
  then show ?case
    apply clarsimp
    by (metis hd_Cons_tl list.set_sel(2) set_ConsD)
next
  case est_post
  then show ?case
    apply clarsimp
    by (metis distinct.simps(2) list.collapse)
qed (simp_all add: distinct_tl)
```

The correctness of the Abstract Queue Lock is expressed by the following RG sentence, which describes a closed system of `n` threads, each repeatedly calls Acquire and then Release in an infinite loop. We omit the critical section between Acquire and Release, as it does not access the lock.

The Acquire procedure consists of two steps: enqueuing and spinning. The Release procedure consists of only the dequeuing step.

Each thread can only be in the queue at most once, so the invariant requires the queue to be distinct.

The queue is initially empty; hence the global precondition. Being a closed system, there is no external actor, so the rely is the identity relation, and the guarantee is the universal relation. The system executes continuously, as the outer infinite loop never terminates; hence, the global postcondition is the empty set.

```
theorem qlock_global:
  assumes "0 < n"
  shows "annotated
  global_init: {| ´queue = [] |}  global_rely: Id
    ‖ i < n @
  { {| i ∉ set ´queue |}, queue_contract i }

  WHILEa True DO
    {stable_guard: {| i ∉ set ´queue |} }
    NoAnno (´queue := ´queue @ [i]) .;
    { {| i ∈ set ´queue |} }
    NoAnno (WHILE hd ´queue ≠ i DO SKIP OD) .;
    { {| at_head i ´queue |}}
    NoAnno (´queue := tl ´queue)
  OD

  // {| distinct ´queue |} { for_others queue_contract i, {} }
  global_guar: UNIV  global_post: {}"
  apply rg_proof_expand
     apply (method_basic; fastforce)
```

```
    apply (method_spinloop; fastforce)
   using qlock_rel apply fastforce
  using assms by fastforce
```

end

6 Ticket Lock

```
theory Function_Supplementary
imports Main
begin
```

This theory contains some function-related definitions and associated lemmas that are not included in the built-in library. They are grouped into two sections:

1. Predicates that describe functions that are injective or surjective when restricted to subsets of their domains or images.
2. A higher-order function that performs a list of updates on a function.

The content of this theory was conceived during a project on formal program verification of locks (i.e. mutexes). The new definitions and lemmas arose from the proof of data refinement from an abstract queue-lock to a ticket-lock.

Inspired by the theories *List Index* (Nipkow 2010) and *Fixed-Length Vectors* (Hupel 2023) on the Archive of Formal Proofs, we hope that these new definitions and lemmas may also be of help to others.

6.1 Helpers: Inj, Surj and Bij

It is sometimes useful to describe a function that is not injective in itself, but is injective when its image is restricted to a subset.

For example, consider the function $\{a \mapsto 1, b \mapsto 2, c \mapsto 2\}$. This function is not injective, but if its image is restricted to $\{1\}$, the new function $\{a \mapsto 1\}$ becomes injective.

This motivates the following definition.

```
definition inj_img :: "('a ⇒ 'b) ⇒ 'b set ⇒ bool" where
  "inj_img f B ≡ ∀ x1 x2. f x1 = f x2 ∧ f x1 ∈ B ⟶ x1 = x2"
```

Similarly, the next definition describes a function that becomes surjective when its codomain is restricted to a subset.

In other words, “surj_codom f B” means that *every element in* B *is mapped to by* f.

For example, consider the function that maps from the domain $\{a, b\}$ to the codomain $\{1, 2\}$ with the graph $\{a \mapsto 1, b \mapsto 1\}$. This function is not surjective, but if its codomain is restricted to $\{1\}$, then the new function becomes surjective.

```
definition surj_codom :: "('a ⇒ 'b) ⇒ 'b set ⇒ bool" where
  "surj_codom f B ≡ ∀ y ∈ B. (∃ x. f x = y)"
```

We can also describe a function that remains surjective on a subset of its domain.

In other words, “surj_on f A” means that *mappings that originate from* A *already span the entire codomain.*

Note that this is a notion stronger than plain surjectivity, which will be shown in the later subsection “Surj-Related”.

```
definition surj_on :: "('a ⇒ 'b) ⇒ 'a set ⇒ bool" where
  "surj_on f A ≡ ∀ y. (∃ x ∈ A. f x = y)"
```

Note that all three definitions above are most likely not included in the built-in library, as suggested by the outputs of the following search-commands.

```
find_consts name:"inj"
find_consts name:"surj"
```

6.1.1 Inj-Related

```
lemma inj_implies_inj_on: "inj f ⟹ inj_on f A"
  using inj_on_subset by blast

lemma inj_implies_inj_img: "inj f ⟹ inj_img f B"
  by (simp add: injD inj_img_def)

lemma inj_img_empty: "inj_img f {}"
  by (fastforce simp: inj_img_def)

lemma inj_img_singleton: "∀ x. f x ≠ b ⟹ inj_img f {b}"
  by (fastforce simp: inj_img_def)

lemma inj_img_subset:
  "⟦ inj_img f B ; B' ⊆ B ⟧ ⟹ inj_img f B'"
  by (fastforce simp: inj_img_def)

lemma inj_img_superset:
  "⟦ inj_img f B ; ∀ x. f x ∉ B' - B ⟧ ⟹ inj_img f B'"
  by (fastforce simp: inj_img_def)

lemma inj_img_not_mapped_to: "∀x. f x ∉ B ⟹ inj_img f B"
  by (fastforce simp: inj_img_def)

lemma inj_img_add_one_extra:
  "⟦ inj_img f B ; ∀ x. f x ≠ b ⟧ ⟹ inj_img f (B ∪ {b})"
  by (fastforce simp: inj_img_def)

lemma inj_img_union_1:
  "⟦ inj_img f B1 ; inj_img f B2 ⟧ ⟹ inj_img f (B1 ∪ B2)"
  by (fastforce simp: inj_img_def)

lemma inj_img_union_2:
  "⟦ inj_img f B1 ; ∀ x. f x ∉ B2 ⟧ ⟹ inj_img f (B1 ∪ B2)"
  by (simp add: inj_img_not_mapped_to inj_img_union_1)

lemma inj_img_fun_upd_notin:
  "⟦ inj_img f B ; ∀ x. f x ≠ b ⟧ ⟹ inj_img (fun_upd f a b) B"
  by (fastforce simp: inj_img_def)

lemma inj_img_fun_upd_singleton:
  "∀ x. f x ≠ b ⟹ inj_img (fun_upd f a b) {b}"
  by (simp add: inj_img_fun_upd_notin inj_img_singleton)
```

6.1.2 Surj-Related

Lemmas related to "surj codom".

```
lemma surj_implies_surj_codom: "surj f ⟹ surj_codom f B"
```

```
  by (metis surjD surj_codom_def)

lemma surj_codom_triv: "surj_codom f (f ‘ A)"
  by (fastforce simp: surj_codom_def)

lemma surj_codom_univ: "surj_codom f UNIV = surj f"
  by (metis surj_codom_def surj_def UNIV_I)

lemma surj_codom_empty: "surj_codom f {}"
  by (fastforce simp: surj_codom_def)

lemma surj_codom_singleton: "b ∈ range f ⟹ surj_codom f {b}"
  by (fastforce simp: surj_codom_def)

lemma surj_codom_subset:
  "⟦ surj_codom f B ; B’ ⊆ B ⟧ ⟹ surj_codom f B’"
  by (fastforce simp: surj_codom_def)

lemma surj_codom_union:
  "⟦ surj_codom f B1 ; surj_codom f B2 ⟧ ⟹ surj_codom f (B1 ∪ B2)"
  by (fastforce simp: surj_codom_def)
```

Lemmas related to “surj on”.

```
lemma surj_on_implies_surj: "surj_on f A ⟹ surj f"
  by (metis surj_def surj_on_def)

lemma surj_on_univ: "surj_on f UNIV = surj f"
  by (metis UNIV_I surjD surj_on_def surj_on_implies_surj)

lemma surj_on_never_emptyset: "¬ surj_on f {}"
  by (fastforce simp: surj_on_def)

lemma surj_on_superset:
  "⟦ surj_on f A ; A ⊆ A’ ⟧ ⟹ surj_on f A’"
  by (fastforce simp: surj_on_def)

lemma surj_on_union:
  "⟦ surj_on f A1 ; surj_on f A2 ⟧ ⟹ surj_on f (A1 ∪ A2)"
  by (fastforce simp: surj_on_superset)
```

6.1.3 Bij and Inv

This section relates the new definitions to the existing “bijective between” and “inverse” definitions.

```
lemma bij_betw_implies_inj_img: "bij_betw f UNIV B ⟹ inj_img f B"
  by (fastforce simp: bij_betw_def inj_implies_inj_img)

lemma bij_betw_implies_surj_codom: "bij_betw f A B ⟹ surj_codom f B"
  by (fastforce intro: f_the_inv_into_f_bij_betw simp: surj_codom_def)

lemma bij_betw_implies_surj_on: "bij_betw f A UNIV ⟹ surj_on f A"
  by (meson UNIV_I bij_betw_iff_bijections surj_on_def)
```

Other lemmas

```
lemma bij_extension:
  assumes "a ∉ A"
      and "b ∉ B"
```

```
    and "bij_betw f A B"
  shows "bij_betw (fun_upd f a b) (A ∪ {a}) (B ∪ {b})"
  by (metis assms bij_betw_combine bij_betw_cong bij_betw_singleton_iff disjoint_insert(1)
          fun_upd_other fun_upd_same inf_bot_right)

lemma bij_remove_one:
  assumes "a ∈ A"
    and "bij_betw f A B"
  shows "bij_betw f (A - {a}) (B - {f a})"
  using assms by (fastforce simp: bij_betwE bij_betw_DiffI)

lemma set_remove_one_element:
  assumes "x ∉ B"
    and "B ⊆ A"
    and "A - {x} ⊆ B"
  shows "A - {x} = B"
  using assms by blast

lemma inv_image_restrict_inj:
  assumes "bij_betw f A B"
    and "inj_img f B"
    and "f a ∈ B"
  shows "a ∈ inv f ‘ B"
  using assms by (fastforce simp: f_inv_into_f inj_img_def rev_image_eqI)

lemma inv_image_restrict:
  assumes "inj_on f A"
    and "f a ∈ B"
    and "∀x. (f x ∈ B ⟶ x ∈ A)"
  shows "a ∈ inv f ‘ B"
  using assms by (fastforce simp: f_inv_into_f inj_onD rev_image_eqI)

lemma inv_image_restrict_neg:
  assumes "bij_betw f A B"
    and "f a ∉ B"
    and "∀x. (f x ∈ B ⟶ x ∈ A)"
  shows "a ∉ inv f ‘ B"
  using assms apply clarsimp
  by (metis (mono_tags, lifting) f_inv_into_f f_the_inv_into_f_bij_betw range_eqI)

lemma inv_image_restrict_neg':
  assumes "surj_codom f B"
    and "f a ∉ B"
    and "∀x. (f x ∈ B ⟶ x ∈ A)"
  shows "a ∉ inv f ‘ B"
  using assms
  by (fastforce simp: surj_codom_def f_inv_into_f rangeI)

lemma bij_betw_inv1:
  assumes "bij_betw f A B"
    and "inj_img f B"
    and "f a ∈ B"
  shows "inv f (f a) = a"
  using assms by (fastforce simp: f_inv_into_f inj_img_def)

lemma bij_betw_inv2:
  assumes "bij_betw f A B"
    and "b ∈ B"
```

```
    shows "f (inv f b) = b"
  by (metis assms bij_betw_imp_surj_on f_inv_into_f rangeI)

lemma surj_codom_inj_on_vimage_bij_betw:
  "⟦ surj_codom f B ; inj_on f (vimage f B) ⟧ ⟹ bij_betw f (vimage f B) B"
  apply (rule bij_betwI')
  by (fastforce simp: inj_onD surj_codom_def)+
```

6.2 Helpers: Multi-Updates on Functions

```
fun fun_upd_list :: "('a ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a ⇒ 'b)" where
  "fun_upd_list f [] = f"
| "fun_upd_list f (xy # xys) = fun_upd (fun_upd_list f xys) (fst xy) (snd xy)"
```

This notion can also be defined following the `foldl` pattern, although this alternative form is not used.

```
fun fun_upd_list_l :: "('a ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a ⇒ 'b)" where
  "fun_upd_list_l f [] = f"
| "fun_upd_list_l f (xy # xys) = fun_upd_list_l (fun_upd f (fst xy) (snd xy)) xys"
```

Examples of the two definitions above.

```
value "fun_upd_list   (λx.0::nat) [(1::nat,1),(4,3),(6,6),(4,4)] 4"
value "fun_upd_list_l (λx.0::nat) [(1::nat,1),(4,3),(6,6),(4,4)] 4"
```

Both definitions above resemble "folds" with some un-currying, as shown by the following two lemmas.

```
lemma fun_upd_list_is_foldr:
  "fun_upd_list f0 pairs = foldr (λ pair f. fun_upd f (fst pair) (snd pair)) pairs f0"
  by (induct pairs; fastforce)

lemma fun_upd_list_l_is_foldl:
  "fun_upd_list_l f0 pairs = foldl (λ f pair. f(fst pair := snd pair)) f0 pairs"
  by (induct pairs arbitrary: f0; force)
```

These two definitions are equivalent when every domain-value is updated at most once.

```
lemma fun_upd_list_l_distinct_rewrite:
  "distinct (map fst (xy # xys))
   ⟹ fun_upd_list_l (fun_upd f (fst xy) (snd xy))  xys
     = fun_upd        (fun_upd_list_l f xys)        (fst xy) (snd xy)"
proof (induct xys arbitrary: xy f)
  case (Cons xy2 xys2)
  thus ?case
    by (metis (no_types, lifting) distinct_length_2_or_more fun_upd_list_l.simps(2) fun_upd_twist
list.simps(9))
qed (fastforce)

lemma fun_upd_list_defs_distinct_equiv:
  "distinct (map fst pairs) ⟹ fun_upd_list f pairs = fun_upd_list_l f pairs"
proof (induct pairs)
  case (Cons xy xys)
  thus ?case
    by (fastforce simp: fun_upd_list_l_distinct_rewrite)
qed (fastforce)
```

Smaller propositions

```
lemma fun_upd_list_distinct_rewrite:
  "distinct (map fst (xy # xys))
```

```
  ⟹ fun_upd_list (fun_upd f (fst xy) (snd xy))  xys
    = fun_upd      (fun_upd_list f xys)       (fst xy) (snd xy)"
  by (simp add: fun_upd_list_defs_distinct_equiv fun_upd_list_l_distinct_rewrite)

lemma fun_upd_list_hd_1:
  "fun_upd_list f (zip (x # xs) (y # ys)) x = y"
  by simp

lemma fun_upd_list_hd_2:
  "⟦ xs ≠ [] ; ys ≠ [] ⟧ ⟹ fun_upd_list f (zip xs ys) (hd xs) = hd ys"
  by (metis fun_upd_list_hd_1 list.collapse)

lemma fun_upd_list_not_hd:
  assumes "a ≠ x"
  shows "fun_upd_list f (zip (x # xs) (y # ys)) a = fun_upd_list f (zip xs ys) a"
  using assms by simp

lemma fun_upd_list_not_updated_map:
  assumes "a ∉ set (map fst xys)"
    shows "fun_upd_list f xys a = f a"
  using assms by (induction xys, simp_all)

lemma fun_upd_list_not_updated_zip:
  assumes "a ∉ set xs"
    shows "fun_upd_list f (zip xs ys) a = f a"
  by (metis assms fun_upd_list_not_updated_map in_set_takeD map_fst_zip_take)
```

6.2.1 Ordering of Updates

The next two lemmas shows that the ordering of the updates does not matter, as long as the updates are distinct.

```
lemma fun_upd_list_distinct_reorder:
  assumes "distinct (map fst pairs)"
      and "ab ∈ set pairs"
    shows "fun_upd_list f pairs
          = (fun_upd_list f (remove1 ab pairs)) (fst ab := snd ab)"
using assms
proof (induct pairs)
  case (Cons xy xys)
  thus ?case
  proof (cases "ab = xy")
    case True
    thus ?thesis by simp
  next
    case False

    hence "(fun_upd_list f (remove1 ab (xy # xys))) (fst ab := snd ab)
              = fun_upd_list f (xy # xys)"
      using Cons.hyps Cons.prems by fastforce
    thus ?thesis by fastforce
  qed
qed (fastforce)

lemma fun_upd_list_distinct_reorder_general:
  assumes "distinct (map fst pairs1)"
      and "distinct (map fst pairs2)"
      and "set pairs1 = set pairs2"
```

```
    shows "fun_upd_list f pairs1 = fun_upd_list f pairs2"
using assms
proof (induct pairs1 arbitrary: pairs2)
  case (Cons xy xys)
  hence "fun_upd_list f pairs2
         = (fun_upd_list f (remove1 xy pairs2))(fst xy := snd xy)"
    by (metis list.set_intros(1) fun_upd_list_distinct_reorder Cons.prems(2,3))
  also have "... = (fun_upd_list f xys)(fst xy := snd xy)"
    by (metis (mono_tags, lifting) Cons.hyps Cons.prems distinct_map distinct_remove1 list.simps(9)

                                   remove1.simps(2) set_remove1_eq)
  also have "... = fun_upd_list f (xy # xys)"
    by simp
  ultimately show ?case
    by presburger
qed (fastforce)
```

6.2.2 Surjective

```
lemma helper_surj_zip_1:
  assumes "a ∈ set xs"
      and "length xs = length ys"
    shows "fun_upd_list f (zip xs ys) a ∈ set ys"
using assms
proof (induction xs arbitrary: ys)
  case (Cons x xs)
  thus ?case
    apply (cases "x = a")
     apply (metis Cons.prems(2) fun_upd_list_hd_1 length_0_conv list.distinct(1) list.exhaust_sel

                 list.set_intros(1))
    by (metis Cons.IH Cons.prems(1,2) fun_upd_list_not_hd length_Suc_conv list.set_intros(2)
set_ConsD)
qed (fastforce)

lemma fun_upd_list_surj_zip_1:
  assumes "length xs = length ys"
    shows "fun_upd_list f (zip xs ys) ` set xs ⊆ set ys"
  using assms helper_surj_zip_1 by force

lemma fun_upd_list_surj_map_1:
  "(fun_upd_list f xys) ` set (map fst xys) ⊆ set (map snd xys)"
  by (metis fun_upd_list_surj_zip_1 length_map zip_map_fst_snd)

lemma fun_upd_list_surj_map_2:
  assumes "distinct (map fst xys)"
    shows "set (map snd xys) ⊆ (fun_upd_list f xys) ` set (map fst xys)"
using assms proof (induct xys)
  case (Cons xy tail)
  { fix b assume assms_b: "b ∈ set (map snd (xy # tail))"
    hence "∃ a ∈ set (map fst (xy # tail)). (fun_upd_list f (xy # tail)) a = b"
    proof (cases "b = snd xy")
      case True
      show ?thesis using True by simp
    next
      case False
      hence 2: "∃ a ∈ set (map fst tail). (fun_upd_list f tail) a = b"
        using Cons assms_b by fastforce
```

```
    { fix aa assume 3: "aa ∈ set (map fst tail) ∧ (fun_upd_list f tail) aa = b"
      from 3 have 5: "(fun_upd_list f (xy # tail)) aa = b"
        using fun_upd_list_not_hd Cons.prems
        by (metis (no_types, lifting) distinct.simps(2) list.simps(9) zip_map_fst_snd)
      from 5 have ?thesis
        by (metis 3 list.set_intros(2) list.simps(9))
    }
    thus ?thesis using 2 by blast
  qed }
  thus ?case by blast
qed (fastforce)
```

6.2.3 Injective

```
lemma helper_inj_head:
  assumes f_def: "f = fun_upd_list f0 (zip xs ys)"
      and distinct_ys: "distinct ys"
      and length_equal: "length xs = length ys"
      and non_empty: "xs ≠ []"
      and 0: "a ∈ set xs ∧ b ∈ set xs ∧ a ≠ b"
      and 1: "a = hd xs ∧ b ∈ set (tl xs)"
    shows "f a ≠ f b"
  using assms
  by (metis distinct.simps(2) fun_upd_list_hd_1 fun_upd_list_not_hd helper_surj_zip_1 length_0_conv

            length_tl list.collapse)

lemma helper_inj_tail:
  assumes "distinct xs"
      and "distinct ys"
      and "length xs = length ys"
      and "a ∈ set (tl xs)"
      and "b ∈ set (tl xs)"
      and "a ≠ b"
    shows "fun_upd_list f (zip xs ys) a ≠ fun_upd_list f (zip xs ys) b"
using assms proof (induct xs arbitrary: ys)
  case (Cons x xs)

  have a_elem: "a ∈ set (tl (x # xs))" using Cons.prems(4) by simp
  have b_elem: "b ∈ set (tl (x # xs))" using Cons.prems(5) by simp

  have a_not_hd: "a ≠ x" using Cons.prems(1) Cons.prems(4) by force
  have b_not_hd: "b ≠ x" using Cons.prems(1) Cons.prems(5) by force



  have a: "fun_upd_list f (zip (x # xs) ys) a = fun_upd_list f (zip xs (tl ys)) a"
    using a_not_hd fun_upd_list_not_hd
    by (metis Cons.prems(3) length_0_conv list.collapse list.distinct(1))

  have uneq: "fun_upd_list f (zip xs (tl ys)) a ≠ fun_upd_list f (zip xs (tl ys)) b"
    by (metis (no_types, lifting) a_elem b_elem helper_inj_head
        Cons.hyps Cons.prems(1) Cons.prems(2) Cons.prems(3) Cons.prems(6)
        distinct_tl length_tl list.collapse list.sel(2) list.sel(3) set_ConsD)

  have b: "fun_upd_list f (zip xs (tl ys)) b = fun_upd_list f (zip (x # xs) ys) b"
    using b_not_hd fun_upd_list_not_hd
    by (metis Cons.prems(3) length_0_conv list.collapse list.distinct(1))
```

```
  from a uneq b show ?case by simp
qed (simp)

theorem fun_upd_list_inj_zip:
  assumes "distinct xs"
      and "distinct ys"
      and "length xs = length ys"
      and "xs ≠ []"
    shows "inj_on (fun_upd_list f (zip xs ys)) (set xs)"
proof-
{ fix a b assume 0: "a ∈ set xs ∧ b ∈ set xs ∧ a ≠ b"
  hence "(a = hd xs ∧ b ∈ set (tl xs)) ∨
        (b = hd xs ∧ a ∈ set (tl xs)) ∨
        (a ∈ set (tl xs) ∧ b ∈ set (tl xs))"
    using assms(4) by (metis list.collapse set_ConsD)
  moreover
  { assume "a = hd xs ∧ b ∈ set (tl xs)"
    hence "(fun_upd_list f (zip xs ys)) a ≠ (fun_upd_list f (zip xs ys)) b"
      using 0 assms helper_inj_head by metis }
  moreover
  { assume "b = hd xs ∧ a ∈ set (tl xs)"
    hence "(fun_upd_list f (zip xs ys)) a ≠ (fun_upd_list f (zip xs ys)) b"
      using 0 assms helper_inj_head by metis }
  moreover
  { assume "a ∈ set (tl xs) ∧ b ∈ set (tl xs)"
    hence "fun_upd_list f (zip xs ys) a ≠ fun_upd_list f (zip xs ys) b"
      using helper_inj_tail by (metis 0 assms(1) assms(2) assms(3)) }
  ultimately have "(fun_upd_list f (zip xs ys)) a ≠ (fun_upd_list f (zip xs ys)) b"
    by force }
  thus ?thesis by (meson inj_onI)
qed

theorem fun_upd_list_surj_zip:
  assumes "f = fun_upd_list f0 (zip xs ys)"
      and "distinct xs"
      and "length xs = length ys"
    shows "f ‘ set xs = set ys"
  by (metis assms fun_upd_list_surj_map_2 fun_upd_list_surj_zip_1
            inf.absorb_iff2 inf.order_iff zip_eq_conv)

theorem fun_upd_list_bij_betw_zip:
  assumes "distinct xs"
      and "distinct ys"
      and "length xs = length ys"
      and "xs ≠ []"
    shows "bij_betw (fun_upd_list f (zip xs ys)) (set xs) (set ys)"
  using assms
  by (fastforce simp add: bij_betw_def fun_upd_list_inj_zip fun_upd_list_surj_zip)

lemma fun_upd_list_distinct:
  assumes "distinct (map snd (xy # xys))"
      and "f x ∉ set (map snd (xy # xys))"
    shows "fun_upd_list f xys x ≠ snd xy"
  by (metis assms fun_upd_list_not_updated_map fun_upd_list_surj_map_1
            distinct.simps(2) image_eqI list.set_intros(1) list.simps(9) subsetD)

theorem inj_img_fun_upd_list_map:
```

```
  assumes "distinct (map snd xys)"
      and "∀ x. f x ∉ set (map snd xys)"
    shows "inj_img (fun_upd_list f xys) (set (map snd xys))"
using assms proof (induct xys)
  case (Cons xy xys)
  hence"inj_img (fun_upd_list f xys) ({snd xy} ∪ set (map snd xys))"
    by (fastforce simp: fun_upd_list_distinct inj_img_def)
  thus ?case
    unfolding inj_img_def apply clarsimp
    by (metis Cons.prems(1,2) fun_upd_list_distinct)
qed (fastforce simp: inj_img_not_mapped_to)

theorem inj_img_fun_upd_list_zip:
  assumes "distinct ys"
      and "length xs = length ys"
      and "∀ x. f x ∉ set ys"
    shows "inj_img (fun_upd_list f (zip xs ys)) (set ys)"
  by (metis assms inj_img_fun_upd_list_map map_snd_zip)
```

6.2.4 Set- and List-Intervals

```
lemma fun_upd_list_new_interval:
  assumes "length xs = length ys"
  shows "fun_upd_list f (zip xs ys) i ∈ {f i} ∪ set ys"
  apply (cases "i ∈ set xs")
   apply (fastforce simp: assms intro: helper_surj_zip_1)
  by (fastforce intro: fun_upd_list_not_updated_zip)


lemma helper_interval_length:
  "length [1 ..< length xs + 1] = length xs"
  apply (subst length_upt)
  by fastforce

lemma helper_interval_union:
  "{0::nat} ∪ {1 ..< n + 1} = {0 ..< n + 1}"
  by force

lemma fun_upd_list_interval:
  "fun_upd_list (λx.0) (zip xs [1 ..< length xs + 1]) z ∈ {0 ..< length xs + 1}"
  apply (cases "z ∈ set xs")
   apply (metis Un_iff set_upt helper_interval_union helper_interval_length helper_surj_zip_1)
  by (metis fun_upd_list_not_updated_zip add.commute atLeastLessThan_iff less_numeral_extra(1)

          trans_less_add1 zero_le)

theorem fun_upd_list_interval_bij:
  assumes "f = fun_upd_list (λx.0) (zip xs [1 ..< length xs + 1])"
      and "distinct xs"
    shows "bij_betw f {i. 1 ≤ f i} {1 ..< length xs + 1}"
proof-
  have set_xs : "set [1 ..< length xs + 1] = {1 ..< length xs + 1}"
    by force
  have "set xs = {i. 1 ≤ f i}"
  proof (rule antisym)
    show "set xs ⊆ {i. 1 ≤ f i}"
      by (metis One_nat_def assms(1) atLeastLessThan_iff helper_interval_length helper_surj_zip_1
```

```
              set_xs mem_Collect_eq subset_code(1))
    show "{i. 1 ≤ f i} ⊆ set xs"
        by (metis assms(1) fun_upd_list_not_updated_zip CollectD not_one_le_zero subsetI)

  qed
  thus ?thesis
    by (metis (mono_tags, lifting) assms(1,2) fun_upd_list_bij_betw_zip helper_interval_length
        One_nat_def add.right_neutral add_Suc_right bij_betwI' distinct_upt
        empty_iff empty_set le_numeral_extra(4) list.size(3)
        set_xs upt_eq_Nil_conv)
qed

end
```

6.3 Basic Definitions

```
theory Lock_Ticket

imports
  RG_Annotated_Commands
  Function_Supplementary

begin

type_synonym thread_id = nat

definition positive_nats :: "nat set" where
  "positive_nats ≡ { n. 0 < n }"
```

The state of the Ticket Lock consists of three fields.

```
record tktlock_state =
  now_serving :: "nat"
  next_ticket :: "nat"
  myticket :: "thread_id ⇒ nat"
```

Every thread locally stores a ticket number, and this collection of local variables is modelled globally by the `myticket` function.

When Thread `i` joins the queue, it sets `myticket i` to be the value `next_ticket`, and atomically increments `next_ticket`; this corresponds to the atomic Fetch-And-Add instruction, which is supported on most computer systems. Thread `i` then waits until the `now_serving` value becomes equal to its own ticket number `myticket i`. When Thread `i` leaves the queue, it increments `now_serving`.

These steps correspond to the following code for Acquire and Release. Note that we use forward function composition to model the Fetch-And-Add instruction.

```
  acquire ≡ ((myticket i := next_ticket) ∘>
              (next_ticket := next_ticket + 1));
             WHILE now_serving ≠ myticket i DO SKIP OD)

  release ≡ now_serving := now_serving + 1
```

Conceptually, Thread `i` is in the queue if and only if `now_serving ≤ myticket i` and is at the head if and only if `now_serving = myticket i`.

Now, in the initial state, every thread holds the number 0 as its ticket, and both now_serving and next_ticket are set to 1.

```
abbreviation tktlock_init :: "tktlock_state set" where
  "tktlock_init ≡ {| ´myticket = (λj. 0) ∧
    ´now_serving = 1 ∧ ´next_ticket = 1 |}"
```

We further define a shorthand for describing the set of ticket in use; i.e. those numbers from now_serving up to, but not including next_ticket. This shorthand will later be used in the invariant.

```
abbreviation tktlock_contending_set :: "tktlock_state ⇒ thread_id set" where
  "tktlock_contending_set s ≡ { j. now_serving s ≤ myticket s j }"
```

We now formalise the invariant of the Ticket Lock.

```
abbreviation tktlock_inv :: "tktlock_state set" where
  "tktlock_inv ≡ {| ´now_serving ≤ ´next_ticket ∧
    1 ≤ ´now_serving ∧
    (∀ j. ´myticket j < ´next_ticket) ∧
    bij_betw ´myticket ´tktlock_contending_set {´now_serving ..< ´next_ticket} ∧
    inj_img ´myticket positive_nats |}"
```

The first three clauses are basic inequalities.

The penultimate clause stipulates that the function myticket of every valid state is bijective between the set of queuing/contending threads (those threads whose tickets are not smaller than now_serving) and .

The final clause ensures that the function myticket is injective when 0 is excluded from its codomain. In other words, all threads, whose tickets are non-zero, hold unique tickets.

As for the contract, the first clause ensures that the local variable myticket i does not change. Meanwhile, the global variables next_ticket and now_serving must not decrease, as stipulated by the second and third clauses of the contract.

The last two clauses of the contract correspond to the two clauses of the contract of the Abstract Queue Lock, where i ∈ set queue and at_head i queue under the Abstract Queue Lock respectively translate to now_serving ≤ myticket i and now_serving = myticket i under the Ticket Lock.

```
abbreviation tktlock_contract :: "thread_id ⇒ tktlock_state rel" where
  "tktlock_contract i ≡ {| ºmyticket i = ªmyticket i ∧
    ºnext_ticket ≤ ªnext_ticket ∧
    ºnow_serving ≤ ªnow_serving ∧
    (ºnow_serving ≤ ºmyticket i ⟷ ªnow_serving ≤ ªmyticket i) ∧
    (ºnow_serving = ºmyticket i ⟶ ªnow_serving = ªmyticket i) |}"
```

We further state and prove some helper lemmas that will be used later.

```
lemma tktlock_contending_set_rewrite:
  "tktlock_contending_set s ∪ {i} = {|´(≠) i ⟶ now_serving s ≤ ´(myticket s)|}"
  by fastforce

lemma tktlock_used_tickets_rewrite:
  assumes "now_serving s ≤ next_ticket s"
    shows "{now_serving s ..< next_ticket s} ∪ {next_ticket s}
         = {now_serving s ..< Suc (next_ticket s)}"
  by (fastforce simp: assms atLeastLessThanSuc)

lemma tktlock_enqueue_bij:
  assumes "myticket s i < now_serving s"
```

```
      and "bij_betw (myticket s) (tktlock_contending_set s) {now_serving s ..< next_ticket
s}"
    shows "bij_betw ( (myticket s)(i := next_ticket s) )
                    ( tktlock_contending_set s ∪ {i} )
                    ( {now_serving s ..< next_ticket s} ∪ {next_ticket s} )"
  apply (rule bij_extension)
  using assms by fastforce+

lemma tktlock_enqueue_inj:
  assumes "s ∈ tktlock_inv"
  shows "inj_img ((myticket s)(i := next_ticket s)) positive_nats"
  apply(subst inj_img_fun_upd_notin)
  using assms by (fastforce simp: nat_less_le)+

method clarsimp_seq = clarsimp, standard, clarsimp
```

6.4 RG Theorems

The RG sentence of the first instruction of Acquire.

```
lemma tktlock_acq1:
  "rely: tktlock_contract i  guar: for_others tktlock_contract i
   inv:  tktlock_inv    anno_code:
   { ⦃ ´myticket i < ´now_serving ⦄ }
     BasicAnno ((´myticket[i] ← ´next_ticket) ∘>
                (´next_ticket ← ´next_ticket + 1))
   { ⦃ ´now_serving ≤ ´myticket i ⦄ }"
proof method_anno_ultimate
  case est_guar
  thus ?case
    apply clarsimp_seq
     apply (fastforce simp: less_Suc_eq)
    using tktlock_contending_set_rewrite tktlock_enqueue_bij
    by (fastforce simp: atLeastLessThanSuc tktlock_enqueue_inj)
next
  case est_post
  thus ?case
    apply clarsimp_seq
     apply (fastforce simp: less_Suc_eq)
    using tktlock_contending_set_rewrite tktlock_enqueue_bij
    by (fastforce simp: atLeastLessThanSuc tktlock_enqueue_inj)
qed (fastforce)+
```

A helper lemma for the Release procedure.

```
lemma tktlock_rel_helper:
  assumes inv1: "now_serving s = myticket s i"
      and inv2: "myticket s i ≤ next_ticket s"
      and inv3: "Suc 0 ≤ myticket s i"
      and inv4: "∀j. myticket s j < next_ticket s"
      and bij_old: "bij_betw (myticket s)
                          ⦃myticket s i ≤ ´(myticket s)⦄
                          {myticket s i ..< next_ticket s}"
    shows "bij_betw (myticket s)
                    ⦃Suc (myticket s i) ≤ ´(myticket s)⦄
                    {Suc (myticket s i) ..< next_ticket s}"
proof -
  have thread_rewrite:
    "⦃ Suc (myticket s i) ≤ ´(myticket s)⦄ = {j. myticket s i ≤ myticket s j} - {i}"
```

```
    apply (subst set_remove_one_element[where B="{j. Suc (myticket s i) ≤ myticket s j}"];
clarsimp)
    by(metis CollectI Suc_leI assms(5) bij_betw_def inj_onD order_le_imp_less_or_eq)
  have ticket_rewrite:
    "{Suc (myticket s i) ..< next_ticket s} = {myticket s i ..< next_ticket s} - {myticket
s i}"
    by fastforce
  have "bij_betw (myticket s)
                 ( {j. myticket s i ≤ myticket s j} - {i} )
                 ( {myticket s i ..< next_ticket s} - {myticket s i} )"
    by (rule bij_remove_one; clarsimp simp: bij_old)
  thus ?thesis
    by (clarsimp simp: thread_rewrite ticket_rewrite)
qed
```

The RG sentence for the Release procedure.

```
lemma tktlock_rel:
  "rely: tktlock_contract i
   guar: for_others tktlock_contract i
   inv:  tktlock_inv

   code: { ⦃ ´now_serving = ´myticket i ⦄ }
           ´now_serving := ´now_serving + 1
         { ⦃ ´myticket i < ´now_serving ⦄ }"
proof method_basic_inv
  case est_inv
  thus ?case
    by (clarsimp, fastforce simp: Suc_le_eq intro!: tktlock_rel_helper)
next
  case est_guar
  thus ?case
    by (clarsimp, fastforce simp: less_eq_Suc_le nat_less_le positive_nats_def inj_img_def)
qed (fastforce)+
```

The RG sentence for a thread that performs Acquire and then Release.

```
lemma tktlock_local:
 "rely: tktlock_contract i  guar: for_others tktlock_contract i
  inv: tktlock_inv     anno_code:

   { ⦃ ´myticket i < ´now_serving ⦄ }
   BasicAnno ((´myticket[i] ← ´next_ticket) ∘>
              (´next_ticket ← ´next_ticket + 1)) .;
   { ⦃ ´now_serving ≤ ´myticket i ⦄ }
   NoAnno (WHILE ´now_serving ≠ ´myticket i DO SKIP OD) .;
   { ⦃ ´now_serving = ´myticket i ⦄ }
   NoAnno (´now_serving := ´now_serving + 1)
   { ⦃ ´myticket i < ´now_serving ⦄ }"
  apply (method_anno_ultimate, goal_cases)
    using tktlock_acq1 apply fastforce
   apply (clarsimp, method_spinloop; fastforce)
  using tktlock_rel by fastforce
```

The RG sentence for a thread that repeatedly performs Acquire and then Release in an infinite loop.

```
lemma tktlock_local_loop:
 "rely: tktlock_contract i  guar: for_others tktlock_contract i
  inv:  tktlock_inv    anno_code:
```

```
   { ⦃ ´myticket i < ´now_serving ⦄ }
   WHILEa True DO
    {stable_guard: ⦃ ´myticket i < ´now_serving ⦄ }
     BasicAnno ((´myticket[i] ← ´next_ticket) ∘>
                (´next_ticket ← ´next_ticket + 1)) .;
     { ⦃ ´now_serving ≤ ´myticket i ⦄ }
     NoAnno (WHILE ´now_serving ≠ ´myticket i DO SKIP OD) .;
     { ⦃ ´now_serving = ´myticket i ⦄ }
     NoAnno (´now_serving := ´now_serving + 1)
   OD
   { ⦃ ´myticket i < ´now_serving ⦄ }"
proof method_anno_ultimate
  case body
  thus ?case
    using tktlock_local by (fastforce simp: Int_commute)
qed (fastforce)+
```

The global RG sentence for a set of threads, each of which repeatedly performs Acquire and then Release in an infinite loop.

```
theorem tktlock_global:
  assumes "0 < n"
    shows "annotated
  global_init: ⦃ ´now_serving = 1 ∧ ´next_ticket = 1 ∧ ´myticket = (λj. 0) ⦄
  global_rely: Id
    ∥ i < n @

  { ⦃ ´myticket i < ´now_serving ⦄, tktlock_contract i }
  WHILEa True DO
    {stable_guard: ⦃ ´myticket i < ´now_serving ⦄ }
    BasicAnno ((´myticket[i] ← ´next_ticket) ∘>
               (´next_ticket ← ´next_ticket + 1)) .;
    { ⦃ ´now_serving ≤ ´myticket i ⦄ }
    NoAnno (WHILE ´now_serving ≠ ´myticket i DO SKIP OD) .;
    { ⦃ ´now_serving = ´myticket i ⦄ }
    NoAnno (´now_serving := ´now_serving + 1)
  OD

  // tktlock_inv { for_others tktlock_contract i, {} }
  global_guar: UNIV
  global_post: {}"
proof method_anno_ultimate
  case (local_sat i)
  thus ?case using tktlock_local_loop by fastforce
next
  case (pre i)
  thus ?case
    using bij_betwI' inj_img_def positive_nats_def by fastforce
next
  case (guar_imp_rely i j)
  thus ?case
    by auto[1]
qed (fastforce simp: assms)+

end
```

7 Circular-Buffer Queue-Lock

This theory imports Annotated Commands to access the rely-guarantee library extensions, and also imports the Abstract Queue Lock to access the definitions of the type-synonym `thread_id` and the abbreviation `at_head`.

```
theory Lock_Circular_Buffer

imports
  RG_Annotated_Commands
  Lock_Abstract_Queue

begin

type_synonym index = nat

datatype flag_status = Pending | Granted
```

We assume a fixed number of threads, and the size of the circular array is 1 larger the number of threads.

```
consts NumThreads :: nat

abbreviation ArraySize :: "nat" where
  "ArraySize ≡ NumThreads + 1"
```

The state of the Circular Buffer Lock consists of the following fields:

- `myindex`: a function that maps each thread to an array-index (where the array is modelled by `flag_mapping` below).
- `flag_mapping`: an array of size `ArraySize` that stores values of type `flag_status`.
- `tail`: an index representing the tail of the queue, used when a thread enqueues.
- `aux_head`: an auxiliary variable that stores the index used by the thread at the head of the queue; the head of the queue spins on the flag `flag_mapping aux_head`.
- `aux_queue`: the auxiliary queue of threads.
- `aux_mid_release`: an auxiliary variable that signals if a thread has executed the first instruction of `release`, but not the second.

```
record cblock_state =
  myindex :: "thread_id ⇒ index"
  flag_mapping :: "index ⇒ flag_status"
  tail :: index
  aux_head :: index
  aux_queue :: "thread_id list"
  aux_mid_release :: "thread_id option"
```

We initialise the array of flags (`flag_mapping`) with `Granted` in the zeroth entry and `Pending` in all other entries. The indices `tail` and `aux_head` are initialised to 0. The queue is initially empty, and no thread is in the middle of `release`. (See the conference article for an example.)

```
definition cblock_init :: "cblock_state set" where
  "cblock_init ≡ {|
    ´flag_mapping = (λ _. Pending)(0 := Granted) ∧
    ´tail = 0 ∧
```

```
    ´aux_queue = [] ∧
    ´aux_head = 0 ∧
    ´aux_mid_release = None ⦄"
```

Similar to the Abstract Queue Lock, the `acquire` procedure of the Circular Buffer Lock consists of two conceptual steps, and corresponds to the pseudocode below. (1) To join the queue, Thread `i` stores the global index `tail` locally as `myindex i`, and atomically increments `tail` modulo the array size. (2) Thread `i` then spins on its flag, which is the entry in the array at index `myindex i`. When this flag changes from `Pending` to `Granted`, the thread has reached the head of the queue.

```
acquire ≡ ((myindex i := tail) ∘>
            (tail := (tail + 1) mod ArraySize));
           WHILE flag_mapping (myindex i) = Pending DO SKIP OD
```

When Thread `i` releases the lock, it sets its flag to `Pending`. Then it sets the flag of the next thread to `Granted`, which corresponds to the 'next' entry in the array, modulo the array size. This is encoded as the pseudocode below.

```
release ≡ flag_mapping[myindex i] := Pending ;
           flag_mapping[(myindex i + 1) mod ArraySize] := Granted
```

Auxiliary Variables. The `release` procedure consists of the single conceptual step of exiting the queue, but is implemented here as two separate instructions. Hence, the auxiliary variable `aux_mid_release` indicates when a thread is between the two lines of `release`, and allows us to express the assertion there.

The other two auxiliary variables, `aux_head` (the *head-index*) and `aux_queue`, store information that can in principle be inferred from the concrete variables (i.e. the non-auxiliary variables). However, explicitly recording this information as auxiliary variables greatly simplifies the verification process.

In the code, these auxiliary variables need to be updated atomically with the relevant instructions. Below is the code of `release` with the auxiliary variables included. (Auxiliary variables are added to `acquire` in a similar way.)

```
release ≡ ⟨ flag_mapping[myindex i] := Pending ∘>
            aux_mid_release := Some i ⟩ ;
          ⟨ flag_mapping[(myindex i + 1) mod ArraySize] := Granted ∘>
            aux_queue := tl aux_queue ∘>
            aux_head := (aux_head + 1) mod ArraySize ∘>
            aux_mid_release := None ⟩
```

Recall that we assume a fixed number of threads. This constant is furthermore assumed positive, which we enforce with the use of the following locale.

locale `numthreads_positive =`
  assumes `assm_locale: "0 < NumThreads"`
begin

7.1 Invariant

A notion that helps us state the queue-clause of the invariant. The list of indices use by the queuing threads is a contiguous list of integers modulo ArraySize. Note the possibility of “wrapping around”, which is covered by the “else” clause in the definition.

```
definition used_indices :: "cblock_state ⇒ index list" where
  "used_indices s ≡ (if aux_head s ≤ tail s
     then [aux_head s ..< tail s]
     else [aux_head s ..< ArraySize] @ [0 ..< tail s])"

lemma distinct_used_indices: "distinct (used_indices s)"
  using used_indices_def by fastforce

lemma length_used_indices:
  "length (used_indices s) = (if aux_head s ≤ tail s
     then tail s - aux_head s
     else ArraySize - aux_head s + tail s)"
  using used_indices_def by force
```

The invariant of the Circular Buffer Lock is stated as separate parts below. The first definition invar_flag relates flag_mapping with the head-index aux_head, and consists of two clauses. (1) At every index that is not the head-index, the flag must be Pending. (2) As for the head-index itself, there are two possibilities. When the thread at the head of the queue invoked release but has only executed its first instruction, aux_mid_release becomes set to Some i; in this case, the flag at the head-index is set to Pending, but the thread remains in the queue. In all other cases, aux_mid_release = None, and the flag at the head-index is always Granted.

```
definition invar_flag :: "cblock_state set" where
  "invar_flag ≡ {|
    (∀ i ≠ ´aux_head. ´flag_mapping i = Pending) ∧
    (´flag_mapping ´aux_head = Pending ⟷ ´aux_mid_release ≠ None) |}"
```

The next clause invar_queue describes the relationship between the auxiliary queue and the other variables, including the set used_indices. The clause involving map further implies a number of properties, such as the distinctness of aux_queue (which mirrors the invariant of the Abstract Queue Lock), and the injectivity of myindex (i.e. each queuing thread has a unique index).

```
definition invar_queue :: "cblock_state set" where
  "invar_queue ≡ {|
    (∀ i. i ∈ set ´aux_queue ⟶ i < NumThreads) ∧
    (map ´myindex ´aux_queue = ´used_indices) |}"
```

The overall invariant, cblock_invar, is the conjunction of invar_flag and invar_queue above, with additional inequalities concerning tail, aux_head, and NumThreads.

```
definition invar_bounds :: "cblock_state set" where
  "invar_bounds ≡ {|
        ´tail < ArraySize ∧
    ´aux_head < ArraySize |}"

abbreviation cblock_invar :: "thread_id ⇒ cblock_state set" where
  "cblock_invar i ≡
    invar_flag ∩ invar_bounds ∩ invar_queue ∩ {| i < NumThreads |}"

lemmas cblock_invariants =
  invar_flag_def
  invar_bounds_def
```

```
  invar_queue_def
  used_indices_def
```

7.1.1 Invariant Methods

We set up methods that generate structured proofs with named subgoals, to help us prove the clauses of the invariant.

```
theorem thm_method_invar_flag:
  assumes "∀ i ≠ aux_head s. flag_mapping s i = Pending"
      and "flag_mapping s (aux_head s) = Pending
           ⟷ aux_mid_release s ≠ None"
    shows "s ∈ invar_flag"
  using assms invar_flag_def by force

method method_invar_flag =
  cases rule:thm_method_invar_flag,
  goal_cases non_head_pending head_maybe_granted

theorem thm_method_invar_queue:
  assumes "∀ i. i ∈ set (aux_queue s) ⟶ i < NumThreads"
      and "map (myindex s) (aux_queue s) = (used_indices s)"
    shows "s ∈ invar_queue"
  using assms invar_queue_def by force

method method_invar_queue =
  cases rule:thm_method_invar_queue,
  goal_cases bound_thread_id map_used_indices

theorem thm_method_invar:
  assumes flag:  "s ∈ invar_flag"
      and bound: "s ∈ invar_bounds ∧ i < NumThreads"
      and queue: "s ∈ invar_queue"
    shows "s ∈ cblock_invar i"
  using assms by fastforce

method method_cblock_invar =
  cases rule:thm_method_invar,
  goal_cases flag bound queue
```

7.1.2 Invariant Lemmas

The initial state satisfies the invariant.

```
lemma cblock_init_invar:
  assumes assm_init:  "s ∈ cblock_init"
      and assm_bound: "i < NumThreads"
    shows "s ∈ cblock_invar i"
proof method_cblock_invar
  case flag
  thus ?case
    using assms
    by (method_invar_flag; force simp: cblock_init_def)
next
  case bound
  thus ?case
    using assms
    by (force simp: assm_locale cblock_init_def invar_bounds_def)
next
```

```
  case queue
  thus ?case
    using assms
    by (method_invar_queue; force simp: cblock_init_def used_indices_def)
qed
```

In a state that satisfies the flag-invariant, a thread is the head of the queue if its flag is Granted. (If the flag of a thread is Pending, the thread may still be at the head of the queue. In this case, the thread must be between the two instructions in `release`.)

```
lemma only_head_is_granted:
  assumes "s ∈ invar_flag"
      and "flag_mapping s i = Granted"
    shows "i = aux_head s"
  using assms by (force simp: invar_flag_def)
```

Let `s` be a state that satisfies the bounds-invariant, with n queuing threads. If we start from the `aux_head` index, and "advance" n steps (with potential wrap-around), then we reach the global `tail` index.

```
lemma head_tail_mod:
  "s ∈ invar_bounds ⟹
    tail s = (aux_head s + length (used_indices s)) mod (ArraySize)"
  by (fastforce simp: mod_if used_indices_def invar_bounds_def)
```

If a state satisfies the queue-invariant (namely the clause with the `map` function, then the `myindex` function is injective on the set of queuing threads. In other words, every queuing thread has a unique index in a state that satisfies the queue-invariant.

```
lemma invar_map_inj_on:
  "s ∈ invar_queue ⟹ inj_on (myindex s) (set (aux_queue s))"
  using distinct_map
  by (fastforce simp: invar_queue_def distinct_used_indices)
```

In a state that satisfies the queue-invariant, the length of the queue is equal to the length of the list of used indices.

```
lemma used_indices_map_queue:
  "s ∈ invar_queue ⟹ used_indices s = map (myindex s) (aux_queue s)"
  unfolding used_indices_def invar_queue_def used_indices_def
  by clarsimp

lemma length_used_indices_queue:
  "s ∈ invar_queue ⟹ length (used_indices s) = length (aux_queue s)"
  by (fastforce simp: used_indices_map_queue)
```

In a state that fully satisfies the invariant, if there is a thread that is not in the queue, then the length of the queue must be smaller than the total number of threads.

```
lemma queue_bounded:
  assumes "s ∈ cblock_invar i"
      and "i ∉ set (aux_queue s)"
    shows "length (aux_queue s) < NumThreads"
proof-
  have "length (used_indices s) ≤ NumThreads"
    using assms(1)
    by (fastforce simp: invar_bounds_def length_used_indices )
  hence "card (set (aux_queue s)) ≤ NumThreads"
    using assms(1)
    by (fastforce intro: le_trans intro!:  card_length simp: length_used_indices_queue)
  moreover have "card (set (aux_queue s)) = 0 ⟷ aux_queue s = []"
```

```
    by fastforce
  moreover have "finite (set (aux_queue s))"
    using calculation by fastforce
  moreover have "card (set (aux_queue s)) = NumThreads
                ⟷ (∀ j < NumThreads. j ∈ set (aux_queue s))"
  proof-
    { assume "card (set (aux_queue s)) = NumThreads"
      hence "set (aux_queue s) = {j. j < NumThreads}"
        using assms by (force simp add: invar_queue_def card_subset_eq subsetI)
      hence "∀ j < NumThreads. j ∈ set (aux_queue s)"
        by blast }
    moreover
    { assume "∀ j < NumThreads. j ∈ set (aux_queue s)"
      hence "card (set (aux_queue s)) = NumThreads"
        using assms by fastforce }
    ultimately
      show ?thesis by blast
  qed

  ultimately have "card (set (aux_queue s)) < NumThreads"
    using assms nat_less_le by blast

  thus ?thesis
    using assms
    by (metis used_indices_map_queue Int_iff distinct_card distinct_map distinct_used_indices)
qed
```

If a state that satisfies the bound- and queue-invariants, and if the queue is non-empty, then the index held by the head of the queue must be the same as `aux_head`.

```
lemma head_and_head_index:
  assumes "s ∈ invar_bounds ∩ invar_queue"
      and "aux_queue s ≠ []"
    shows "myindex s (hd (aux_queue s)) = aux_head s"
proof-
  have "myindex s (hd (aux_queue s)) = hd (used_indices s)"
    using assms
    by (simp add: used_indices_map_queue hd_map)
  also have "... = aux_head s"
    using assms
    by (fastforce simp: invar_queue_def invar_bounds_def upt_rec used_indices_def)
  ultimately show ?thesis
    by fastforce
qed
```

In a state that satisfies the full invariant, if no thread is half-way through *release* and Thread `i` is at the head of the queue, then the flag of Thread `i` must be Granted.

```
lemma head_is_granted:
  assumes "s ∈ cblock_invar i"
      and "aux_mid_release s = None"
      and "i = hd (aux_queue s)"
      and "aux_queue s ≠ []"
    shows "flag_mapping s (myindex s i) = Granted"
proof-
  have "myindex s i = aux_head s"
    using assms by (fastforce intro: head_and_head_index)
  thus ?thesis
    using assms
    by (fastforce intro: flag_status.exhaust simp: invar_flag_def)
```

qed

In a state that satisfies the queue-invariant, the global index `tail` is never held by a thread. Indeed, `tail` is meant to be "free" for the next thread that joins the queue. Note that when a thread is not in the queue, its index `i` becomes outdated, and `tail` may cycle back and coincide with `i`.

```
lemma tail_never_used:
  assumes "s ∈ invar_queue"
    shows "∀ j ∈ set (aux_queue s). myindex s j ≠ tail s"
proof-
  have "tail s ∉ set (used_indices s)"
    unfolding used_indices_def by clarsimp
  thus ?thesis
    unfolding invar_queue_def
    by (fastforce simp: assms used_indices_map_queue rev_image_eqI)
qed
```

In a state that satisfies the full invariant, if the `tail` index is right before the `aux_head` index, then it must be the case that every thread is in the queue.

```
lemma used_indices_full:
  assumes "s ∈ cblock_invar i"
      and "(tail s + 1) mod ArraySize = aux_head s"
    shows "length (used_indices s) = NumThreads"
  using assms
  apply (clarsimp simp: used_indices_def)
  apply (intro conjI impI)
     apply (metis Suc_eq_plus1 add_diff_cancel_left' diff_zero head_tail_mod le_add_diff_inverse

                  length_used_indices lessI linorder_not_le mod_Suc plus_1_eq_Suc)
    apply (fastforce simp: Suc_diff_le)
   by (metis mod_Suc_le_divisor)+
```

Conversely, if not every thread is in the queue, then the `tail` index is not right before the `aux_head` index.

```
lemma space_available:
  assumes assm_invar: "s ∈ cblock_invar i"
      and assm_q: "i ∉ set (aux_queue s)"
    shows "(tail s + 1) mod ArraySize ≠ aux_head s"
  using assms queue_bounded length_used_indices_queue
  by (fastforce simp: used_indices_full)
```

The next lemma relates the *append* operation on the `aux_head` and `tail` indices to the *append* operation on the list of `used_indices`. (The second and the last assumptions are the most crucial ones. The rest are side-condition checks.)

```
lemma used_indices_append:
  assumes "s ∈ cblock_invar i"
      and "aux_head s' = aux_head s"
      and "length (used_indices s) < NumThreads"
      and "(tail s + 1) mod ArraySize ≠ aux_head s"
      and "tail s' = (tail s + 1) mod ArraySize"
    shows "used_indices s' = used_indices s @ [tail s]"
proof (cases "aux_head s' ≤ tail s'")
  case True
  hence ln1: "tail s' = (tail s + 1)"
    using assms apply clarsimp
    by (metis Suc_eq_plus1 bot_nat_0.extremum_unique head_tail_mod mod_Suc mod_mod_trivial)
```

```
  thus ?thesis
    using assms used_indices_def ln1 by fastforce
next
  case False
  hence a: "¬ aux_head s' ≤ tail s'" .
  thus ?thesis
  proof (cases "tail s' = 0")
    case True
    thus ?thesis
      using assms apply clarsimp
      by (metis (no_types, lifting) Suc_eq_plus1 Suc_lessI Zero_not_Suc append.right_neutral

                assms(5) invar_bounds_def linorder_not_le mem_Collect_eq mod_less upt_Suc

                upt_eq_Nil_conv used_indices_def)
  next
    case False
    thus ?thesis
    proof -
      have "tail s < tail s'"
        using assms(1) assms(5) apply clarsimp
        by (metis False Suc_eq_plus1 head_tail_mod lessI mod_Suc mod_mod_trivial)
      thus ?thesis
        by (metis a used_indices_def assms(2,5) Suc_eq_plus1 append.assoc less_Suc_eq_le

                mod_less_eq_dividend not_less_eq order_less_le upt_Suc_append zero_less_Suc)
    qed
  qed
qed
```

7.2 Contract

The contract of the Circular Buffer Lock is devised along three observations: (1) local variables do not change; (2) global variables may change; and (3) auxiliary variables change similarly as in the Abstract Queue Lock.

The first two areas are covered by `contract_raw`. The only local variable `myindex i` does not change. The global variable `tail` may change, but is not included in the contract, as changes to `tail` are not restricted. However, the other global variable `flag_mapping` is allowed to change only in specific ways. As `flag_mapping` stores information about the head of the conceptual queue, its allowed changes naturally relate to the *head stays the head* property. Under the Circular Buffer Lock, Thread `i` is at the head of the queue when `flag_mapping (myindex i) = Granted`. Meanwhile, note that `myindex i` can become outdated if Thread `i` is not in the queue. Hence, we need the premise `i ∈ set ºaux_queue` before the *head stays the head* statement in the final clause of `contract_raw`.

```
definition contract_raw :: "thread_id ⇒ cblock_state rel" where
  "contract_raw i ≡ {|
    (i ∈ set ºaux_queue
     ⟶ ºflag_mapping (ºmyindex i) = Granted
     ⟶ ªflag_mapping (ªmyindex i) = Granted) ∧
    (ºmyindex i = ªmyindex i) |}"
```

For the auxiliary variable `aux_queue` we require the same two clauses as in the contract of the Abstract Queue Lock. As for `aux_mid_release`, only the head of the queue can invoke `release` and hence modify `aux_mid_release`. Therefore, the second clause of `contract_aux` has the extra equality in the consequent.

```
definition contract_aux :: "thread_id ⇒ cblock_state rel" where
  "contract_aux i ≡ ⦃
    (i ∈ set ºaux_queue ⟷ i ∈ set ªaux_queue) ∧
    (at_head i ºaux_queue ⟶ at_head i ªaux_queue ∧ ºaux_mid_release = ªaux_mid_release)
⦄"
```

The two definitions above combine into the overall contract.

```
abbreviation cblock_contract :: "thread_id ⇒ cblock_state rel" where
  "cblock_contract t ≡ contract_raw t ∩ contract_aux t"

lemmas cblock_contracts[simp] = contract_raw_def contract_aux_def
```

7.3 RG Lemmas

```
abbreviation acq_line1 :: "thread_id ⇒ cblock_state ⇒ cblock_state" where
  "acq_line1 i ≡
    (´myindex[i] ← ´tail) ∘>
    (´tail ← (´tail + 1) mod ArraySize) ∘>
    (´aux_queue ← ´aux_queue @ [i])"

lemma acq_1_invar:
  assumes assm_old: "s ∈ cblock_invar i"
      and assm_new: "s' = acq_line1 i s"
      and assm_pre: "i ∉ set (aux_queue s)"
    shows "s' ∈ cblock_invar i"
proof method_cblock_invar
  case flag
  have "(∀ j ≠ aux_head s. flag_mapping s j = Pending) ∧
        (flag_mapping s (aux_head s) = Pending ⟷ aux_mid_release s ≠ None)"
    using assm_old by (fastforce simp: invar_flag_def)
  hence "(∀ j ≠ aux_head s'. flag_mapping s' j = Pending) ∧
         (flag_mapping s' (aux_head s') = Pending ⟷ aux_mid_release s' ≠ None)"
    using assm_new by fastforce
  thus ?case
    by (fastforce simp: invar_flag_def)
next
  case bound
  have "aux_head s' < ArraySize"
    using assm_old assm_new by (fastforce simp: invar_bounds_def)
  moreover have "tail s' < ArraySize"
    using assm_new by fastforce
  ultimately show ?case
    using assm_old assm_new by (fastforce simp: invar_bounds_def)
next
  case queue show ?case
  proof method_invar_queue
    case bound_thread_id
    have "∀ j. j ∈ set (aux_queue s) ⟶ j < NumThreads"
      using assm_old assm_new by (fastforce simp: invar_queue_def)
    moreover have "set (aux_queue s') = set (aux_queue s) ∪ {i}"
      using assm_new by fastforce
    moreover have "i < NumThreads"
      using assm_old assm_new by fastforce
    ultimately show ?case
      by fastforce
  next
    case map_used_indices
    have "map (myindex s') (aux_queue s') = map (myindex s) (aux_queue s) @ [myindex s'
```

```
i]"
      using assm_new assm_pre by fastforce
    also have ln1: "... = used_indices s @ [myindex s' i]"
      using assm_old by (fastforce simp: used_indices_def invar_queue_def)
    also have "... = used_indices s @ [tail s]"
      using assm_new by fastforce
    also have "... = used_indices s'"
    proof-
      have ahead: "aux_head s = aux_head s'"
        using assms by fastforce
      have "length (used_indices s) < NumThreads"
        using assm_pre assm_old
        by (fastforce simp: length_used_indices_queue queue_bounded)
      moreover have "(tail s + 1) mod ArraySize ≠ aux_head s"
        using assm_old assm_pre space_available by fastforce
      moreover have "tail s' = (tail s + 1) mod (ArraySize)"
        using assm_new by simp
      ultimately show ?thesis
        by (metis ahead assm_old used_indices_append)
    qed
    ultimately show ?case by fastforce
  qed
qed

theorem cblock_acq1:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ i ∉ set ´aux_queue ⦄ }
  BasicAnno (acq_line1 i)
    { ⦃ i ∈ set ´aux_queue ⦄ }"
  apply method_anno_ultimate
    using acq_1_invar by fastforce+

theorem cblock_acq2:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i       code:
    { ⦃ i ∈ set ´aux_queue ⦄ }
  WHILE ´flag_mapping (´myindex i) = Pending DO SKIP OD
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }"
proof method_spinloop
  case est_post
  thus ?case
  proof-
    { fix s assume assm_s: "s ∈ cblock_invar i ∩ ⦃ i ∈ set ´aux_queue ⦄ ∩
                            ⦃ ´flag_mapping (´myindex i) ≠ Pending ⦄"
      hence ln1:"aux_queue s ≠ []"
        by force
      have ln2:"flag_mapping s (aux_head s) ≠ Pending"
        using assm_s invar_flag_def by force
      hence ln3:"myindex s i = aux_head s"
        using assm_s invar_flag_def
        by (metis (mono_tags, lifting) IntE mem_Collect_eq)
      have "i = hd (aux_queue s) ∧ s ∈ ⦃ ´aux_mid_release = None ⦄"
        apply (intro conjI)
         using ln1 ln3 assm_s
         apply (metis (lifting) Int_Collect head_and_head_index inf_commute inj_onD invar_bounds_def
                                invar_flag_def invar_map_inj_on invar_queue_def list.set_sel(1))
```

```
        using ln2 ln3  assm_s
        by (fastforce simp: invar_flag_def flag_status.exhaust)
    }
    thus ?thesis by fastforce
  qed
qed (fastforce+)


abbreviation rel_line1 :: "thread_id ⇒ cblock_state ⇒ cblock_state" where
  "rel_line1 i ≡ (´flag_mapping[´myindex i] ← Pending) ○>
                 (´aux_mid_release ← Some i)"

lemma rel_1_same:
  "s' = rel_line1 i s ⟹
    (myindex s = myindex s') ∧
    (∀ j ≠ myindex s i. flag_mapping s j = flag_mapping s' j) ∧
    (tail s = tail s') ∧
    (aux_head s = aux_head s') ∧
    (aux_queue s = aux_queue s')"
  by simp

lemma rel_1_invar:
  assumes assm_old: "s ∈ cblock_invar i"
      and assm_new: "s' = rel_line1 i s"
      and assm_pre: "at_head i (aux_queue s) ∧ aux_mid_release s = None"
    shows "s' ∈ cblock_invar i"
proof method_cblock_invar
  case flag show ?case
    apply method_invar_flag
     using  assm_new assm_old assm_pre
     by (fastforce simp: invar_flag_def head_and_head_index)+
next
  case bound
  thus ?case
    using  assm_old invar_bounds_def assm_new
    by (metis (no_types, lifting) rel_1_same Int_iff mem_Collect_eq)
next
  case queue show ?case
    apply (method_invar_queue)
    using assm_old assm_new apply (fastforce simp: invar_queue_def)
    by (metis (lifting) assm_old assm_new used_indices_map_queue used_indices_def rel_1_same
IntE)
qed

lemma rel_1_est_guar:
  assumes "s ∈ {| ´aux_queue ≠ [] ∧
                 hd ´aux_queue = i ∧
                ´aux_mid_release = None |}
              ∩ cblock_invar i"
      and "s' = rel_line1 i s"
    shows "(s, s') ∈ for_others cblock_contract i
                ∩ pred_to_rel (cblock_invar i)"
proof-
  { fix j assume assm_u_t: "j ≠ i"
    have "j ∈ set (aux_queue s)
              ⟶ flag_mapping s  (myindex s  j) = Granted
              ⟶ flag_mapping s' (myindex s' j) = Granted"
      using assms assm_u_t
```

```
      by (fastforce intro: simp: head_and_head_index inj_onD dest: invar_map_inj_on)
    moreover have "myindex s j = myindex s' j"
      using assms by (fastforce intro: rel_1_same)
    ultimately have "(s, s') ∈ contract_raw j"
      by fastforce }
  moreover
  { fix j assume "j ≠ i"
    hence "hd (aux_queue s) ≠ j"
      using assms(1) by simp
    moreover have "j ∈ set (aux_queue s) ⟷ j ∈ set (aux_queue s')"
      using rel_1_same assms(2) by simp
    ultimately have "(s, s') ∈ contract_aux j"
      by fastforce }
  moreover have "(s, s') ∈ pred_to_rel (cblock_invar i)"
    using assms rel_1_invar by fastforce
  ultimately show ?thesis
    by fastforce
qed

theorem cblock_rel1:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }
  BasicAnno (rel_line1 i)
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }"
proof method_anno_ultimate
  case est_guar
  thus ?case
    using rel_1_invar
    by (fastforce dest: invar_map_inj_on simp: inj_on_contraD)
next
  case est_post
  thus ?case
    using rel_1_est_guar by fastforce
qed (fastforce+)

abbreviation rel_line2 :: "thread_id ⇒ cblock_state ⇒ cblock_state" where
  "rel_line2 i ≡
    (´flag_mapping[((´myindex i + 1) mod ArraySize)] ← Granted) ∘>
    (´aux_queue ← tl ´aux_queue) ∘>
    (´aux_head ← (´aux_head + 1) mod ArraySize) ∘>
    (´aux_mid_release ← None)"

lemma rel_2_same:
  "s' = rel_line2 i s ⟹
    myindex s = myindex s' ∧
       tail s = tail s' ∧
    (∀ j ≠ (myindex s i + 1) mod ArraySize.
    flag_mapping s j = flag_mapping s' j)"
  by fastforce

lemma rel_2_invar:
  assumes assm_old: "s ∈ cblock_invar i"
      and assm_pre: "at_head i (aux_queue s) ∧ aux_mid_release s = Some i"
      and assm_new: "s' = rel_line2 i s"
    shows "s' ∈ cblock_invar i"
proof method_cblock_invar
  case flag show ?case
```

```
  apply (method_invar_flag)
      using assm_new assm_old assm_pre
      by (force simp: head_and_head_index invar_flag_def)+
next
  case bound
  have "tail s' < ArraySize"
    using  assm_new assm_old
    by (fastforce simp: invar_bounds_def)
  moreover have "aux_head s' < ArraySize"
    using assms
    by fastforce
  moreover have "i < NumThreads"
    using assm_old assm_new by fastforce
  ultimately show ?case
    using invar_bounds_def by blast
next
  case queue show ?case
  proof method_invar_queue
    case bound_thread_id
    show ?case
      using assm_new assm_old assm_pre
      by (fastforce simp: invar_queue_def list.set_sel(2))
  next
    case map_used_indices
    have same: "tail s = tail s' ∧
             myindex s = myindex s'"
      using assm_new by (fastforce intro: rel_2_same)
    have "aux_queue s ≠ []"
      using assm_pre by fastforce
    hence d: "aux_head s ≠ tail s"
      using assm_old head_and_head_index tail_never_used by force
    have t: "aux_queue s' = tl (aux_queue s)"
      using assm_new by fastforce
    have m: "map (myindex s) (aux_queue s) = used_indices s"
      using assm_old invar_queue_def by fastforce
    have "used_indices s' = tl (used_indices s)"
    proof-
      { assume a: "aux_head s ≤ tail s"
        hence 1: "aux_head s + 1 < ArraySize"
          using d assm_old invar_bounds_def by force
        hence 2: "aux_head s' = aux_head s + 1"
          using assm_new mod_less by force
        hence 3: "aux_head s' ≤ tail s'"
          using a d 2 same by fastforce

        have "used_indices s = [aux_head s ..< tail s]"
          using a used_indices_def by simp
        also have "... = aux_head s # [aux_head s + 1 ..< tail s]"
          using a d upt_eq_Cons_conv by fastforce
        also have "... = aux_head s # [aux_head s' ..< tail s]"
          using 2 by fastforce
        also have "... = aux_head s # [aux_head s' ..< tail s']"
          using assm_new rel_2_same by fastforce
        also have "... = aux_head s # used_indices s'"
          using 3 used_indices_def by fastforce

        ultimately have ?thesis
          by simp }
```

```
      moreover
      { assume a: "aux_head s > tail s ∧ aux_head s = ArraySize - 1"
        have "aux_head s' = (aux_head s + 1) mod ArraySize"
          using assm_new by simp
        also have "... = 0"
          using a Suc_eq_plus1 diff_Suc_1 by presburger
        also have "... ≤ tail s'"
          by simp
        ultimately have b: "used_indices s' = [0 ..< tail s']"
          using used_indices_def by presburger

        from a have "used_indices s = aux_head s # [0 ..< tail s]"
          using used_indices_def by fastforce
        also have "... = aux_head s # used_indices s'"
          using same b by simp
        ultimately have ?thesis by simp }
      moreover
      { assume a: "tail s < aux_head s ∧ aux_head s ≠ ArraySize - 1"
        hence b: "aux_head s < ArraySize - 1"
          using assm_old invar_bounds_def by force
        hence "aux_head s + 1 = (aux_head s + 1) mod ArraySize"
          by simp
        also have "... = aux_head s'"
          using assm_new by simp

        ultimately have c: "tail s' < aux_head s' ∧ aux_head s + 1 = aux_head s'"
          using a same by simp
        hence d: "used_indices s' = [aux_head s' ..< ArraySize] @ [0 ..< tail s']"
          using used_indices_def by simp

        from a have "used_indices s = [aux_head s ..< ArraySize] @ [0 ..< tail s]"
          using used_indices_def by simp
        also have "... = aux_head s # [aux_head s' ..< ArraySize] @ [0 ..< tail s]"
          using a b c upt_rec by force
        also have "... = aux_head s # used_indices s'"
          using same d by simp

        ultimately have ?thesis by simp }
      ultimately show ?thesis by force
    qed

    hence "map (myindex s) (aux_queue s') = used_indices s'"
      by (simp add: t m map_tl)
    thus ?case using same by (simp add: invar_queue_def)
  qed
qed

lemma rel_2_est_guar:
  assumes assm_old : "s ∈ cblock_invar i"
      and assm_pre : "at_head i (aux_queue s) ∧ aux_mid_release s = Some i"
      and assm_new : "s' = rel_line2 i s"
    shows "(s, s') ∈ for_others cblock_contract i
                  ∩ pred_to_rel (cblock_invar i)"
proof-
  { fix j assume u: "j ≠ i"
    hence "(s, s') ∈ contract_raw j"
    proof-
      have "myindex s j = myindex s' j"
```

```
      using assms rel_2_same by presburger
    moreover
    { assume "j ∈ set (aux_queue s) ∧ flag_mapping s (myindex s j) = Granted"
      hence "flag_mapping s' (myindex s j) = Granted"
        using assms by simp
      hence "flag_mapping s' (myindex s' j) = Granted"
        using assms rel_2_same by (metis (no_types, lifting)) }
    ultimately show ?thesis by simp
  qed
  moreover have "(s, s') ∈ contract_aux j"
  proof-
    have s: "tl (aux_queue s) = aux_queue s' ∧
             hd (aux_queue s) = i ∧
             i ≠ j"
      using assm_new assm_pre u by simp
    hence "j ∈ set (aux_queue s) ⟷ j ∈ set (aux_queue s')"
      by (metis RG_Tran.nth_tl hd_conv_nth list.sel(2) list.set_sel(2) set_ConsD)
    thus ?thesis using s by simp
  qed
  ultimately have "(s, s') ∈ cblock_contract j"
    by simp }
 moreover have "(s, s') ∈ pred_to_rel (cblock_invar i)"
  using assms rel_2_invar by force
 ultimately show ?thesis by simp
qed

theorem cblock_rel2:
 "rely: cblock_contract i   guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }
  BasicAnno (rel_line2 i)
    { ⦃ i ∉ set ´aux_queue ⦄ }"
proof method_anno_ultimate
  case est_guar
  thus ?case
    using rel_2_est_guar by fastforce
next
  case est_post
  thus ?case
    using rel_2_invar apply clarsimp
    by (metis (mono_tags, lifting) distinct.simps(2) distinct_map distinct_used_indices

                                   invar_queue_def list.collapse mem_Collect_eq)
qed (fastforce+)
```

7.4 RG Theorems

```
theorem cblock_acq:
 "rely: cblock_contract i   guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ i ∉ set ´aux_queue ⦄ }
  BasicAnno (acq_line1 i) .;
    { ⦃ i ∈ set ´aux_queue ⦄ }
  NoAnno (WHILE ´flag_mapping (´myindex i) = Pending DO SKIP OD)
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }"
  apply method_anno_ultimate
  using cblock_acq1 cblock_acq2 by fastforce+
```

```
theorem cblock_rel:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }
  BasicAnno (rel_line1 i) .;
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }
  BasicAnno (rel_line2 i)
    { ⦃ i ∉ set ´aux_queue ⦄ }"
  apply method_anno_ultimate
  using cblock_rel1 cblock_rel2 annquin_simp by blast+

theorem cblock_local:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ i ∉ set ´aux_queue ⦄ }
  BasicAnno (acq_line1 i) .;
    { ⦃ i ∈ set ´aux_queue ⦄ }
  NoAnno (WHILE ´flag_mapping (´myindex i) = Pending DO SKIP OD) .;
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }
  BasicAnno (rel_line1 i) .;
    { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }
  BasicAnno (rel_line2 i)
    { ⦃ i ∉ set ´aux_queue ⦄ }"
  apply (method_anno_ultimate)
     using cblock_acq1 annquin_simp apply blast
    using cblock_acq2 annquin_simp apply force
   using cblock_rel1 annquin_simp apply blast
  using cblock_rel2 annquin_simp by blast
```

When Sledgehammer is applied directly to one of the subgoals of the next theorem `cblock_local_loop`, several solvers do find proofs but do not report back. However, when that subgoal is explicitly copied into a separate lemma below, sledgehammer does find an SMT proof.

```
lemma lma_tmp:
  assumes
  "rely: cblock_contract t ∩ pred_to_rel (cblock_invar t)
   guar: invar_and_guar (cblock_invar t) (for_others cblock_contract t)
   anno_code:
     {⦃t ∉ set ´aux_queue⦄ ∩ cblock_invar t}
   add_invar (cblock_invar t) (BasicAnno (acq_line1 t) .;
     {⦃t ∈ set ´aux_queue⦄}
   NoAnno (WHILE ´flag_mapping (´myindex t) = Pending DO SKIP  OD) .;
     {⦃at_head t ´aux_queue ∧ ´aux_mid_release = None⦄}
   BasicAnno (rel_line1 t) .;
     {⦃at_head t ´aux_queue ∧ ´aux_mid_release = Some t⦄}
   BasicAnno (rel_line2 t))
     {⦃t ∉ set ´aux_queue⦄ ∩ cblock_invar t}"
  shows
  "anncom_spec_valid
    ({⦃t ∉ set ´aux_queue⦄ ∩ cblock_invar t ∩ ⦃t ∉ set ´aux_queue⦄)
    (cblock_contract t ∩ pred_to_rel (cblock_invar t))
    (invar_and_guar (cblock_invar t) (for_others cblock_contract t))
    ({⦃t ∉ set ´aux_queue⦄ ∩ cblock_invar t)
    (add_invar (cblock_invar t)
     (BasicAnno (acq_line1 t) .;
      {⦃t ∈ set ´aux_queue⦄}
      NoAnno (WHILE ´flag_mapping (´myindex t) = Pending DO SKIP  OD) .;
      {⦃at_head t ´aux_queue ∧ ´aux_mid_release = None⦄}
      BasicAnno (rel_line1 t) .;
```

```
      {⦃at_head t ´aux_queue ∧ ´aux_mid_release = Some t⦄}
      BasicAnno (rel_line2 t)))"
  using assms annquin_simp
  by (smt (verit) Int_absorb inf_assoc inf_commute)

theorem cblock_local_loop:
 "rely: cblock_contract i    guar: for_others cblock_contract i
  inv:  cblock_invar i  anno_code:
    { ⦃ i ∉ set ´aux_queue ⦄ }
  WhileAnno UNIV
      ( ⦃ i ∉ set ´aux_queue ⦄ )
  ( BasicAnno (acq_line1 i) .;
      { ⦃ i ∈ set ´aux_queue ⦄ }
    NoAnno (WHILE ´flag_mapping (´myindex i) = Pending DO SKIP OD) .;
      { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }
    BasicAnno (rel_line1 i) .;
      { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }
    BasicAnno (rel_line2 i) )
  { {} }"
proof method_anno_ultimate
  case body
  thus ?case
    by (rule lma_tmp, rule cblock_local)
qed (fastforce+)
```

The overall theorem expressing the correctness of the Circular Buffer Lock.

```
theorem cblock_global:
  "annotated global_init: cblock_init global_rely: Id
    ∥ i < NumThreads @

  { ⦃ i ∉ set ´aux_queue ⦄, cblock_contract i }
  WhileAnno UNIV
      ( ⦃ i ∉ set ´aux_queue ⦄ )
  ( BasicAnno (acq_line1 i) .;
      { ⦃ i ∈ set ´aux_queue ⦄ }
    NoAnno (WHILE ´flag_mapping (´myindex i) = Pending DO SKIP OD) .;
      { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = None ⦄ }
    BasicAnno (rel_line1 i) .;
      { ⦃ at_head i ´aux_queue ∧ ´aux_mid_release = Some i ⦄ }
    BasicAnno (rel_line2 i) )

  // cblock_invar i { for_others cblock_contract i, {} }
  global_guar: UNIV global_post: {}"
  apply (method_anno_ultimate)
       apply (fastforce intro!: cblock_local_loop)
      using cblock_init_def cblock_init_invar apply force
     using cblock_contracts cblock_invariants apply fastforce
  by (fastforce simp: assm_locale)+
end
```

End of locale

end

End of theory

Acknowledgement

This work was funded by the Department of Defence, and administered through the Advanced Strategic Capabilities Accelerator.

References

[1] R. J. Colvin, S. Heiner, P. Höfner, and R. C. Su. Rely-guarantee concurrency verification of queued locks in Isabelle/HOL. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2025.

[2] L. Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *Programming Languages and Systems (ESOP)*, pages 348–362, 2003.