Rely-Guarantee Extensions and Locks

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

November 21, 2025

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 **2**

2 Rely-Guarantee (RG) Syntax Extensions **2**
- 2.1 Lifting of Invariants . . . 3
- 2.2 RG Sentences . . . 3
- 2.3 RG Subgoal-Generating Methods . . . 4
  - 2.3.1 Basic . . . 4
  - 2.3.2 Looping constructs . . . 5
  - 2.3.3 Conditionals . . . 7
- 2.4 Parallel Compositions . . . 8
  - 2.4.1 Binary Parallel . . . 8
  - 2.4.2 Multi-Parallel . . . 10
- 2.5 Syntax of Record-Updates . . . 11

3 Annotated Commands **11**
- 3.1 Annotated Quintuples . . . 14
- 3.2 Structured Tactics for Annotated Commands . . . 15
- 3.3 Binary Parallel . . . 16
- 3.4 Helpers: Index Offsets . . . 17
- 3.5 Multi-Parallel . . . 18
- 3.6 The Main Tactics . . . 20

4 Examples Reworked **21**
- 4.1 Setting Elements of an Array to Zero . . . 21
- 4.2 Incrementing a Variable in Parallel . . . 22
- 4.3 FindP . . . 23

5 Abstract Queue Lock **25**

6 Ticket Lock **26**
6.1 Helpers: Inj, Surj and Bij . . . 26
6.1.1 Inj-Related . . . 27
6.1.2 Surj-Related . . . 28
6.1.3 Bij and Inv . . . 29
6.2 Helpers: Multi-Updates on Functions . . . 30
6.2.1 Ordering of Updates . . . 31
6.2.2 Surjective . . . 32
6.2.3 Injective . . . 32
6.2.4 Set- and List-Intervals . . . 33
6.3 Basic Definitions . . . 33
6.4 RG Theorems . . . 35

7 Circular-Buffer Queue-Lock **37**
7.1 Invariant . . . 39
7.1.1 Invariant Methods . . . 40
7.1.2 Invariant Lemmas . . . 41
7.2 Contract . . . 43
7.3 RG Lemmas . . . 43
7.4 RG Theorems . . . 45

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]"
  ⟨proof⟩

theorem weaken_post:
  assumes "Q ⊆ Q'"
      and "⊢ c sat [P, R, G, Q ]"
    shows "⊢ c sat [P, R, G, Q']"
  ⟨proof⟩
```

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}"
  ⟨proof⟩

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}"
  ⟨proof⟩

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]"
  ⟨proof⟩

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}"
  ⟨proof⟩

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}"
  ⟨proof⟩
```

```
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}"
  ⟨proof⟩


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⟩

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}"
  ⟨proof⟩

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}"
  ⟨proof⟩

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}"
⟨proof⟩

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}"
  ⟨proof⟩

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}"
  ⟨proof⟩

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}"
  ⟨proof⟩
```

```
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}"
  ⟨proof⟩

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}"
  ⟨proof⟩

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"
```
⟨*proof*⟩

lemma simp_gen_Un_2:
```
"(⋃ x ∈ {|´(<) (Suc (Suc 0))|}. S x) = S 0 ∪ S 1"
```
⟨*proof*⟩

lemma simp_gen_Un_2_not0:
```
"(⋃ x ∈ {|´(<) (Suc (Suc 0)) ∧ ´(≠) (Suc 0)|}. S x) = S 0"
```
⟨*proof*⟩

lemma simp_gen_Int_2:
```
"(⋂ x ∈ {|´(<) (Suc (Suc 0))|}. S x) = S 0 ∩ S 1"
```
⟨*proof*⟩

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]"
```
⟨*proof*⟩

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}"
```
⟨*proof*⟩

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}"
```
⟨*proof*⟩

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⟩

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']"
```
⟨*proof*⟩

```
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*⟩

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"
```
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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}"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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}"
⟨proof⟩

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)"
⟨proof⟩

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))"
⟨proof⟩

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}"
⟨proof⟩

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]"
```
⟨*proof*⟩

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}"
```
⟨*proof*⟩

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}"
```
⟨*proof*⟩

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)))"
⟨*proof*⟩

lemma Int_set_range_to_offset:
"(⋂i∈{lo..<hi::nat}. f i) = (⋂i<(hi-lo). f (lo + i))"
⟨*proof*⟩

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

lemma Int_set_range_neq_to_offset: "i = lo + ii
⟹ (⋂j∈{lo..<hi≠i}. f j) = (⋂j∈{0..<(hi-lo)≠ ii}. f (lo + j))"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma forall_range_to_offset:
"(∀i∈{lo..<(hi::nat)}. P i) ⟷ (∀i∈{0..<(hi - lo)}. P (lo + i))"
⟨*proof*⟩

lemma SCHEME_map_domain:
"map (λi. rgac i) [lo ..< (N::nat)] = map (λi. rgac (lo + i)) [0..<(N-lo)]"
⟨*proof*⟩

lemma offset_P: "(∀i. lo ≤ i ∧ i < (N::nat) ⟶ P i) ⟹ lo ≤ N ⟹ (∀i. i <(N-lo) ⟶ P (lo + i))"
⟨*proof*⟩

lemma INTER_offset:
shows "(⋂x<((N::nat) - lo). p (lo + x)) = (⋂x∈{lo..<N}. p x)"
⟨*proof*⟩

lemma LT_offset: "(∀i. lo ≤ i ∧ i < (N::nat) ⟶ P i) ⟷ (∀i<N - lo. P (lo + i))"
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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]"
```

⟨*proof*⟩

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]"
```

⟨*proof*⟩

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)"
  ⟨proof⟩

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)"
  ⟨proof⟩

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)"
  ⟨proof⟩

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)"
  ⟨proof⟩

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 ⟧}"
  ⟨proof⟩

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] ⟧}"
  ⟨proof⟩
```

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 |}}"
  ⟨proof⟩

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 |}}"
  ⟨proof⟩

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 |} }"
  ⟨proof⟩
```

Parameterised

```
lemma sum_split:
  "(j::nat) < (n::nat)
  ⟹ sum a {0..<n} = sum a {0..<j} + a j + sum a {j+1..<n}"
  ⟨proof⟩
```

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"
  ⟨proof⟩

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)"
  ⟨proof⟩

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 ⟶ °C j = ᵃC j ⦈},
     {⦇ ´C i = 1 ⦈ }"
```
⟨*proof*⟩

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 ⟶ °C j = ᵃC j ⦈},
     {⦇ ´C i = 1 ⦈ }
   global_guar: {⦇ True ⦈}
   global_post: {⦇ ´y = n ⦈}"
```
⟨*proof*⟩

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 ⟶ °C j = ᵃC j ⦈},
    {⦇ ´C i = 1 ⦈ }
  global_guar: {⦇ True ⦈}
  global_post: {⦇ ´y = n ⦈}"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

record Example3 =

```
  X :: "nat ⇒ nat"
  Y :: "nat ⇒ nat"

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

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

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

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

  global_guar: ⦃True⦄
  global_post: ⦃ ∀ i < n. (ˊX i) mod n=i
               ∧ (∀j<ˊX i. j mod n=i ⟶ ¬P(B!j))
               ∧ (ˊY i<m ⟶ P(B!(ˊY i)) ∧ˊY i≤ m+i)
               ∧ (∃j<n. ˊY j ≤ ˊX i) ⦄"
  ⟨proof⟩
```

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. ˊX i < ˊY j) DO
     IF P(B!(ˊX i)) THEN ˊY:=ˊY (i:=ˊX i) ELSE ˊX:= ˊX (i:=(ˊX i)+ n) FI
   OD,

 ⦃(ˊX i) mod n=i ∧ (∀j<ˊX i. j mod n=i ⟶ ¬P(B!j)) ∧ (ˊY i<m ⟶ P(B!(ˊY i)) ∧ ˊY 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⦄,

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

 COEND
 SAT [
   ⦃ ∀ i < n. ˊX i = i ∧ ˊY 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)|}]"
```
⟨*proof*⟩

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*⟩

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: {}"
  ⟨proof⟩
```

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"`
⟨*proof*⟩

lemma `inj_implies_inj_img: "inj f ⟹ inj_img f B"`
⟨*proof*⟩

lemma `inj_img_empty: "inj_img f {}"`
⟨*proof*⟩

lemma `inj_img_singleton: "∀ x. f x ≠ b ⟹ inj_img f {b}"`
⟨*proof*⟩

lemma `inj_img_subset:`
`"⟦ inj_img f B ; B' ⊆ B ⟧ ⟹ inj_img f B'"`
⟨*proof*⟩

lemma `inj_img_superset:`
`"⟦ inj_img f B ; ∀ x. f x ∉ B' - B ⟧ ⟹ inj_img f B'"`
⟨*proof*⟩

lemma `inj_img_not_mapped_to: "∀x. f x ∉ B ⟹ inj_img f B"`
⟨*proof*⟩

lemma `inj_img_add_one_extra:`

"⟦ inj_img f B ; ∀ x. f x ≠ b ⟧ ⟹ inj_img f (B ∪ {b})"
⟨*proof*⟩

lemma inj_img_union_1:
"⟦ inj_img f B1 ; inj_img f B2 ⟧ ⟹ inj_img f (B1 ∪ B2)"
⟨*proof*⟩

lemma inj_img_union_2:
"⟦ inj_img f B1 ; ∀ x. f x ∉ B2 ⟧ ⟹ inj_img f (B1 ∪ B2)"
⟨*proof*⟩

lemma inj_img_fun_upd_notin:
"⟦ inj_img f B ; ∀ x. f x ≠ b ⟧ ⟹ inj_img (fun_upd f a b) B"
⟨*proof*⟩

lemma inj_img_fun_upd_singleton:
"∀ x. f x ≠ b ⟹ inj_img (fun_upd f a b) {b}"
⟨*proof*⟩

6.1.2 Surj-Related

Lemmas related to "surj codom".

lemma surj_implies_surj_codom: "surj f ⟹ surj_codom f B"
⟨*proof*⟩

lemma surj_codom_triv: "surj_codom f (f ‘ A)"
⟨*proof*⟩

lemma surj_codom_univ: "surj_codom f UNIV = surj f"
⟨*proof*⟩

lemma surj_codom_empty: "surj_codom f {}"
⟨*proof*⟩

lemma surj_codom_singleton: "b ∈ range f ⟹ surj_codom f {b}"
⟨*proof*⟩

lemma surj_codom_subset:
"⟦ surj_codom f B ; B’ ⊆ B ⟧ ⟹ surj_codom f B’"
⟨*proof*⟩

lemma surj_codom_union:
"⟦ surj_codom f B1 ; surj_codom f B2 ⟧ ⟹ surj_codom f (B1 ∪ B2)"
⟨*proof*⟩

Lemmas related to "surj on".

lemma surj_on_implies_surj: "surj_on f A ⟹ surj f"
⟨*proof*⟩

lemma surj_on_univ: "surj_on f UNIV = surj f"
⟨*proof*⟩

lemma surj_on_never_emptyset: "¬ surj_on f {}"
⟨*proof*⟩

lemma surj_on_superset:
"⟦ surj_on f A ; A ⊆ A’ ⟧ ⟹ surj_on f A’"

⟨*proof*⟩

lemma surj_on_union:
"⟦ surj_on f A1 ; surj_on f A2 ⟧ ⟹ surj_on f (A1 ∪ A2)"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma bij_betw_implies_surj_codom: "bij_betw f A B ⟹ surj_codom f B"
⟨*proof*⟩

lemma bij_betw_implies_surj_on: "bij_betw f A UNIV ⟹ surj_on f A"
⟨*proof*⟩

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})"
⟨*proof*⟩

lemma bij_remove_one:
assumes "a ∈ A"
and "bij_betw f A B"
shows "bij_betw f (A - {a}) (B - {f a})"
⟨*proof*⟩

lemma set_remove_one_element:
assumes "x ∉ B"
and "B ⊆ A"
and "A - {x} ⊆ B"
shows "A - {x} = B"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
⟨*proof*⟩

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"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

```
lemma bij_betw_inv2:
  assumes "bij_betw f A B"
      and "b ∈ B"
    shows "f (inv f b) = b"
```
⟨*proof*⟩

```
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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

```
lemma fun_upd_list_l_is_foldl:
  "fun_upd_list_l f0 pairs = foldl (λ f pair. f(fst pair := snd pair)) f0 pairs"
```
⟨*proof*⟩

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*⟩

lemma `fun_upd_list_defs_distinct_equiv:`
```
  "distinct (map fst pairs) ⟹ fun_upd_list f pairs = fun_upd_list_l f pairs"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma `fun_upd_list_hd_1:`
```
  "fun_upd_list f (zip (x # xs) (y # ys)) x = y"
```
⟨*proof*⟩

lemma `fun_upd_list_hd_2:`
```
  "⟦ xs ≠ [] ; ys ≠ [] ⟧ ⟹ fun_upd_list f (zip xs ys) (hd xs) = hd ys"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma `fun_upd_list_not_updated_map:`
```
  assumes "a ∉ set (map fst xys)"
    shows "fun_upd_list f xys a = f a"
```
⟨*proof*⟩

lemma `fun_upd_list_not_updated_zip:`
```
  assumes "a ∉ set xs"
    shows "fun_upd_list f (zip xs ys) a = f a"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
⟨*proof*⟩

lemma fun_upd_list_surj_zip_1:
  assumes "length xs = length ys"
    shows "fun_upd_list f (zip xs ys) ‘ set xs ⊆ set ys"
  ⟨*proof*⟩

lemma fun_upd_list_surj_map_1:
  "(fun_upd_list f xys) ‘ set (map fst xys) ⊆ set (map snd xys)"
  ⟨*proof*⟩

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)"
⟨*proof*⟩

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"
  ⟨*proof*⟩

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"
⟨*proof*⟩

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*⟩

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"
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

lemma helper_interval_length:
  "length [1 ..< length xs + 1] = length xs"
  ⟨*proof*⟩

lemma helper_interval_union:
  "{0::nat} ∪ {1 ..< n + 1} = {0 ..< n + 1}"
  ⟨*proof*⟩

lemma fun_upd_list_interval:
  "fun_upd_list (λx.0) (zip xs [1 ..< length xs + 1]) z ∈ {0 ..< length xs + 1}"
  ⟨*proof*⟩

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*⟩

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)⦘"
  ⟨proof⟩

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)}"
  ⟨proof⟩

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} )"
  ⟨proof⟩

lemma tktlock_enqueue_inj:
  assumes "s ∈ tktlock_inv"
  shows "inj_img ((myticket s)(i := next_ticket s)) positive_nats"
  ⟨proof⟩

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*⟩

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*⟩

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*⟩

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 |} }"
```
⟨*proof*⟩

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*⟩

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*⟩

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)"
  ⟨proof⟩

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)"
  ⟨proof⟩
```

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨*proof*⟩

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*⟩

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"
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

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))"
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

lemma length_used_indices_queue:
  "s ∈ invar_queue ⟹ length (used_indices s) = length (aux_queue s)"
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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"
  ⟨*proof*⟩

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"
  ⟨*proof*⟩

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*⟩

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` ^{o}`aux_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⟩

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 |} }"
  ⟨proof⟩

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⟩


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')"
  ⟨proof⟩

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⟩

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⟩

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⟩
```

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)"
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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 |} }"
⟨*proof*⟩

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 |} }"

⟨*proof*⟩

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 ⦄ }"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

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*⟩

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: {}"
```
⟨*proof*⟩

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.