Abstract. VerifyThis 2018 [http://www.pm.inf.ethz.ch/research/verifythis.html] was a program verification competition associated with ETAPS 2018. It was the 7th event in the VerifyThis competition series. In this entry, we present polished and completed versions of our solutions that we created during the competition.
# Contents

1 Gap Buffer ........................................... 5
   1.1 Challenge ........................................... 5
   1.2 Solution ........................................... 7
      1.2.1 Abstract Specification ......................... 7
      1.2.2 Refinement 1: List with Gap .................... 9
      1.2.3 Implementation on List-Level .................... 9
      1.2.4 Imperative Arrays and Executable Code ........ 11
      1.2.5 Simple Client .................................. 12
   1.3 Shorter Solution ................................... 13
      1.3.1 Abstract Specification ......................... 13
      1.3.2 Refinement 1: List with Gap .................... 13
      1.3.3 Implementation on List-Level .................... 13
      1.3.4 Imperative Arrays ................................ 16
      1.3.5 Executable Code ................................ 17

2 Colored Tiles ........................................ 19
   2.1 Challenge ........................................... 19
   2.2 Solution ........................................... 20
      2.2.1 Problem Specification ......................... 20
      2.2.2 Derivation of Recursion Equations ............. 20
      2.2.3 Verification of Program ......................... 25
      2.2.4 Refinement to Imperative Code .................. 26
      2.2.5 Alternative Problem Specification ............. 27

3 Array-Based Queuing Lock .......................... 33
   3.1 Challenge ........................................... 33
   3.2 Solution ........................................... 35
      3.2.1 General Definitions ............................ 35
      3.2.2 Refinement 1: Ticket Lock with Unbounded Counters .. 36
      3.2.3 Refinement 2: Bounding the Counters ............. 46
      3.2.4 Refinement 3: Using an Array ................... 53
      3.2.5 Transfer Setup .................................. 57
      3.2.6 Main Theorems .................................. 58
1.1 Challenge

A gap buffer is a data structure for the implementation of text editors, which can efficiently move the cursor, as well add and delete characters. The idea is simple: the editor’s content is represented as a character array $a$ of length $n$, which has a gap of unused entries $a[l], \ldots, a[r-1]$, with respect to two indices $l \leq r$. The data it represents is composed as $a[0], \ldots, a[l-1], a[r], \ldots, a[n-1]$. The current cursor position is at the left index $l$, and if we type a character, it is written to $a[l]$ and $l$ is increased. When the gap becomes empty, the array is enlarged and the data from $r$ is shifted to the right.

**Implementation task.** Implement the following four operations in the language of your tool: Procedures `left()` and `right()` move the cursor by one character; `insert()` places a character at the beginning of the gap $a[l]$; `delete()` removes the character at $a[l]$ from the range of text.

```plaintext
procedure left()
    if l != 0 then
        l := l - 1
        r := r - 1
        a[r] := a[l]
    end-if
end-procedure

procedure right()
    // your task: similar to left()
    // but pay attention to the
    // order of statements
end-procedure
```

```plaintext
procedure insert(x: char)
    if l == r then
        // see extended task
        grow()
    end-if
    a[l] := x
    l := l + 1
end-procedure

procedure delete()
    if l != 0 then
        l := l - 1
    end-if
end-procedure
```

**Verification task.** Specify the intended behavior of the buffer in terms of a contiguous representation of the editor content. This can for example be based on strings, functional arrays, sequences, or lists. Verify that the gap buffer implementation satisfies this specification, and that every access to the array is within bounds.

5
CHAPTER 1. GAP BUFFER

*Hint:* For this task you may assume that `insert()` has the precondition \( l < r \) and remove the call to `grow()`. Alternatively, assume a contract for `grow()` that ensures that this call does not change the abstract representation.

**Extended verification task.** Implement the operation `grow()`, specify its behavior in a way that lets you verify `insert()` in a modular way (i.e. not by referring to the implementation of `grow()`), and verify that `grow()` satisfies this specification. *Hint:* You may assume that the allocation of the new buffer always succeeds. If your tool/language supports copying array ranges (such as `System.arraycopy()` in Java), consider using these primitives instead of the loops in the pseudo-code below.

```java
procedure grow()
    var b := new char[a.length + K]

    // b[0..l] := a[0..l]
    for i = 0 to l - 1 do
        b[i] := a[i]
    end-for

    // b[r + K..] := a[r..]
    for i = r to a.length - 1 do
        b[i + K] := a[i]
    end-for

    r := r + K
    a := b
end-procedure
```

**Resources**

1.2. Solution

theory Challenge1
imports lib/VTcomp
begin

Fully fledged specification of textbuffer ADT, and its implementation by a gap buffer.

1.2.1 Abstract Specification

Initially, we modelled the abstract text as a cursor position and a list. However, this gives you an invariant on the abstract level. An isomorphic but invariant free formulation is a pair of lists, representing the text before and after the cursor.

datatype 'a textbuffer = BUF 'a list 'a list

The primitive operations are the empty textbuffer, and to extract the text and the cursor position

definition empty :: 'a textbuffer where empty = BUF [] []
primrec get-text :: 'a textbuffer ⇒ 'a list where get-text (BUF a b) = a @ b
primrec get-pos :: 'a textbuffer ⇒ nat where get-pos (BUF a b) = length a

These are the operations that were specified in the challenge

primrec move-left :: 'a textbuffer ⇒ 'a textbuffer where
move-left (BUF a b) = (if a #[] then BUF (butlast a) (last a#b) else BUF a b)
primrec move-right :: 'a textbuffer ⇒ 'a textbuffer where
move-right (BUF a b) = (if b #[] then BUF (a@[hd b]) (tl b) else BUF a b)
primrec insert :: 'a⇒ 'a textbuffer ⇒ 'a textbuffer where
insert x (BUF a b) = BUF (a@[x]) b
primrec delete :: 'a textbuffer ⇒ 'a textbuffer where
delete (BUF a b) = BUF (butlast a) b
— Note that butlast [] = [] in Isabelle

We can also assign them a meaning wrt position and text

lemma empty-pos[simp]: get-pos empty = 0
  unfolding empty-def by auto
lemma empty-text[simp]: get-text empty = []
  unfolding empty-def by auto
lemma move-left-pos[simp]: get-pos (move-left b) = get-pos b - 1
  — Note that 0 - 1 = 0 in Isabelle
  by (cases b) auto
lemma move-left-text[simp]: get-text (move-left b) = get-text b
  by (cases b) auto
lemma move-right-pos[simp]:
CHAPTER 1. GAP BUFFER

get-pos (move-right b) = min (get-pos b+1) (length (get-text b))
by (cases b) auto

lemma move-right-text[simp]: get-text (move-right b) = get-text b
by (cases b) auto

lemma insert-pos[simp]: get-pos (insert x b) = get-pos b + 1
by (cases b) auto

lemma insert-text: get-text (insert x b)
= take (get-pos b) (get-text b) @ x @ drop (get-pos b) (get-text b)
by (cases b) auto

lemma delete-pos[simp]: get-pos (delete b) = get-pos b - 1
by (cases b) auto

lemma delete-text: get-text (delete b)
= take (get-pos b-1) (get-text b) @ drop (get-pos b) (get-text b)
by (cases b) auto

For the zero case, we can prove a simpler (equivalent) lemma

lemma delete-text0[simp]: get-pos b = 0 => get-text (delete b) = get-text b
by (cases b) auto

To fully exploit the capabilities of our tool, we can (optionally) show that the operations of a text buffer are parametric in its content. Then, we can automatically refine the representation of the content.

definition [no-relAPP]:
textbuffer-rel A ≡
{ (BUF a b, BUF a' b') | a b a' b' .
  (a,a') ∈ ⟨A⟩ list-rel ∧ (b,b') ∈ ⟨A⟩ list-rel }

lemma [param]: (BUF,BUF) ∈ ⟨A⟩ list-rel → ⟨A⟩ list-rel → ⟨A⟩ textbuffer-rel
by (auto simp: textbuffer-rel-def)

lemma [param]: (rec-textbuffer,rec-textbuffer)
∈ (⟨A⟩ list-rel → ⟨A⟩ list-rel→B) → ⟨A⟩ textbuffer-rel → B
by (auto simp: textbuffer-rel-def) parametricity

context
notes[simp] =
  empty-def get-text-def get-pos-def move-left-def move-right-def
  insert-def delete-def conv-to-is-Nil

begin

sepref-decl-op (no-def) empty :: ⟨A⟩ textbuffer-rel .
sepref-decl-op (no-def) get-text :: ⟨A⟩ textbuffer-rel → ⟨A⟩ list-rel .
sepref-decl-op (no-def) get-pos :: ⟨A⟩ textbuffer-rel → nat-rel .
sepref-decl-op (no-def) move-left :: ⟨A⟩ textbuffer-rel → ⟨A⟩ textbuffer-rel .
sepref-decl-op (no-def) move-right :: ⟨A⟩ textbuffer-rel → ⟨A⟩ textbuffer-rel .
sepref-decl-op (no-def) insert :: A → ⟨A⟩ textbuffer-rel → ⟨A⟩ textbuffer-rel .
sepref-decl-op (no-def) delete :: ⟨A⟩ textbuffer-rel → ⟨A⟩ textbuffer-rel .
end
1.2. SOLUTION

1.2.2 Refinement 1: List with Gap

1.2.3 Implementation on List-Level

type-synonym 'a gap-buffer = nat × nat × 'a list

Abstraction Relation

Also called coupling relation sometimes. Can be any relation, here we define it by
an invariant and an abstraction function.

definition gap-α ≡ λ(l,r,buf). BUF (take l buf) (drop r buf)
definition gap-invar ≡ λ(l,r,buf). l ≤ r ∧ r ≤ length buf
abbreviation gap-rel ≡ br gap-α gap-invar

Empty

definition empty1 ≡ RETURN (0,0,[])
lemma empty1-correct: (empty1, RETURN empty) ∈ ⟨gap-rel⟩nres-rel
unfolding empty1-def empty-def
apply refine-vcg
by (auto simp: in-br-conv gap-α-def gap-invar-def)

Left

definition move-left1 ≡ λ(l,r,buf). doN {
  if l≠0 then doN {
    ASSERT (r – 1 < length buf ∧ 1 – 1 < length buf);
    RETURN (l – 1, r – 1, buf ! (r – 1 : = buf ! (l – 1)));
  } else RETURN (l, r, buf)
}

lemma move-left1-correct:
(move-left1, RETURN o move-left) ∈ gap-rel → ⟨gap-rel⟩nres-rel
apply clarsimp
unfolding move-left1-def
apply refine-vcg
apply (auto
  simp: in-br-conv gap-α-def gap-invar-def move-left1-def
  split: prod.splits)
subgoal by (simp add: butlast-take)
subgoal
  by (smt Cons-nth-drop-Suc One-nat-def Suc-pred diff-Suc-less
drop-update-cancel last-take-nth-conv le-trans length-list-update
  less-le-trans neq0-conv nth-list-update-eq)
done

Right

definition move-right1 ≡ λ(l,r,buf). doN {

if \( r < \text{length buf} \) then \$
\begin{align*}
& \text{doN } \\
& \text{ASSERT } (l < \text{length buf}); \\
& \text{RETURN } ((l+1, r+1, \text{buf}[l:=\text{buf}[r]])
\end{align*}
$
\}
else \text{RETURN } (l, r, \text{buf})
\}\]

**Lemma move-right1-correct:**

\( \text{(move-right1.RETURN o move-right) } \in \text{gap-rel } \rightarrow (\text{gap-rel}) \text{nres-rel} \)

\text{apply clarsimp}

\text{unfolding move-right1-def}

\text{apply refine-vcg}

\text{unfolding gap-\( \alpha \)-def gap-invar-def}

\text{apply (auto}

\text{ simp in-br-conv hd-drop-conv-nth take-update-last}

\text{split prod.split)}

\text{by (simp add: drop-Suc tl-drop)}

**Insert and Grow**

\text{definition can-insert } \equiv \lambda (l, r, \text{buf}) . l < r

\text{definition grow1 \( K \) } \equiv \lambda (l, r, \text{buf}) . \text{doN } \$

\begin{align*}
& \text{let } b = \text{op-array-replicate (length buf } + \ K) \text{ default}; \\
& \text{RETURN } (l, r+K, b)
\end{align*}
$
\}

**Lemma grow1-correct[THEN SPEC-trans, refine-vcg]:**

\text{assumes gap-invar gb}

\text{shows grow1 \( \text{gb} \) } \leq (\text{SPEC } (\lambda \text{gb}'.

\text{gap-invar gb' } \\
\land \text{gap-\( \alpha \) gb' } = \text{gap-\( \alpha \) gb} \\
\land (K > 0 \rightarrow \text{can-insert gb'}) )
$

\text{unfolding grow1-def}

\text{apply refine-vcg}

\text{using assms}

\text{unfolding gap-\( \alpha \)-def gap-invar-def can-insert-def}

\text{apply clarsimp-all}

\text{apply (auto simp: op-list-blit-def)}

\text{by (simp add: min-def)}

**Definition insert1 \( x \) } \equiv \lambda (l, r, \text{buf}) . \text{doN } \$

\begin{align*}
& \text{if } (l=r) \text{ then grow1 (length buf } + 1) \ (l, r, \text{buf}) \text{ else RETURN } (l, r, \text{buf}); \\
& \text{RETURN } (l+1, r, \text{buf}[l:=x])
\end{align*}
$
\}$
1.2. SOLUTION

lemma insert1-correct:
\[(\text{insert1.\textsc{RETURN} oo insert}) \in \text{Id} \rightarrow \text{gap-rel} \rightarrow \langle \text{gap-rel} \rangle \text{nres-rel}\]
apply clarsimp
unfolding insert1-def
apply refine-vcg
unfolding insert-def gap-\alpha-def gap-invar-def can-insert-def
apply (auto simp: in-br-cone take-update-last split: prod.split)
done

Delete

definition delete1
\(\equiv \lambda (l,r,\text{buf}).\ \text{if } l>0\ \text{then} \ \text{RETURN} \ (l-1,r,\text{buf}) \ \text{else} \ \text{RETURN} \ (l,r,\text{buf})\)

lemma delete1-correct:
\[(\text{delete1.\textsc{RETURN} o delete}) \in \text{gap-rel} \rightarrow \langle \text{gap-rel} \rangle \text{nres-rel}\]
apply clarsimp
unfolding delete1-def
apply refine-vcg
unfolding gap-\alpha-def gap-invar-def
by (auto simp: in-br-conv butlast-take split: prod.split)

1.2.4 Imperative Arrays and Executable Code

abbreviation gap-impl-assn \(\equiv \text{nat-assn} \times_a \text{nat-assn} \times_a \text{array-assn} \text{id-assn}\)
definition gap-assn A
\(\equiv \text{hr-comp} (\text{hr-comp} \ \text{gap-impl-assn} \ \text{gap-rel}) \ ((\text{the-pure} A)\text{textbuffer-rel})\)

context
notes gap-assn-def[symmetric fcomp-norm-unfold]
begin
sepref-definition empty-impl
is uncurry0 empty1 :: unit-assn \(\Rightarrow_a \text{gap-impl-assn}\)
unfolding empty1-def array.fold-custom-empty
by sepref

sepref-definition move-left-impl
is move-left1 :: gap-impl-assn \(\Rightarrow_a \text{gap-impl-assn}\)
unfolding move-left1-def by sepref

sepref-definition move-right-impl
is move-right1 :: gap-impl-assn \(\Rightarrow_a \text{gap-impl-assn}\)
unfolding move-right1-def by sepref

sepref-definition insert-impl
is uncurry insert1 :: id-assn \(\times_a \text{gap-impl-assn}\)
unfolding move-right1-def by sepref

unfolding insert1-def grow1-def by sepref
— We inline grow1 here


sepref-definition delete-impl
is delete1 :: gap-assn \rightarrow_a gap-assn
unfolding delete1-def by sepref


end

The above setup generated the following refinement theorems, connecting the implementations with our abstract specification:

\[(\text{uncurry0 Challenge1.empty-impl, uncurry0 (RETURN Challenge1.empty)}) \in \text{unit-assn}^k \rightarrow_a \text{gap-assn} \ ? A\]
\[(\text{move-left-impl, RETURN } \circ \text{move-left}) \in (\text{gap-assn} \ ? A)^d \rightarrow_a \text{gap-assn} \ ? A\]
\[(\text{move-right-impl, RETURN } \circ \text{move-right}) \in (\text{gap-assn} \ ? A)^d \rightarrow_a \text{gap-assn} \ ? A\]
\[(\text{CONSTRAINT } \text{is-pure} \ ? A \implies \text{uncurry Challenge1.insert-impl, uncurry (RETURN } \circ \text{Challenge1.insert)}) \in \ ? A^k \rightarrow_a (\text{gap-assn} \ ? A)^d \rightarrow_a \text{gap-assn} \ ? A\]
\[(\text{delete-impl, RETURN } \circ \text{delete}) \in (\text{gap-assn} \ ? A)^d \rightarrow_a \text{gap-assn} \ ? A\]

export-code move-left-impl move-right-impl insert-impl delete-impl
in SML-imp module-name Gap-Buffer
in OCCaml imp module-name Gap-Buffer
in Haskell module-name Gap-Buffer
in Scala module-name Gap-Buffer

1.2.5 Simple Client

definition client ≡ RETURN (fold (\lambda f. f) [ ]
  insert (1::int),
  insert (2::int),
  insert (3::int),
  insert (5::int),
  move-left,
  insert (4::int),
  move-right,
  insert (6::int),
  delete
] empty)

lemma client ≤ SPEC (\lambda r. get-text r=[1,2,3,4,5])
unfolding client-def
by (simp add: delete-text insert-text)

sepref-definition client-impl
is uncurry0 client :: unit-assn^k \rightarrow_a gap-assn id-assn
unfolding client-impl fold.simps id-def comp-def
1.3. SHORTER SOLUTION

by sepref

ML-val
@ {code client-impl} ()

end

1.3 Shorter Solution

theory Challenge1-short
imports lib/VTcomp
begin

Small specification of textbuffer ADT, and its implementation by a gap buffer.
Annotated and elaborated version of just the challenge requirements.

1.3.1 Abstract Specification

datatype 'a textbuffer = BUF (pos: nat) (text: 'a list)
— Note that we do not model the abstract invariant — pos in range — here, as it is not
strictly required for the challenge spec.

These are the operations that were specified in the challenge. Note: Isabelle has
type inference, so we do not need to specify types. Note: We exploit that, in Isabelle,
we have $0 - 1 = 0$.

primrec move-left where move-left (BUF p t) = BUF (p - 1) t
primrec move-right where move-right (BUF p t) = BUF (min (length t) (p + 1)) t
primrec insert where insert x (BUF p t) = BUF (p + 1) (take p t @ x # drop p t)
primrec delete where delete (BUF p t) = BUF (p - 1) (take (p - 1) t @ drop p t)

1.3.2 Refinement 1: List with Gap

1.3.3 Implementation on List-Level

type-synonym 'a gap-buffer = nat × nat × 'a list

Abstraction Relation

We define an invariant on the concrete gap-buffer, and its mapping to the abstract
model. From these two, we define a relation gap-rel between concrete and abstract
buffers.

definition gap-α ≡ λ(l,r,buf). BUF l (take l buf @ drop r buf)
definition gap-invar ≡ λ(l,r,buf). l ≤ r ∧ r ≤ length buf
abbreviation gap-rel ≡ br gap-α gap-invar
Left

For the operations, we insert assertions. These are not required to prove the list-level specification correct (during the proof, they are inferred easily). However, they are required in the subsequent automatic refinement step to arrays, to give our tool the information that all indexes are, indeed, in bounds.

\[
\text{definition move-left1} \equiv \lambda (l, r, \text{buf}). \text{doN} \{ \\
\text{if } l \neq 0 \text{ then doN} \{ \\
\text{ASSERT}(r-1 < \text{length buf} \land l-1 < \text{length buf}); \\
\text{RETURN}(l-1, r-1, \text{buf}[r-1:=\text{buf}![l-1]]) \\
\} \text{ else RETURN }(l, r, \text{buf}) \\
\}\]

\[
\text{definition move-right1} \equiv \lambda (l, r, \text{buf}). \text{doN} \{ \\
\text{if } r < \text{length buf} \text{ then doN} \{ \\
\text{ASSERT}(l < \text{length buf}); \\
\text{RETURN}(l+1, r+1, \text{buf}[l:=\text{buf}![r]]) \\
\} \text{ else RETURN }(l, r, \text{buf}) \\
\}\]

Right

\[
\text{definition can-insert} \equiv \lambda (l, r, \text{buf}). l < r
\]
1.3. SHORTER SOLUTION

**definition** grow1 \( K \equiv \lambda (l,r,buf). \) doN {
  let b = op-array-replicate (length buf + K) default;
  b ← mop-list-blit buf 0 b 0 l;
  b ← mop-list-blit buf r b (r+K) (length buf - r);
  RETURN (l,r+K,b)
}

— Note: Most operations have also a variant prefixed with `mop`. These are defined in the refinement monad and already contain the assertion of their precondition. The backside is that they cannot be easily used in as part of expressions, e.g., in `buf[l := buf ! r]`, we would have to explicitly bind each intermediate value: `mop-list-get buf r >> mop-list-set buf l`.

**lemma** grow1-correct [THEN SPEC-trans, refine-vcg]:
— Declares this as a rule to be used by the VCG

assumes gap-invar gb

shows grow1 \( K gb \leq (SPEC (\lambda gb'. gap-invar gb').
gap-invar gb'
∧ gap-\(\alpha\) gb' = gap-\(\alpha\) gb
∧ (K>0 → can-insert gb'))

unfolding grow1-def

apply refine-vcg

using assms

unfolding gap-\(\alpha\)-def gap-invar-def can-insert-def

apply clarsimp-all

apply (auto simp: op-list-blit-def)

by (simp add: min-def)

**definition** insert1 \( x \equiv \lambda (l,r,buf). \) doN {
  (l,r,buf) ← if \( l = r \) then grow1 (length buf + 1) (l,r,buf) else RETURN (l,r,buf);
  ASSERT \( l<\text{length buf} \);
  RETURN \( l+1,r,buf[l := x] \)
}

**lemma** insert1-correct:

\( \text{insert1 \& RETURN oo insert} \in \text{Id} \to \text{gap-rel} \to \langle \text{gap-rel} \rangle \text{nres-rel} \)

apply clarsimp

unfolding insert1-def

apply refine-vcg — VCG knows the rule for grow1 already

unfolding insert-def gap-\(\alpha\)-def gap-invar-def can-insert-def

apply (auto simp: in-br-conv take-update-last split: prod split)

done

**Delete**

**definition** delete1

\( \equiv \lambda (l,r,buf). \) if \( l>0 \) then RETURN \( l-1,r,buf \) else RETURN \( l,r,buf \)

**lemma** delete1-correct:

\( \text{delete1 \& RETURN o delete} \in \text{gap-rel} \to \langle \text{gap-rel} \rangle \text{nres-rel} \)
apply clarsimp
unfolding delete1_def
apply refine-vcg
unfolding gap-α-def gap-invar-def
by (auto simp: in-br_conv butlast-take split: prod.split)

1.3.4 Imperative Arrays

The following indicates how we will further refine the gap-buffer: The list will become an array, the indices and the content will not be refined (expressed by nat-assn and id-assn).

abbreviation gap-impl-assn ≡ nat-assn ×ₐ nat-assn ×ₐ array-assn id-assn

sepref-definition move-left-impl
is move-left1 :: gap-impl-assn →ₐ gap-impl-assn
unfolding move-left1_def by sepref

sepref-definition move-right-impl
is move-right1 :: gap-impl-assn →ₐ gap-impl-assn
unfolding move-right1_def by sepref

sepref-definition insert-impl
is uncurry insert1 :: id-assn ×ₐ gap-impl-assn →ₐ gap-impl-assn
unfolding insert1_def grow1_def by sepref
— We inline grow1 here

sepref-definition delete-impl
is delete1 :: gap-impl-assn →ₐ gap-impl-assn
unfolding delete1_def by sepref

Finally, we combine the two refinement steps, to get overall correctness theorems

definition gap-assn ≡ hr-comp gap-impl-assn gap-rel
— hr-comp is composition of refinement relations
context notes gap-assn-def [symmetric fcomp-norm-unfold] begin
lemmas move-left-impl-correct = move-left-impl.refine[FCOMP move-left1-correct]
and move-right-impl-correct = move-right-impl.refine[FCOMP move-right1-correct]
and insert-impl-correct = insert-impl.refine[FCOMP insert1-correct]
and delete-impl-correct = delete-impl.refine[FCOMP delete1-correct]

Proves:

(move-left-impl, RETURN ∘ move-left) ∈ gap-assn →ₐ gap-assn

(move-right-impl, RETURN ∘ move-right) ∈ gap-assn →ₐ gap-assn

(uncurry Challenge1-short.insert-impl,
  uncurry (RETURN ∘ Challenge1-short.insert)) ∈ id-assn ×ₐ gap-assn →ₐ gap-assn
1.3. SHORTER SOLUTION

\((\text{delete-impl}, \text{RETURN} \circ \text{delete}) \in \text{gap-assn}^d \rightarrow_{\sigma} \text{gap-assn}\)

end

1.3.5 Executable Code

Isabelle/HOL can generate code in various target languages.

\begin{verbatim}
export-code move-left-impl move-right-impl insert-impl delete-impl
in SML-imp module-name Gap-Buffer
in OCaml-imp module-name Gap-Buffer
in Haskell module-name Gap-Buffer
in Scala module-name Gap-Buffer
\end{verbatim}

end
CHAPTER 1. GAP BUFFER
Colored Tiles

2.1 Challenge

This problem is based on Project Euler problem #114. Alice and Bob are decorating their kitchen, and they want to add a single row of fifty tiles on the edge of the kitchen counter. Tiles can be either red or black, and for aesthetic reasons, Alice and Bob insist that red tiles come by blocks of at least three consecutive tiles. Before starting, they wish to know how many ways there are of doing this. They come up with the following algorithm:

```java
var count[51] // count[i] is the number of valid rows of size i
count[0] := 1 // []
count[1] := 1 // [B] - cannot have a single red tile
count[2] := 1 // [BB] - cannot have one or two red tiles
for n = 4 to 50 do
    count[n] := count[n-1] // either the row starts with a black tile
    for k = 3 to n-1 do // or it starts with a block of k red tiles
        count[n] := count[n] + count[n-k-1] // followed by a black one
    end-for
    count[n] := count[n]+1 // or the entire row is red
end-for
```

**Verification tasks.** You should verify that at the end, `count[50]` will contain the right number.

**Hint:** Since the algorithm works by enumerating the valid colorings, we expect you to give a nice specification of a valid coloring and to prove the following properties:

1. Each coloring counted by the algorithm is valid.
2. No coloring is counted twice.
3. No valid coloring is missed.
2.2 Solution

theory Challenge2
imports lib/VTcomp
begin

The algorithm describes a dynamic programming scheme.
Instead of proving the 3 properties stated in the challenge separately, we approach
the problem by

1. Giving a natural specification of a valid tiling as a grammar
2. Deriving a recursion equation for the number of valid tilings
3. Verifying that the program returns the correct number (which obviously im-
    plies all three properties stated in the challenge)

2.2.1 Problem Specification
Colors
datatype color = R | B

Direct Natural Definition of a Valid Line
inductive valid where
valid [] |
valid xs ⇒ valid (B # xs) |
valid xs ⇒ n ≥ 3 ⇒ valid (replicate n R @ xs)
definition lcount n = card { l. length l=n ∧ valid l}

2.2.2 Derivation of Recursion Equations
This alternative variant helps us to prove the split lemma below.

inductive valid' where
valid' [] |
n ≥ 3 ⇒ valid' (replicate n R) |
valid' xs ⇒ valid' (B # xs) |
valid' xs ⇒ n ≥ 3 ⇒ valid' (replicate n R @ B # xs)

lemma valid-valid':
valid l ⇒ valid' l
by (induction rule: valid.induct)
(auto 4 4 intro; valid'.intros elim; valid'.cases
        simp: replicate-add[symmetric] append-assoc[symmetric]
    )
lemmas valid-red = valid.intros(3)[OF valid.intros(1), simplified]

lemma valid'-valid:
valid' l \implies valid l
by (induction rule: valid'.induct) (auto intro: valid.intros valid-red)

lemma valid-eq-valid':
valid' l = valid l
using valid-valid' valid'-valid by metis

Additional Facts on Replicate

lemma replicate-iff:
(\forall i<\text{length } l \; l! i = R) \iff (\exists \; n \; l = \text{replicate } n R)
by auto (metis (full-types) in-set-conv-nth replicate-eqI)

lemma replicate-iff2:
(\forall i < n \; l! i = R) \iff (\exists \; l'. \; l = \text{replicate } n R @ l') \text{ if } n < \text{length } l
using that by (auto simp: list-iff-nth-eq nth-append intro: exI[where x = drop n l])

lemma replicate-Conseq:
replicate n x = y @ ys \iff (\exists n'. n = \text{Suc } n' \land x = y \land \text{replicate } n' x = ys)
by (cases n) auto

Main Case Analysis on \texttt{@term valid}

lemma valid-split:
valid l \iff
l = [] \lor
(l!0 = B \land \text{valid } (tl l)) \lor
\text{length } l \geq 3 \land (\forall i < \text{length } l \; l! i = R) \lor
(\exists j < \text{length } l \; j \geq 3 \land (\forall i < j \; l! i = R) \land l! j = B \land \text{valid } (\text{drop } (j + 1) l))
unfolding valid-eq-valid'[symmetric]
apply standard
subgoal
by (erule valid'.cases) (auto simp: nth-append nth-Conseq split: nat.splits)
subgoal
by (auto intro: valid'.intros simp: replicate-iff elim!: disjE1)
(fastforce intro: valid'.intros simp: neq-Nil-conv replicate-iff2 nth-append)+
done

Base cases

lemma lc0-aux:
{l. l = [] \land \text{valid } l} = {[]}  
by (auto intro: valid.intros)

lemma lc0: lcount 0 = 1
by (auto simp: lc0-aux lcount-def)

lemma lc1aux: \{ l. length l=1 \land valid l \} = \{ [R] \}
by (auto intro: valid.intros elim: valid.cases simp: replicate-Cons-eq)

lemma lc2aux: \{ l. length l=2 \land valid l \} = \{ [B,B] \}
by (auto 4 3 intro: valid.intros elim: valid.cases simp: replicate-Cons-eq)

lemma lc3aux: \{ l. length l=3 \land valid l \} = \{ [B,B,B], [R,R,R] \}
by (auto 4 4 intro: valid.intros valid-red[of 3, simplified] elim: valid.cases simp: replicate-Cons-eq)

lemma lcounts-init: lcount 0 = 1 lcount 1 = 1 lcount 2 = 1 lcount 3 = 2
using lc0 lc1aux lc2aux lc3aux unfolding lcount-def by simp-all

The Recursion Case

lemma finite-valid-length: 
  finite \{ l. length l = n \land valid l \} (is finite ?S)
proof –
  have ?S \subseteq \lists \{ R, B \} \cap \{ l. length l = n \}
  by (auto intro: color.exhaust)
  moreover have finite ...
  by (auto intro: lists-of-len-fin1)
  ultimately show ?thesis
  by (rule finite-subset)
qed

lemma valid-line-just-B:
  valid (replicate n B)
by (induction n) (auto intro: valid.intros)

lemma valid-line-aux:
  \{ l. length l = n \land valid l \} \neq \{ \} (is ?S \neq \{ \})
using valid-line-just-B[of n] by force

lemma replicate-unequal-aux:
  replicate x R @ B # l \neq replicate y R @ B # l' (is ?l \neq ?r) if \langle x < y \rangle for l l'
proof –
  have ?l ! x = B ?r ! x = R
  using that by (auto simp: nth-append)
  then show ?thesis
  by auto
qed

lemma valid-prepend-B-iff:
  valid (B # xs) \iff valid xs
by (auto intro: valid.intros elim: valid.cases simp: Cons-replicate-eq Cons-eq-append-conv)
2.2. SOLUTION

\textbf{lemma} \( \text{lcrec} \ n = \text{lcoun} \ (n-1) + 1 + (\sum i=3..<n. \text{lcoun} \ (n-i-1)) \) \text{ if } n>3
\textbf{proof} –

\textbf{have} \{ l. \text{length} \ l = n \land \text{valid} \ l \}
= \{ l. \text{length} \ l = n \land \text{valid} \ (\text{tl} \ l) \land !0=B \}
\lor \{ l. \text{length} \ l = n \land
(\exists i. i < n \land i \geq 3 \land (\forall k < i. \ l!k = R) \land !i = B \land \text{valid} \ (\text{drop} \ (i+1) \ l)) \}
\lor \{ l. \text{length} \ l = n \land (\forall i<n. \ l!i=R) \}
\text{is } ?A = ?B \lor ?D \lor ?C
\textbf{using} \ n > 3 \text{ by (subst valid-split) auto}

\textbf{let} \ ?B1 = ((\#) B) \cdot \{ l. \text{length} \ l = n - \text{Suc} \ 0 \land \text{valid} \ l \}
\textbf{from} \ n > 3 \text{ have } ?B = ?B1
\textbf{apply safe}
\textbf{subgoal for} \ l
\textbf{by (cases \ l) (auto simp: valid-prepend-B-iff)}
\textbf{by} auto
\textbf{have} \ l: \text{card} ?B1 = \text{lcoun} \ (n-1)
\textbf{unfolding} \ \text{lcoun-def} \textbf{ by (auto intro: card-image)}

\textbf{have} \ ?C = \{ \text{replicate} \ n \ R \}
\textbf{by (auto simp: nth-equalityI) }
\textbf{have} \ 2: \text{card} \ \{ \text{replicate} \ n \ R \} = 1
\textbf{by} auto

\textbf{let} \ ?D1=(\bigcup i \in \{3..<n\}. (\lambda i. \text{replicate} \ i \ R @ B \# \ l)' \{ l. \text{length} \ l = n - i - 1 \land \text{valid} \ l \})
\textbf{have} \ ?D =
\bigcup i \in \{3..<n\}. \{ l. \text{length} \ l = n \land (\forall k < i. \ l!k = R) \land !i = B \land \text{valid} \ (\text{drop} \ (i+1) \ l) \}
\textbf{by} auto
\textbf{have} \ \{ l. \text{length} \ l = n \land (\forall k < i. \ l!k = R) \land !i = B \land \text{valid} \ (\text{drop} \ (i+1) \ l) \}
\textbf{if} \ i < n \ 2 < i \text{ for } i
\textbf{apply safe}
\textbf{subgoal for} \ l
\textbf{apply} \ (\text{rule image-eqI}[where x = \text{drop} \ (i+1) \ l])
\textbf{apply} \ (\text{rule nth-equalityI})
\textbf{using} that
\textbf{apply} \ (\text{simp-all split: nat.split add: nth-Cons nth-append})
\textbf{using} \ \text{add-diff-inverse-nat} \textbf{ apply fastforce}
\textbf{done}
\textbf{using} that \textbf{by} \ (\text{simp add: nth-append; fail})+

\textbf{then have} \ \text{D-eq:} \ ?D = ?D1
\textbf{unfolding} \ ?D = \cdot \textbf{ by} auto

\textbf{have} \ \text{inj: inj-on} \ (\lambda l. \text{replicate} \ x \ R @ B \# \ l)' \{ l. \text{length} \ l = n - \text{Suc} \ x \land \text{valid} \ l \} \textbf{ for } x
\textbf{unfolding} \ \text{inj-on-def} \textbf{ by} auto
have \( \ast \):

\[
(\lambda l. \text{replicate } x \; R \; @ \; B \; \# \; l) \setminus \{ l. \text{length } l = n - \text{Suc } x \land \text{valid } l \} \cap \\
(\lambda l. \text{replicate } y \; R \; @ \; B \; \# \; l) \setminus \{ l. \text{length } l = n - \text{Suc } y \land \text{valid } l \} = \{ \}
\]

if \( 3 \leq x < y < n \) for \( x, y \)

using that \( \text{replicate-unequal-aux} [\text{OF } x < y] \) by auto

have 3: \( \text{card } ?D1 = (\sum_{i=3..<n} \text{lcount } (n-i-1)) \)

proof (subt card-Union-disjoint, goal-cases)

\begin{enumerate}
\item case 3
\begin{enumerate}
\item show \(?case\)

proof (clarsimp, goal-cases)

\begin{enumerate}
\item case prems: (\( I \; x \; y \))

from prems show \(?case\)

\begin{enumerate}
\item apply –

\begin{enumerate}
\item apply (rule linorder-cases[of \( x \; y \)])
\item apply (rule \( \ast \); assumption)
\item apply (simp; fail)
\item apply (subt Int-commute; rule \( \ast \); assumption)
\end{enumerate}

done
\end{enumerate}

qed
\end{enumerate}

next
\item case 4
\begin{enumerate}
\item show \(?case\)

proof (subt sum.reindex, unfold inj-on-def , clarsimp, goal-cases)

\begin{enumerate}
\item case prems: (\( I \; x \; y \))

with \( \ast \) of \( x \; y \) valid-line-aux[of \( n - \text{Suc } x \)] show \(?case\)

\begin{enumerate}
\item by (rule linorder-cases[of \( x \; y \)], auto)
\end{enumerate}

next
\end{enumerate}

\item case 2
\begin{enumerate}
\item then show \(?case\)

\begin{enumerate}
\item by (simp add: lcount-def card-image[of \( \text{OF } \text{inj} \)])
\end{enumerate}

qed
\end{enumerate}

qed (auto intro: finite-subset[of \( \text{OF } \text{finite-valid-length} \) ])
\end{enumerate}

show \(?thesis\)

apply (subt lcount-def)

unfolding \( ?A = \ast \; ?B = \ast \; ?C = \ast \; \text{D-eq} \)

apply (subt card-Un-disjoint)

\begin{enumerate}
\item apply (blast intro: finite-subset[of \( \text{OF } \text{finite-valid-length} \) ] )
\end{enumerate}

subgoal

using Cons-replicate-eq[of \( B - n \; R \) ] replicate-unequal-aux by fastforce

apply (subt card-Un-disjoint)

\begin{enumerate}
\item apply (blast intro: finite-subset[of \( \text{OF } \text{finite-valid-length} \) ] )
\end{enumerate}

unfolding 1 2 3

by (auto simp: Cons-replicate-eq Cons-eq-append-conv)
2.2. SOLUTION

qed

2.2.3 Verification of Program

Inner Loop: Summation

definition sum-prog \( \Phi_{l u f} \) ≡
nfoldli \([l..<u]\) \((\lambda i. \text{True}) \((\lambda i. \text{s. doN}) \)
ASSERT (\Phi_{i});
RETURN (s+f i)
)

lemma sum-spec[THEN SPEC-trans, refine-vcg];
assumes \( l\leq u \)
assumes \( \forall i. l\leq i \Longrightarrow i<u \Longrightarrow \Phi_i \)
shows sum-prog \( \Phi_{l u f} \leq \text{SPEC}(\lambda r. r=(\sum i=l..<u.f i)) \)
unfolding sum-prog-def
supply nfoldli-upt-rule
apply refine-vcg
using asms
apply auto
done

Main Program

definition icount \( M \equiv \text{doN} \{
\}
ASSERT (M>2);
let c = op-array-replicate (M+1) 0;
let c = c[0:=1, 1:=1, 2:=1, 3:=2];

ASSERT (\forall i<4. c!i = lcount i);

\( c \leftarrow \text{nfoldli} [4..<M+1] \((\lambda i. \text{True}) \((\lambda n c. \text{doN}) \{\)
\{let sum = (\sum i=3..<n. c!(n-i-1));\}\)
sum ← sum-prog (\( \lambda i. n-i-l < \text{length} \ c \) \( \lambda n c. \text{doN} \) \{
\} c;

ASSERT (\forall i<M. c!i = lcount i);

ASSERT (M < \text{length} \ c);
RETURN (c!M)
\}

Abstract Correctness Statement

theorem icount-correct: \( M>2 \implies \text{icount} \ M \leq \text{SPEC}(\lambda r. r=\text{lcount} \ M) \)
unfolding icount-def
thm nfoldli-upt-rule
supply nfoldli-upt-rule\[where
I=\lambda n.\ n+1 \land (\forall i n. c!i = lcount i), \ refine-vcg] apply refine-vcg apply (auto simp:) subgoal for \ i
   apply (subgoal-tac \ i \in\ \{0,1,2,3\}) using \ lcounts-init by (auto)
subgoal for \ i c j
   apply (cases \ j < i)
   apply auto
   apply (subgoal-tac \ i = j)
   apply auto
   apply (subst \ \ lcrec [where \ n = j])
   apply auto
   done
done

2.2.4 Refinement to Imperative Code

sepref-definition \ icount-impl \ is \ icount :: \ nat-assn^k \to \alpha \ nat-assn unfolding \ icount-def \ sum-prog-def
by \ sepref

Main Correctness Statement

As the main theorem, we prove the following Hoare triple, stating: starting from the empty heap, our program will compute the correct result \( (lcount M) \).

\textbf{theorem} \ icount-impl-correct:
\[ M > 2 \implies <\text{emp}> \ icount-impl \ M <\lambda r. \ r = lcount M >_1 \]
\textbf{proof} –
\begin{itemize}
  \item note \ A = \ icount-impl,refine[to-hnr, THEN hn-refineD]
  \item note \ A = A[unfolded autoref-tag-defs]
  \item note \ A = A[unfolded \ hn-cxt-def \ pure-def, \ of \ M, \ simplified]
  \item note \ [sep-heap-rules] = A
\end{itemize}

assume \ M > 2

show \ ?thesis
  using \ icount-correct[OF \ M > 2]
  by (sep-auto simp: refine-pw-simps pw-le-iff)
\textbf{qed}

Code Export

\begin{itemize}
  \item export-code \ icount-impl \ in \ SML-imp \ module-name \ Tiling
  \item export-code \ icount-impl \ in \ OCaml-imp \ module-name \ Tiling
  \item export-code \ icount-impl \ in \ Haskell \ module-name \ Tiling
\end{itemize}
2.2. SOLUTION

export-code icount-impl in Scala-impl module-name Tiling

2.2.5 Alternative Problem Specification

Alternative definition of a valid line that we used in the competition

context fixes l :: color list begin

  inductive valid-point where
  | [ i+2<length l; l!i=R; l!(i+1) = R; l!(i+2) = R ] => valid-point i
  | [ 1<i+1<length l; l!(i) = R; l!(i+1) = R ] => valid-point i
  | [ 2<i+1<length l; l!(i-1) = R; l!(i) = R; l!(i+1) = R ] => valid-point i
  | [ i<length l; l!i=B ] => valid-point i

  definition valid-line = (\forall i<length l. valid-point i)
end

lemma valid-lineI:
  assumes \ A. i < length l \lra valid-point l i
  shows valid-line l
  using assms unfolding valid-line-def by auto

lemma valid-B-first:
  valid-point xs i => i < length xs => valid-point (B # xs) (i + 1)
  by (auto intro: valid-point.intros simp: numeral-2-eq-2 elim!: valid-point.cases)

lemma valid-line-prepend-B:
  valid-line (B # xs) if valid-line xs
  using that
  apply --
  apply (rule valid-lineI)
  subgoal for i
    by (cases i) (auto intro: valid-B-first[simplified] valid-point.intros simp: valid-line-def)
  done

lemma valid-drop-B:
  valid-point xs (i - 1) if valid-point (B # xs) i i > 0
  using that
  apply cases
    apply (fastforce intro: valid-point.intros)
  subgoal
    by (cases i = 1) (auto intro: valid-point.intros(2))
  subgoal
    unfolding numeral-nat by (cases i = 2) (auto intro: valid-point.intros(3))
  apply (fastforce intro: valid-point.intros)
  done

lemma valid-line-drop-B:
  valid-line xs if valid-line (B # xs)
\textbf{using} that unfolding \texttt{valid-line-def} \\
\textbf{proof} (safe, goal-cases) \\
\textbf{case} (1 \ i) \\
\textbf{with} \texttt{valid-drop-B[of \ xs \ i + 1]} \ \textbf{show} ?case \\
\textbf{by} auto \\
\textbf{qed} \\

\textbf{lemma} \texttt{valid-line-prepend-B-iff}: \\
\texttt{valid-line (B # xs) } \iff \texttt{ valid-line xs} \\
\textbf{using} \texttt{valid-line-prepend-B valid-line-drop-B by metis} \\

\textbf{lemma} \texttt{cases-valid-line}: \\
\textbf{assumes} \\
\texttt{l} = [ ] \lor \\
(\texttt{ | 0 = B } \land \texttt{ valid-line (tl \ l)} ) \lor \\
\texttt{\texttt{length \ i } \geq 3 } \land (\forall \ i < \texttt{length \ l}. \texttt{i ! i } = \texttt{R} ) \lor \\
(\exists \ j < \texttt{length \ l}. \texttt{j } \geq 3 \land (\forall \ i < j. \texttt{i ! i } = \texttt{R} ) \land \texttt{l ! j } = \texttt{B } \land \texttt{ valid-line (drop (j + 1) \ l)} ) \\
\texttt{is ?a } \lor \texttt{ ?b } \lor \texttt{ ?c } \lor \texttt{ ?d} \\
\texttt{shows} \texttt{ valid-line \ l} \\
\textbf{proof} – \\
\textbf{from} \texttt{assms} \ \textbf{consider} \ (empty)?a \ | \ (B) \sim \ ?a \land \ ?b \ | \ (all-red) \ ?c \ | \ (R-B) \ ?d \\
\textbf{by} blast \\
\textbf{then show} \ ?thesis \\
\textbf{proof} cases \\
\textbf{case} empty \\
\textbf{then show} \ ?thesis \\
\textbf{by} (simp add: \texttt{valid-line-def}) \\
\textbf{next} \\
\textbf{case} B \\
\textbf{then show} \ ?thesis \\
\textbf{by} (cases l) (auto simp: \texttt{valid-line-prepend-B-iff}) \\
\textbf{next} \\
\textbf{case} prems: all-red \\
\textbf{show} ?thesis \\
\textbf{proof} (rule valid-lineI) \\
\textbf{fix} i \ \textbf{assume} i < \texttt{length \ l} \\
\textbf{consider} i = 0 \ | \ i = 1 \ | \ i > 1 \\
\textbf{by} atomize-elim auto \\
\textbf{then show} valid-point \ l i \\
\textbf{using} \texttt{(i < \_)} \ \textbf{prems} \ \textbf{by} cases (auto 4 4 intro: valid-point.intros) \\
\textbf{qed} \\
\textbf{next} \\
\textbf{case} R-B \\
\textbf{then obtain} j \ \textbf{where} j: \\
\texttt{j < length \ l} \ 3 \leq \ j \ \texttt{(\forall i < j. \texttt{i ! i } = \texttt{R} )} \ 1 \texttt{j = B valid-line (drop (j + 1) \ l)} \\
\textbf{by} blast \\
\textbf{show} ?thesis \\
\textbf{proof} (rule valid-lineI) \\
\textbf{fix} i \ \textbf{assume} i < \texttt{length \ l}
with \( j \geq 3 \): consider \( i \leq j - 3 \mid i = j - 2 \mid i = j - 1 \mid i = j \mid i > j \) by atomize_elim auto then show valid-point \( l \ i \)

proof cases
  case 5
  with valid-line \( \cdot \mid i < \text{length } l \) have valid-point (drop (\( j + 1 \)) \( l \)) (\( i - j - 1 \)) unfolding valid-line-def by auto
  then show \(?\)thesis
    using \( i > j \) by cases (auto intro: valid-point.intros)

qed

qed

lemma valid-line-cases:
  \( l = \[] \lor \)
  (\( l \neq \[] = B \land \text{valid-line } (tl \ l) \lor \)
  length \( l \geq 3 \land (\forall i < \text{length } l. l! i = R) \lor \)
  (\( \exists j < \text{length } l. j \geq 3 \land (\forall i < j. l! i = R) \land l! j = B \land \text{valid-line } (\text{drop } (j + 1) \ l)) \)
if valid-line \( l \)

proof (cases \( l = \[] \))
  case True then show \(?\)thesis
    by (simp add: valid-line-def)

next
  case False
  show \(?\)thesis
  proof (cases \( l \neq B \))
    case True
    with \( \[] \neq \[] \) have \( l = B \# tl \ l \)
    by (cases \( l \) auto
        with valid-line \( l \) True show \(?\)thesis
        by (metis valid-line-prepend-B-iff)

next
  case False
  from valid-line \( l \) \( \[] \neq \[] \) have valid-point \( l \ l \)
  unfolding valid-line-def by auto
  with False have red-start: length \( l \geq 3 \l !0 = R \l!1 = R \l!2 = R \)
  by (auto elim!: valid-point.cases simp: numeral-2-eq-2)
  show \(?\)thesis
  proof (cases \( \forall i < \text{length } l. l! i = R \))
    case True
    with \( \text{length } l \geq 3 \) show \(?\)thesis
    by auto
  next
    case False
    let \( ?S = \{ j. j < \text{length } l \land j \geq 3 \land l! j = B \} \) let \( ?j = \text{Min } ?S \)
    have B-ge-3: \( i \geq 3 \) if \( l! i = B \) for \( i \)
    proof –
consider \( i = 0 \mid i = 1 \mid i = 2 \mid i \geq 3 \)
by atomize-elim auto
then show \( i \geq 3 \)
using red-start \( l! i = B \) by cases auto qed
from False obtain \( i \) where \( l! i = B \) \( i < \text{length } l \) \( i \geq 3 \)
by (auto intro: B-ge-3 color.exhaust)
then have \( ?j \in ?S \)
by -(rule Min-in, auto)
have \( \forall i < ?j. l! i = R \)
proof -
(\begin{align*}
\text{fix } i \text{ assume } i &< ?j \text{ } l! i = B \\
\text{then have } i &\geq 3 \\
\text{by (auto intro: B-ge-3)} \\
\text{with } \text{red-start } \( ?j \in ?S \) \text{ have } i \in ?S \\
\text{by auto} \\
\text{then have } ?j &\leq i \\
\text{by (auto intro: Min-le)} \\
\text{with } \text{have False} \\
\text{by simp}
\end{align*})
then show ?thesis
by (auto intro: color.exhaust) qed
with \( ?j \in ?S \) obtain \( j \) where \( j < \text{length } l \) \( j \geq 3 \) \( \forall i < j. l! i = R \) \( l! j = B \)
by blast
moreover have valid-line (drop \( (j + 1) \) \( l \))
proof (rule valid-lineI)
fix \( i \) assume \( i < \text{length (drop } (j + 1) \) \( l \))
with \( j \) have valid-point \( l (j + i + 1) \)
unfolding valid-line-def by auto
then show valid-point (drop \( (j + 1) \) \( l \)) \( i \)
proof cases
case 2
then show ?thesis
using \( j \) by (cases \( i \)) (auto intro: valid-point.intros)
next
case prems: 3
consider \( i = 0 \mid i = 1 \mid i > 1 \)
by atomize-elim auto
then show ?thesis
using \( j \) prems by cases (auto intro: valid-point.intros)
qed (auto intro: valid-point.intros)
qed
ultimately show ?thesis
by auto
qed
qed
2.2. SOLUTION

**qed**

**lemma valid-line-split:**
valid-line $l \iff$

- $l = [] \lor$
- $(l!0 = B \land \text{valid-line}(tl\ l)) \lor$
- $\text{length}\ l \geq 3 \land (\forall\ i < \text{length}\ l.\ l!i = R) \lor$
- $(\exists\ j < \text{length}\ l.\ j \geq 3 \land (\forall\ i < j.\ l!i = R) \land l!j = B \land \text{valid-line}(\text{drop}(j + 1)\ l))$

**using valid-line-cases cases-valid-line by blast**

Connection to the easier definition given above

**lemma valid-valid-line:**
valid $l \iff$ valid-line $l$

**by** (induction $l$ rule: length-induct, subst valid-line-split, subst valid-split, auto)

**end**
Array-Based Queuing Lock

3.1 Challenge

Array-Based Queuing Lock (ABQL) is a variation of the Ticket Lock algorithm with a bounded number of concurrent threads and improved scalability due to better cache behaviour.

We assume that there are $N$ threads and we allocate a shared Boolean array $\text{pass}[\ldots]$ of length $N$. We also allocate a shared integer value $\text{next}$. In practice, $\text{next}$ is an unsigned bounded integer that wraps to 0 on overflow, and we assume that the maximal value of $\text{next}$ is of the form $kN - 1$. Finally, we assume at our disposal an atomic $\text{fetch_and_add}$ instruction, such that $\text{fetch_and_add}(\text{next}, 1)$ increments the value of $\text{next}$ by 1 and returns the original value of $\text{next}$.

The elements of $\text{pass}[\ldots]$ are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of $\text{pass}[\ldots]$ is set to false, except $\text{pass}[0]$ which is set to true, allowing the first coming thread to acquire the lock. Variable $\text{next}$ contains the number of the first available place in the waiting queue and is initialized to 0.

Here is an implementation of the locking algorithm in pseudocode:

```pseudocode
procedure abql_init()
  for $i = 1 \text{ to } N - 1$
    $\text{pass}[i] := \text{false}$
  $\text{pass}[0] := \text{true}$
  $\text{next} := 0$
end-procedure

function abql_acquire()
  var $\text{my_ticket} := \text{fetch_and_add}(\text{next}, 1) \mod N$
  while not $\text{pass}[\text{my_ticket}]$
    end-while
  return $\text{my_ticket}$
end-function

procedure abql_release($\text{my_ticket}$)
  $\text{pass}[\text{my_ticket}] := \text{false}$
  $\text{pass}[(\text{my_ticket} + 1) \mod N] := \text{true}$
end-procedure
```

Each thread that acquires the lock must eventually release it by calling $\text{abql_release}(\text{my_ticket})$. 
where \texttt{my\_ticket} is the return value of the earlier call of \texttt{abql\_acquire()}. We assume that no thread tries to re-acquire the lock while already holding it, neither it attempts to release the lock which it does not possess. Notice that the first assignment in \texttt{abql\_release()} can be moved at the end of \texttt{abql\_acquire()}.

**Verification task 1.** Verify the safety of ABQL under the given assumptions. Specifically, you should prove that no two threads can hold the lock at any given time.

**Verification task 2.** Verify the fairness, namely that the threads acquire the lock in order of request.

**Verification task 3.** Verify the liveness under a fair scheduler, namely that each thread requesting the lock will eventually acquire it.

You have liberty of adapting the implementation and specification of the concurrent setting as best suited for your verification tool. In particular, solutions with a fixed value of \texttt{N} are acceptable. We expect, however, that the general idea of the algorithm and the non-deterministic behaviour of the scheduler shall be preserved.
3.2 Solution

theory Challenge3
imports lib/VTcomp lib/DF-System
begin

The Isabelle Refinement Framework does not support concurrency. However, Isabelle is a general purpose theorem prover, thus we can model the problem as a state machine, and prove properties over runs.

For this polished solution, we make use of a small library for transition systems and simulations: VerifyThis2018.DF-System. Note, however, that our definitions are still quite ad-hoc, and there are lots of opportunities to define libraries that make similar proofs simpler and more canonical.

We approach the final ABQL with three refinement steps:

1. We model a ticket lock with unbounded counters, and prove safety, fairness, and liveness.
2. We bound the counters by \( \text{mod } N \) and \( \text{mod } (k+N) \) respectively.
3. We implement the current counter by an array, yielding exactly the algorithm described in the challenge.

With a simulation argument, we transfer the properties of the abstract system over the refinements.

The final theorems proving safety, fairness, and liveness can be found at the end of this chapter, in Subsection 3.2.6.

3.2.1 General Definitions

We fix a positive number \( N \) of threads

consts \( N :: \text{nat} \)

specification \((N) \text{ N-not0 \{simp, intro!\}: N\neq0 \text{ by auto}}\)

lemma \((N-gt0 \{simp, intro!\}: 0<N \text{ by (cases } N) \text{ auto})\)

A thread’s state, representing the sequence points in the given algorithm. This will not change over the refinements.

datatype thread =
| INIT
| is-WAIT: WAIT (ticket: nat)
| is-HOLD: HOLD (ticket: nat)
| is-REL: REL (ticket: nat)
3.2.2 Refinement 1: Ticket Lock with Unbounded Counters

System’s state: Current ticket, next ticket, thread states

**Type-synonym**

\[ \text{astate} = \text{nat} \times \text{nat} \times (\text{nat} \rightarrow \text{thread}) \]

**Abbreviation**

\[ \text{cc} \equiv \text{fst} \]
\[ \text{nn s} \equiv \text{fst} (\text{snd s}) \]
\[ \text{tts s} \equiv \text{snd} (\text{snd s}) \]

The step relation of a single thread

**Inductive**

\[ \text{astep-sng} \]

| enter-wait: \[ \text{astep-sng} \text{ (c}, n, \text{INIT}) \text{ (c}, n+1, \text{WAIT n}) \]
| loop-wait: \[ c \neq k \Rightarrow \text{astep-sng} \text{ (c}, n, \text{WAIT k}) \text{ (c}, n, \text{WAIT k}) \]
| exit-wait: \[ \text{astep-sng} \text{ (c}, n, \text{WAIT c}) \text{ (c}, n, \text{HOLD c}) \]
| start-release: \[ \text{astep-sng} \text{ (c}, n, \text{HOLD k}) \text{ (c}, n, \text{REL k}) \]
| release: \[ \text{astep-sng} \text{ (c}, n, \text{REL k}) \text{ (k+1}, n, \text{INIT}) \]

The step relation of the system

**Inductive**

\[ \text{alstep} \text{ for t where} \]

\[ [ \text{t} < N; \text{astep-sng} \text{ (c}, n, \text{ts t}) \text{ (c'}, n', \text{s'}) ] \]

\[ \Rightarrow \text{alstep} \text{ (c}, n, \text{ts}) \text{ (c'}, n', \text{ts(t}=\text{s'})) \]

Initial state of the system

**Definition**

\[ \text{as}_0 \equiv (0, 0, \lambda -. \text{INIT}) \]

**Interpretation**

\[ A: \text{system as}_0 \text{ alstep} \]

In our system, each thread can always perform a step

**Lemma**

\[ \text{never-blocked: A.can-step l s} \longleftrightarrow l < N \]

**Apply**

\[ \text{cases s, cases tts s l, simp} \]

**Unfolding**

\[ A\text{-can-step-def} \]

**Apply**

\[ \text{clarsimp simp: alstep.simps astep-sng.simps; blast} \]

**Done**

Thus, our system is in particular deadlock free

**Interpretation**

\[ A: \text{df-system as}_0 \text{ alstep} \]

**Apply**

\[ \text{unfold-locales} \]

**Subgoal for s**

**Using**

\[ \text{never-blocked[of 0 s]} \]

**Unfolding**

\[ A\text{-can-step-def} \]

**By**

\[ \text{auto} \]

**Done**

**Safety: Mutual Exclusion**

Predicates to express that a thread uses or holds a ticket

**Definition**

\[ \text{has-ticket s k} \equiv s \text{=WAIT k} \lor s \text{=HOLD k} \lor s \text{=REL k} \]
3.2. SOLUTION

lemma has-ticket-simps[simp]:
¬has-ticket INIT k
has-ticket (WAIT k) \( k' \to k'=k \)
has-ticket (HOLD k) \( k' \to k'=k \)
has-ticket (REL k) \( k' \leftarrow k'=k \)
unfolding has-ticket-def by auto

definition locks-ticket s k ≡ s=HOLD k \lor s=REL k
lemma locks-ticket-simps[simp]:
¬locks-ticket INIT k
¬locks-ticket (WAIT k) k'
locks-ticket (HOLD k) k'\to k'=k
locks-ticket (REL k) k'\leftarrow k'=k
unfolding locks-ticket-def by auto

lemma holds-imp-uses: locks-ticket s k \implies has-ticket s k
unfolding locks-ticket-def by auto

We show the following invariant. Intuitively, it can be read as follows:

- Current lock is less than or equal next lock
- For all threads that use a ticket (i.e., are waiting, holding, or releasing):
  - The ticket is in between current and next
  - No other thread has the same ticket
  - Only the current ticket can be held (or released)

definition invar1 \equiv \lambda (c,n,ts).
\begin{align*}
c \leq n \\
& \land (\forall t. t < N \land \text{has-ticket (ts t) k} \implies \\
& c \leq k \land k < n \\
& \land (\forall t'. t' < N \land \text{has-ticket (ts t') k' \land t \neq t' \implies k \neq k')} \\
& \land (\forall k. k \neq c \implies \neg \text{locks-ticket (ts t) k})
\end{align*}

lemma is-invar1: A.is-invar invar1
apply rule
subgoal by (auto simp: invar1-def as0-def)
subgoal for s s'
apply (clarify)
apply (erule alstep.cases)
apply (erule alstep-sng.cases)
apply (clarsimp-all simp: invar1-def)
apply fastforce
apply fastforce
apply fastforce
apply fastforce
  by (metis Suc-le-eq holds-imp-uses locks-ticket-def le-neq-implies-less)
done

From the above invariant, it’s straightforward to show mutual exclusion

\[ \begin{align*}
  & A.\text{reachable} \ s; \\
  & t < N; t' < N; t \neq t'; \text{is-HOLD} \ (tts \ s \ t); \text{is-HOLD} \ (tts \ s \ t') \\
  \end{align*} \implies False \]

apply (cases tts \ s \ t; simp)
apply (cases tts \ s \ t'; simp)
using A.invar-reachable[OF is-invar1, of \ s]
apply (auto simp: invar1-def)
by (metis locks-ticket-simps(3) has-ticket-simps(3))

\[ \begin{align*}
  & A.\text{reachable'; }[A.\text{reachable} \ s; \\
  & t < N; t' < N; t \neq t'; \\
  & \text{locks-ticket} \ (tts \ s \ t) \ tk; \text{locks-ticket} \ (tts \ s \ t') \ tk' \\
  \end{align*} \implies False \]

apply (cases tts \ s \ t; simp; cases tts \ s \ t'; simp)
apply (cases tts \ s \ t'; simp)
using A.invar-reachable[OF is-invar1, of \ s]
apply (clarsimp-all simp: invar1-def)
unfolding locks-ticket-def has-ticket-def
apply metis+
done

Fairness: Ordered Lock Acquisition

We first show an auxiliary lemma: Consider a segment of a run from \( i \) to \( j \). Every thread that waits for a ticket in between the current ticket at \( i \) and the current ticket at \( j \) will be granted the lock in between \( i \) and \( j \).

\[ \begin{align*}
  & \text{lemma fair-aux: } \\
  & \text{assumes } R: A.\text{is-run} \ s \\
  & \text{assumes } A: i < j \ \text{cc} \ (s \ i) \leq k \ k < \text{cc} \ (s \ j) \ t < N \ tts \ (s \ i) \ t = \text{WAIT} \ k \\
  & \text{shows } \exists l. i \leq l \land l < j \land tts \ (s \ l) \ t = \text{HOLD} \ k \\
  & \text{proof} – \\
  & \text{interpret } A: \text{run } as_0 \ \text{alstep} \ s \ \text{by unfold-locales fact} \\
  & \text{from } A \text{ show } ?\text{thesis} \\
  & \text{proof (induction } j - i \text{ arbitrary: } i) \\
  & \text{case } 0 \\
  & \text{then show } ?\text{case by auto} \\
  & \text{next} \\
  & \text{case } (\text{Suc } i') \\
  & \text{hence } [\text{simp}]: i' = j - \text{Suc } i \ \text{by auto} \\
  & \text{note } IH = \text{Suc.kyps(1)[OF this]} \\
\]
obtain $t'$ where $\text{alstep } t'(s \ i) (s \ (\text{Suc } i))$ by (rule A.stepE)
then show $?case using Suc.prems
proof cases
  case $(1 \ c \ n \ ts \ c' \ n' \ s')$
  note [simp] = $I(1,2,3)$

  from $A.run-invar[OF \ is-invar1, \ of] \ have \ invar1 \ (c,n,ts)$ by auto
  note $IH = \ this[unfolded \ invar1-def, \ simplified]$

  from $I(4)$ show $?thesis$
  proof (cases rule: astep-sng.cases)
    case enter-wait
    then show $?thesis
      using $IH$ Suc.prems apply (auto)
      by (metis $I(2)$ Suc-leD Suc-lessI fst-conv leD distinct(1))
  next
    case (loop-wait $k$
    then show $?thesis
      using $IH$ Suc.prems apply (auto)
      by (metis $I(2)$ Suc-leD Suc-lessI fst-conv leD)
  next
    case exit-wait
    then show $?thesis
      apply (cases $t' = t$)
      subgoal
        using Suc.prems apply clarsimp
        by (metis $I(2)$ Suc-leD Suc-lessI fst-conv-upd-same leD
            less-or-eq-imp-le snd-conv)
      subgoal
        using Suc.prems IH
        apply auto
        by (metis $I(2)$ Suc-leD Suc-lessI fst-conv leD)
      done
  next
    case (start-release $k$
    then show $?thesis
      using $IH$ Suc.prems apply (auto)
      by (metis $I(2)$ Suc-leD Suc-lessI fst-conv leD thread distinct(7))
  next
    case (release $k$
    then show $?thesis
      apply (cases $t' = t$)
      using $IH$ Suc.prems apply (auto)
      by (metis $I(2)$ Suc-leD Suc-lessI fst-conv
          locks-ticket-simps(4) le-antisym not-less-eq-eq
          has-ticket-simps(2) has-ticket-simps(4))
    qed
  qed
lemma \texttt{s-case-expand}: 
\[(\text{case } s \text{ of } (c, n, ts) \Rightarrow P (c \text{ n} \text{ ts}) ) = P (cc s) (nn s) (tts s)\]  
by (auto split: prod.splits)

A version of the fairness lemma which is very detailed on the actual ticket numbers. We will weaken this later.

lemma \texttt{fair-aux2}:  
assumes \texttt{RUN}: A.\texttt{is-run s}  
assumes \texttt{ACQ}: \texttt{t<}\texttt{N} \texttt{tts (s i)} \texttt{t=INIT} \texttt{tts (s (Suc i))} \texttt{t=WAIT} \texttt{k}  
assumes \texttt{HOLD}: \texttt{i<j} \texttt{tts (s j)} \texttt{t = HOLD k}  
assumes \texttt{WAIT}: \texttt{t'}<\texttt{N} \texttt{tts (s i)} \texttt{t'} = \texttt{WAIT} \texttt{k}'  
obtains \texttt{l} where \texttt{i<l} \texttt{l<j} \texttt{tts (s l)} \texttt{t'} = \texttt{HOLD} \texttt{k}'  
proof –  
interpret A: run as \texttt{0} alstep s by unfold-locales fact from \texttt{ACQ \texttt{WAIT}}  
have [simp]: \texttt{t\neq t'} \texttt{t'}\neq t by auto  
from \texttt{ACQ} have [simp]: \texttt{nn (s i)} = \texttt{k} \land \texttt{nn (s (Suc i))} = \texttt{Suc k}  
\land \texttt{cc (s (Suc i))} = \texttt{cc (s i)} \land \texttt{tts (s (Suc i))} = \texttt{(tts (s i))(t:=WAIT k)}  
apply (rule-tac A.\texttt{stepE[of i]})  
apply (erule alstep.cases)  
apply (erule astep-sng.cases)  
by (auto simp: nth-list-update split: if-splits)

from A.\texttt{run-invar[OF is-invar1, of i]} have invar1 \texttt{(s i)} by auto  
note I1 = this [unfolded invar1-def, unfolded s-case-expand, simplified]

from \texttt{ACQ \texttt{HOLD}} have \texttt{Suc i \neq j} by auto with \texttt{HOLD have Suc i < j by auto}

have X1: \texttt{cc (s i)} \leq \texttt{k'} using I1 \texttt{WAIT by fastforce}  
have X2: \texttt{k'} < \texttt{cc (s j)}  
using A.\texttt{run-invar[OF is-invar1, of j, unfolded invar1-def s-case-expand]}  
using \texttt{k'} < \texttt{k} /	exttt{t<\texttt{N}} \texttt{HOLD(2)}  
apply clarsimp  
by (metis locks-ticket-simps(3) has-ticket-simps(3))

from \texttt{fair-aux[OF \texttt{RUN Suc i < j, of k'} t', simplified]} obtain \texttt{l} where \texttt{l>Suc i \texttt{l<j} \texttt{tts (s l)} \texttt{t'} = HOLD k'}  
using \texttt{WAIT X1 X2 by auto}

thus \texttt{?thesis}  
apply (rule-tac that[of \texttt{l}])  
by auto

qed
3.2. SOLUTION

lemma find-hold-position:
  assumes RUN: A.is-run s
  assumes WAIT: t<N tts (s i) t = WAIT tk
  assumes NWAIT: i<j tts (s j) t ≠ WAIT tk
  obtains l where i<l l≤j tts (s l) t = HOLD tk
proof —
  interpret A: run as0 alstep s by unfold-locales fact

  from WAIT(2) NWAIT have ∃l. i<l ∧ l≤j ∧ tts (s l) t = HOLD tk
proof (induction j−i arbitrary: i)
  case 0
  then show ?case by auto
next
  case (Suc i)

  hence [simp]: i'=j − Suc i by auto
  note IH = Suc.hyps(1)[OF this]

  obtain t' where alstep t'(s i) (s (Suc i)) by (rule A.stepE)
  then show ?case
apply —
apply (cases t=t',erule alstep.cases; erule astep-sng.cases)
apply auto
using IH Suc.prems Suc.hyps(2)
apply (auto)
apply (metis Suc-lessD Suc-lessI fun-upd-same snd-conv)
apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
apply (metis Suc.prems(1) Suc-lessD Suc-lessI fun-upd-triv)
apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
done
qed
thus ?thesis using that by blast
qed

Finally we can show fairness, which we state as follows: Whenever a thread t gets a ticket, all other threads t' waiting for the lock will be granted the lock before t.

theorem fair:
  assumes RUN: A.is-run s
  assumes ACQ: t<N tts (s i) t=INIT is-WAIT (tts (s (Suc i)) t)
  — Thread t calls acquire in step i
  assumes HOLD: i<j is-HOLD (tts (s j) t)
  — Thread t holds lock in step j
  assumes WAIT: t'<N is-WAIT (tts (s i) t')
  — Thread t' waits for lock at step i
  obtains l where i<l l<j is-HOLD (tts (s l) t')
  — Then, t' gets lock earlier
CHAPTER 3. ARRAY-BASED QUEUING LOCK

proof
obtain k where Wk: tts (s (Suc i)) t = WAIT k using ACQ
  by (cases tts (s (Suc i)) t) auto

obtain k' where Wk': tts (s i) t' = WAIT k' using WAIT
  by (cases tts (s i) t') auto

from ACQ HOLD have Suc i \neq j by auto with HOLD have Suc i < j by auto

obtain j' where H': Suc i < j' \leq j tts (s j') t = HOLD k
  apply (rule find-hold-position[OF RUN \langle t < N \rangle Wk Suc i < j \rangle])
  using HOLD(2) by auto

show ?thesis
  apply (rule fair-aux2[OF RUN ACQ(1,2) Wk - H'(3) WAIT(1) Wk'])
  subgoal using H'(1) by simp
  subgoal apply (erule that) using H'(2) by auto
  done
qed

Liveness

For all tickets in between the current and the next ticket, there is a thread that has this ticket

definition invar2
\equiv \lambda (c,n,ts). \forall k. c \leq k \land k < n \rightarrow (\exists t < N. has-ticket (ts t) k)

lemma is-invar2: A.is-invar invar2
  apply rule
  subgoal by (auto simp: invar2-def aso-def)
  subgoal for s s'
    apply (clarsimp simp: invar2-def)
    apply (erule alstep.cases; erule astep-sng.cases; clarsimp)
    apply (metis less-antisym has-ticket-simps(1))
    subgoal by (metis has-ticket-simps(2))
    subgoal by (metis has-ticket-simps(2))
    subgoal by (metis has-ticket-simps(3))
    subgoal
      apply (frule A.invar-reachable[OF is-invar1])
      unfolding invar1-def
      apply clarsimp
      by (metis Suc-leD locks-ticket-simps(4)
          not-less-eq-eq has-ticket-simps(4))
  done
  done

If a thread t is waiting for a lock, the current lock is also used by a thread

corollary current-lock-used:
3.2. SOLUTION

assumes $R : A.\text{reachable}(c,n,ts)$
assumes $\text{WAIT}: t < N \text{ ts t} = \text{WAIT} k$
obtains $t'$ where $t' < N \text{ has-ticket } (ts t') c$
using $A.\text{invar-reachable}[\text{OF is-invar2 } R]$
and $A.\text{invar-reachable}[\text{OF is-invar1 } R] \text{ WAIT}$
unfolding invar1-def invar2-def apply auto
by (metis (full-types) le-neq-implies-less not-le order-mono-setup refl)

Used tickets are unique (Corollary from invariant 1)

lemma has-ticket-unique: $[[A.\text{reachable } (c,n,ts);$
$t < N; \text{has-ticket } (ts t) k; t' < N; \text{has-ticket } (ts t') k]] \Rightarrow t' = t$
apply (drule $A.\text{invar-reachable}[\text{OF is-invar1}]$)
by (auto simp: invar1-def)

We define the thread that holds a specified ticket

definition tkt-thread $≡ \lambda ts k. \text{THE } t. t < N \land \text{has-ticket } (ts t) k$
lemma tkt-thread-eq:
assumes $R : A.\text{reachable}(c,n,ts)$
assumes $A : t < N \text{ has-ticket } (ts t) k$
shows $\text{tkt-thread } ts k = t$
using has-ticket-unique[\text{OF } R]
unfolding tkt-thread-def
using $A$ by auto

lemma holds-only-current:
assumes $R : A.\text{reachable}(c,n,ts)$
assumes $A : t < N \text{ locks-ticket } (ts t) k$
shows $k = c$
using $A.\text{invar-reachable}[\text{OF is-invar1 } R] \text{ A unfolding invar1-def}$
using holds-imp-uses by blast

For the inductive argument, we will use this measure, that decreases as a single
thread progresses through its phases.

definition tweight $s ≡$
case $s$ of $\text{WAIT} - \Rightarrow 3 :: \text{nat}$ | $\text{HOLD} - \Rightarrow 2$ | $\text{REL} - \Rightarrow 1$ | $\text{INIT} \Rightarrow 0$

We show progress: Every thread that waits for the lock will eventually hold the
lock.

theorem progress:
assumes $\text{FRUN}: A.\text{is-fair-run } s$
assumes $A : t < N \text{ is-WAIT } (\text{tts } (s i) t)$
shows $\exists j > i. \text{is-HOLD } (\text{tts } (s j) t)$
proof
interpret $A : \text{fair-run as} s_0 \text{ alstep } s$ by unfold-locales fact

from $A$ obtain $k$ where $Wk : \text{tts } (s i) t = \text{WAIT } k$
by \((\text{cases } \text{tts} \ (s \ i) \ t) \text{ auto}\)

We use the following induction scheme:

- Either the current thread increases (if we reach \(k\), we are done)
- (lex) the thread using the current ticket makes a step
- (lex) another thread makes a step

\[
\text{define } lrel \ \text{where} \ lrel \equiv \\
\text{inv-image (measure id } <\text{lex}*> \ \text{measure id } <\text{lex}*> \ \text{measure id}) \ (\lambda i. (k = \text{cc} \ (s \ i), \\
\text{tweight} \ (\text{tts} \ (s \ i)) \ (\text{tkt-thread} \ (\text{tts} \ (s \ i)) \ (\text{cc} \ (s \ i)), \\
\text{A.dist-step} \ (\text{tkt-thread} \ (\text{tts} \ (s \ i)) \ (\text{cc} \ (s \ i)) \ i)) \\

\text{have } \text{wf } lrel \ \text{unfolding } lrel-def \ \text{by } \text{auto} \\
\text{then show } \text{?thesis using } A(I) \ Wk \\
\text{proof (induction } i) \\
\text{case (less } i) \\
\[
\]

We name the components of this and the next state

\[
\text{obtain } c \ n \ ts \ \text{where } [\text{simp}]: s \ i = (c,n,ts) \ \text{by } (\text{cases } s \ i) \\
\text{from } A.\text{run-reachable}[of } i \ \text{have } R: A.\text{reachable } (c,n,ts) \ \text{by } \text{simp} \\
\text{obtain } c' \ n' \ ts' \ \text{where } [\text{simp}]: s \ (\text{Suc } i) = (c',n',ts') \\
\text{by } (\text{cases } s \ (\text{Suc } i)) \\
\text{from } A.\text{run-reachable}[of } \text{Suc } i \ \text{have } R': A.\text{reachable } (c',n',ts') \\
\text{by } \text{simp} \\
\text{from less.prems have } \text{WAIT}[\text{simp}]: ts \ t = \text{WAIT } k \ \text{by } \text{simp} \\
\[
\]

If thread \(t\) left waiting state, we are done

\[
\text{assume } ts' \ t \neq \text{WAIT } k \\
\text{hence } ts' \ t = \text{HOLD } k \ \text{using } \text{less.prems} \\
\text{apply } (\text{rule-tac } A.\text{stepE}[of } i) \\
\text{apply } (\text{auto elim!}: \text{alstep.cases astep-sng.cases split: if-splits}) \\
\text{done} \\
\text{hence } ?\text{case by auto} \\
\text{moreover } \{ \\
\text{assume } [\text{simp}]: ts' \ t = \text{WAIT } k \\
\}
\]

Otherwise, we obtain the thread \(tt\) that holds the current lock

\[
\text{obtain } tt \ \text{where } UTT: tt <N \text{ has-ticket } (ts \ tt) \ c \\
\text{using } \text{current-lock-used}[of c \ n \ ts \ t \ k]
3.2. SOLUTION

and less prems A.run-reachable[of i]
by auto

have [simp]: tkt-thread ts c = tt using tkt-thread-eq[OF R UTT].
note [simp] = tt<N

have A.can-step tt (s i) by (simp add: never-blocked)
hence ?case proof (cases rule: A.rstep-cases)
  case (other t') — Another thread than tt makes a step.

The current ticket and tt’s state remain the same

hence [simp]: c' = c ∧ ts' tt = ts tt
using has-ticket-unique[OF R UTT, of t']
unfolding A.rstep-def
using holds-only-current[OF R, of t']
by (force elim!: alstep_cases astep-sng_cases)

Thus, tt is still using the current ticket

have [simp]: tkt-thread ts' c = tt
using UTT tkt-thread-eq[OF R', of tt c] by auto

Only the distance to tt’’s next step has decreased

have (Suc i, i) ∈ lrel
unfolding lrel-def tweight-def by (simp add: other)

And we can apply the induction hypothesis

with less.IH[of Suc i] ⟨t<N⟩ show ?thesis
apply (auto) using Suc-lessD by blast
next
  case THIS: this — The thread tt that uses the current ticket makes a step

  show ?thesis
  proof (cases ∃k'. ts tt = REL k')
    case True — tt has finished releasing the lock
    then have [simp]: ts tt = REL c
  using UTT by auto

Thus, current increases

have [simp]: c' = Suc c
using THIS apply —
unfolding A.rstep-def
apply (erule alstep_cases, erule astep-sng_cases)
by auto

But is still less than k

from A.invar-reachable[OF is-invar1 R] have k>c
apply (auto simp: invar1-def)
CHAPTER 3. ARRAY-BASED QUEUING LOCK

And we can apply the induction hypothesis

\[
\textbf{hence (Suc } i, i) \in \text{rel} \\
\text{unfolding \text{rel-def} by auto} \\
\text{with less.IH [of Suc } i \} t<N \} \text{ show ?thesis} \\
\text{apply (auto) using Suc-lessD by blast} \\
\text{next} \\
\text{case False — tt has acquired the lock, or started releasing it}
\]

Hence, current remains the same, but the weight of tt decreases

\[
\textbf{hence [simp] :} \\
\quad c' = c \\
\quad \land \text{tweight (ts tt)} > \text{tweight (ts' tt)} \\
\quad \land \text{has-ticket (ts' tt) c} \\
\quad \text{using THIS UTT apply —} \\
\quad \text{unfolding A.rstep-def} \\
\quad \text{apply (erule alstep.cases, erule astep-sng.cases)} \\
\quad \text{by (auto simp: has-ticket-def twight-def)}
\]

\(tt\) still holds the current lock

\[
\textbf{have [simp]:} \text{tkt-thread ts'} = tt \\
\text{using tkt-thread-eq [OF R' tt<N, of c] by simp}
\]

And we can apply the IH

\[
\textbf{have (Suc } i, i) \in \text{rel unfolding \text{rel-def} by auto} \\
\text{with less.IH [of Suc } i \} t<N \} \text{ show ?thesis} \\
\text{qed (auto) using Suc-lessD by blast} \\
\text{qed}
\]

\[
\text{ultimately show ?case by blast}
\]

\[
\text{qed}
\]

3.2.3 Refinement 2: Bounding the Counters

We fix the \(k\) from the task description, which must be positive

\[
\text{consts k::nat} \\
\text{specification (k) k-not0 [simp]: } k \neq 0 \text{ by auto} \\
\text{lemma k-gt0 [simp]: } 0 < k \text{ by (cases k) auto}
\]

System’s state: Current ticket, next ticket, thread states

\[
\text{type-synonym \text{bstate} = nat \times nat \times (nat \Rightarrow thread)}
\]

The step relation of a single thread
inductive bstep-sng where
  enter-wait: bstep-sng (c,n,INIT) (c,(n+1) mod (k*N),WAIT (n mod N))
  loop-wait: c≠tk ⇒ bstep-sng (c,n,WAIT tk) (c,n,WAIT tk)
  exit-wait: bstep-sng (c,n,WAIT c) (c,n,HOLD c)
  start-release: bstep-sng (c,n,HOLD tk) (c,n,REL tk)
  release: bstep-sng (c,n,REL tk) ((tk+1) mod N,n,INIT)

The step relation of the system, labeled with the thread \( t \) that performs the step

inductive blstep for \( t \) where

\[
\begin{align*}
& [ t < N; bstep-sng (c,n,ts t) (c',n',s') ] \\
& \quad \implies blstep t (c,n,ts) (c',n',ts(t:=s'))
\end{align*}
\]

Initial state of the system

definition bs₀ ≡ \( (0,0,λ\cdot\text{INIT}) \)

interpretation B: system bs₀ blstep.

lemma b-never-blocked: B.can-step l s ←→ l < N
  apply (cases s; cases tts s l; simp)
  unfolding B.can-step-def
  apply (clarsimp simp: blstep.simps bstep-sng.simps; blast)+ done

interpretation B: df-system bs₀ blstep
  apply unfold-locales
  subgoal for s
  using b-never-blocked[of 0 s]
  unfolding B.can-step-def
  by auto
  done

Simulation

We show that the abstract system simulates the concrete one.

A few lemmas to ease the automation further below

lemma nat-sum-gtZ-iff[simp]:
  finite s ⇒ sum f s ≠ (0::nat) ←→ (∃x∈s. f x ≠ 0)
  by simp

lemma n-eq-Suc-subs1-conv[simp]: n = Suc (n − Suc 0) ←→ n≠0 by auto

lemma mod-mult-mod-0[simp]: x mod (k \* N) mod N = x mod N
  by (meson dvd-eq-mod-eq-0 mod-mod-cancel mod-mult-self2-is-0)

lemma mod-eq-imp-eq-aux: b mod N = (a::nat) mod N ⇒ a≤b ⇒ b < a+N ⇒ b = a
  by (metis Groups.add-ac add-0-right)
le-add-diff-inverse less-diff-conv2 nat-minus-mod
   nat-minus-mod-plus-right mod-if

lemma mod-eq-imp-eq:
[ [ b ≤ x; x < b + N; b ≤ y; y < b + N; x mod N = y mod N ] ] ⇒ x=y

proof -
   assume a1: b ≤ y
   assume a2: y < b + N
   assume a3: b ≤ x
   assume a4: x < b + N
   assume a5: x mod N = y mod N
   have f6: x < y + N
      using a4 a1 by linarith
   have y < x + N
      using a3 a2 by linarith
   then show ?thesis
      using f6 a5 by (metis (no-types) mod-eq-imp-eq-aux nat-le-linear)
qed

Map the ticket of a thread

fun map-ticket where
  map-ticket f INIT = INIT
  | map-ticket f (WAIT tk) = WAIT (f tk)
  | map-ticket f (HOLD tk) = HOLD (f tk)
  | map-ticket f (REL tk) = REL (f tk)

lemma map-ticket-addsims[simp]:
  map-ticket f t = INIT ←→ t=INIT
  map-ticket f t = WAIT tk ←→ (∃tk'. tk=f tk' ∧ t=WAIT tk')
  map-ticket f t = HOLD tk ←→ (∃tk'. tk=f tk' ∧ t=HOLD tk')
  map-ticket f t = REL tk ←→ (∃tk'. tk=f tk' ∧ t=REL tk')
  by (cases t; auto)+

We define the number of threads that use a ticket

fun ni-weight :: thread ⇒ nat where
  ni-weight INIT = 0 | ni-weight - = 1

lemma ni-weight-le1[simp]: ni-weight s ≤ Suc 0
  by (cases s) auto

definition num-ni ts ≡ \sum_{i=0..<N} ni-weight (ts i)
lemma num-ni-init[simp]: num-ni (λ -. INIT) = 0 by (auto simp: num-ni-def)

lemma num-ni-upd:
  t<N ⇒ num-ni (ts(t:=s)) = num-ni ts - ni-weight (ts t) + ni-weight s
  by (auto
      simp: num-ni-def if-distrib[of ni-weight] sum.If-cases algebra-simps
      simp: sum-diff1-nat
  )
3.2. SOLUTION

\textbf{lemma num-ni-nz-if}: \([t < N; ts t \neq \text{INIT}] \implies \text{num-ni ts} \neq 0\)

\begin{verbatim}
apply (cases ts t)
by (simp-all add: num-ni-def force+)
\end{verbatim}

\textbf{lemma num-ni-leN}: \(\text{num-ni ts} \leq N\)

\begin{verbatim}
apply (clarsimp simp: num-ni-def)
apply (rule order-trans)
apply (rule sum-bounded-above[where \(K=1\)])
apply auto
\end{verbatim}

We provide an additional invariant, considering the distance of \(c\) and \(n\). Although we could probably get this from the previous invariants, it is easy enough to prove directly.

\textbf{definition invar3} \(\equiv \lambda (c,n,ts). n = c + \text{num-ni ts}\)

\textbf{lemma is-invar3}: \(\text{A.is-invar invar3}\)

\begin{verbatim}
apply (rule)
subgoal by (auto simp: invar3-def as 0-def)
subgoal for s s'
  apply clarify
  apply (erule alstep.cases, erule astep-sng.cases)
  apply (auto simp: invar3-def num-ni-upd)
  using holds-only-current by fastforce
\end{verbatim}

We establish a simulation relation: The concrete counters are the abstract ones, wrapped around.

\textbf{definition sim-rel1} \(\equiv \lambda (c,n,ts) (ci,ni,tsi).
\begin{align*}
  ci &= c \mod N \\
  ni &= n \mod (k*N) \\
  tsi &= (\text{map-ticket} (\lambda t. t \mod N)) o ts
\end{align*}\)

\textbf{lemma sraux}: \(\text{sim-rel1 (c,n,ts) (ci,ni,tsi)} \implies ci = c \mod N \land ni = n \mod (k*N)\)

\begin{verbatim}
by (auto simp: sim-rel1-def Let-def)
\end{verbatim}

\textbf{lemma sraux2}: \([\text{sim-rel1 (c,n,ts) (ci,ni,tsi); t<N}] \implies tsi t = \text{map-ticket} (\lambda x. x \mod N) (ts t)\)

\begin{verbatim}
by (auto simp: sim-rel1-def Let-def)
\end{verbatim}

\textbf{interpretation sim1}: \(\text{simulation1 \ as_0 alstep bs_0 blstep sim-rel1}\)

\textbf{proof unfold-locales}

\textbf{show sim-rel1 as_0 bs_0}

\begin{verbatim}
  by (auto simp: sim-rel1-def as_0-def bs_0-def)
\end{verbatim}

\textbf{next}
fix as bs t bs'
assume Ra-aux: Areachable as
   and Rc-aux: Breachable bs
   and SIM: sim-rel1 as bs
   and CS: blstep t bs bs'

obtain c n ts where [simp]: as=(c,n,ts) by (cases as)
obtain ci ni tsi where [simp]: bs=(ci,ni,tsi) by (cases bs)
obtain ci' ni' tsi' where [simp]: bs'=(ci',ni',tsi') by (cases bs')
from Ra-aux have Ra: Areachable (c,n,ts) by simp
from Rc-aux have Rc: Breachable (ci,ni,tsi) by simp

from CS have t<N by cases auto

have [simp]: n = c + num-ni ts
   using A.invar-reachable[OF is-invar3 Ra, unfolded invar3-def] by simp

have AUX1: c≤tk tk<c+N if ts t = WAIT tk for tk
   using that A.invar-reachable[OF is-invar1 Ra]
   apply (auto simp: invar1-def)
   using t<N: apply fastforce
   using ⟨t<N⟩ num-ni-leN[of ts] by fastforce

from SIM CS have ∃as'. alstep t as as' ∧ sim-rel1 as as' bs'
   apply simp
   apply (erule blstep.cases)
   apply (erule bstep-sng.cases)
   apply clasimp-all
   subgoal
   apply (intro exI conjI)
   apply (rule alstep.intros)
   apply (simp add: sim-rel1-def Let-def)
   apply (simp add: sraux sraux2)
   apply (rule astep-sng.enter-wait)
   apply (simp add: sim-rel1-def; intro conjI ext)
   apply (auto simp: sim-rel1-def Let-def mod-simps)
   done
   subgoal
   apply (clarsimp simp: sraux sraux2)
   apply (intro exI conjI)
   apply (rule alstep.intros)
   apply (simp add: sim-rel1-def Let-def)
   apply clasimp
   apply (rule astep-sng.loop-wait)
   apply (auto simp: sim-rel1-def Let-def mod-simps)
   done
   subgoal
   apply (clarsimp simp: sraux sraux2)
   subgoal for tk′
3.2. SOLUTION

apply (subgoal-tac tk' = c)
apply (intro exI conjI)
apply (rule alstep.intros)
apply (simp add: sim-relI-def Let-def)
apply clarsimp
apply (rule astep-sng.exit-wait)
apply (auto simp: sim-rel1-def Let-def mod-simps)
done

subgoal
apply (clarsimp simp: sraux sraux2)
apply (intro exI conjI)
apply (rule alstep.intros)
apply (simp add: sim-relI-def Let-def)
apply clarsimp
apply (rule astep-sng.start-release)
apply (auto simp: sim-rel1-def Let-def mod-simps)
done

subgoal
apply (clarsimp simp: sraux sraux2)
apply (intro exI conjI)
apply (rule alstep.intros)
apply (simp add: sim-relI-def Let-def)
apply clarsimp
apply (rule astep-sng.release)
apply (auto simp: sim-rel1-def Let-def mod-simps)
done

then show \( \exists \text{as}'. \sim-rel1 \text{as}' \text{bs}' \wedge \text{alstep t as as}' \) by blast

next
fix as bs l
assume A.reachable as B.reachable bs sim-relI as bs A.can-step l as
then show B.can-step l bs using b-never-blocked never-blocked by simp
qed

Transfer of Properties

We transfer a few properties over the simulation, which we need for the next refinement step.

lemma xfer-locks-ticket:
assumes locks-ticket (map-ticket (\lambda t. t mod N) (ts t)) tki
obtains tk where tki = tk mod N locks-ticket (ts t) tk
using assms unfolding locks-ticket-def
by auto
lemma \(b\)-holds-only-current:
\[
[B.\text{reachable } (c, n, ts); t < N; \text{locks-ticket } (ts t) \, tk] \implies tk = c
\]
apply (rule sim1.xfer-reachable, assumption)
apply (clarsimp simp: sim-rel1-def)
apply (erule xfer-locks-ticket)+
using holds-only-current by blast

lemma \(b\)-mutual-exclusion\':
\[
[B.\text{reachable } s; t < N; t' < N; t \neq t'; \text{locks-ticket } (ts s t) \, tk; \text{locks-ticket } (ts s t') \, tk']
\implies False
\]
apply (rule sim1.xfer-reachable, assumption)
apply (clarsimp simp: sim-rel1-def)
apply (erule xfer-locks-ticket)+
apply (drule (3) mutual-exclusion\'; simp)
done

lemma xfer-has-ticket:
assumes has-ticket (map-ticket (\(\lambda t. \, t \mod N\)) (ts t)) tki
obtains tk where tki = tk mod N has-ticket (ts t) tk
using assms unfolding has-ticket-def
by auto

lemma has-ticket-in-range:
assumes Ra: A.\text{reachable } (c.n,ts) and \(t < N\) and U: has-ticket (ts t) tk
shows c \leq tk \land tk < c + N
proof –

have [simp]: n = c + num-ni ts
using A.invar-reachable[OF is-invar3 Ra, unfolded invar3-def] by simp

show c \leq tk \land tk < c + N
using A.invar-reachable[OF is-invar1 Ra] U
apply (auto simp: invar1-def)
using t < N apply fastforce
using t < N; num-ni-leN[of ts] by fastforce
qed

lemma \(b\)-has-ticket-unique:
\[
[B.\text{reachable } (c.i,ni.tsi); t < N; \text{has-ticket } (tsi t) \, tk; t' < N; \text{has-ticket } (tsi t') \, tk']
\implies t' = t
\]
apply (rule sim1.xfer-reachable, assumption)
apply (auto simp: sim-rel1-def)
subgoal for c n ts
apply (erule xfer-has-ticket)+
apply simp
subgoal for tk tk'
apply (subgoal-tac tk = tk')
apply simp
3.2. SOLUTION

```isar
cases s; cases tts s l; simp
```
We establish another invariant that states that the ticket numbers are bounded.

**Definition** \textit{invar4}

\[
\equiv \lambda (c,n,ts). c \lt N \land (\forall t \lt N. \forall tk. \text{has-ticket} (ts t) \implies tk \lt N)
\]

**Lemma** \textit{is-invar4}: \textit{B.is-invar invar4}

apply (rule)

subgoal by (auto simp: \textit{invar4-def \& \& \& \& bs0-def})

subgoal for \textit{s s'}

apply clarify

apply (erule \textit{blstep.cases}, erule \textit{bstep-sng.cases})

unfolding \textit{invar4-def}

apply safe

apply (metis \textit{N-gt0 fun-upd-other fun-upd-same mod-mod-trivial nat-mod-len has-ticket-simps(2)})

apply (metis \textit{fun-upd-triv})

apply (metis \textit{fun-upd-other fun-upd-same has-ticket-simps(3)})

apply (metis \textit{fun-upd-other fun-upd-same has-ticket-def has-ticket-simps(4)})

using \textit{mod-less-divisor apply blast}

apply (metis \textit{fun-upd-apply thread.distinct(1) thread.distinct(3)}

\textit{thread.distinct(5) has-ticket-def})

done

We define a predicate that describes that a thread of the system is at the release sequence point — in this case, the array does not have a set bit, otherwise, the set bit corresponds to the current ticket.

**Definition** \textit{is-REL-state} \equiv \lambda ts. \exists t \lt N. \exists tk. ts t = \textit{REL} tk

**Lemma** \textit{is-REL-state-simps[simp]}:

\[
t \lt N \implies \text{is-REL-state} (ts(t:=\textit{REL} tk))
\]

\[
t \lt N \implies \neg \text{is-REL} (ts t) \implies \neg \text{is-REL} s'
\]

\[
\implies \text{is-REL-state} (ts(t:=s')) \leftrightarrow \text{is-REL-state} ts
\]

unfolding \textit{is-REL-state-def}

apply (auto; fail) 

apply auto 

by (metis \textit{thread.distinct(12)})

**Lemma** \textit{is-REL-state-aux1}:

assumes \textit{R}: \textit{B.reachable (c,n,ts)}

assumes \textit{REL}: \textit{is-REL-state ts}

assumes \textit{t \lt N} and [simp]: \textit{ts t = \textit{WAIT} tk}

shows \textit{tk \neq c}

using \textit{REL unfolding} \textit{is-REL-state-def}

apply clarify

subgoal for \textit{t' tk'}

using \textit{b-has-ticket-unique[OF R \& \& \& \& ts t = \textit{WAIT} tk']}

using \textit{b-holds-only-current[OF R, of t' tk']}
by (auto)
done

lemma is-REL-state-aux2:
assumes R: B.reachable (c,n,ts)
assumes A: t<N ts t = REL tk
shows ¬is-REL-state (ts;:=INIT))
using b-holds-only-current[OF R] A
using b-mutual-exclusion[OF R]
apply (clarsimp simp: is-REL-state-def)
by fastforce

Simulation relation that implements current ticket by array

definition sim-rel2 ≡ λ(c,n,ts) (ci,ni,tsi).
(if is-REL-state ts then
 ci = replicate N False
else
 ci = (replicate N False)(c:=True)
)
∧ ni = n
∧ tsi = ts

interpretation sim2: simulationI bs0 blstep cs0 clstep sim-rel2
proof unfold-locales
show sim-rel2 bs0 cs0
by (auto simp: sim-rel2-def bs0-def cs0-def is-REL-state-def)

next
fix bs cs t cs'
assume Rc-aux: B.reachable bs
and Rd-aux: C.reachable cs
and SIM: sim-rel2 bs cs
and CS: clstep t cs cs'

obtain c n ts where [simp]: bs=(c,n,ts) by (cases bs)
obtain ci ni tsi where [simp]: cs=(ci,ni,tsi) by (cases cs)
obtain ci' ni' tsi' where [simp]: cs'=(ci',ni',tsi') by (cases cs')
from Rc-aux have Rc: B.reachable (c,n,ts) by simp
from Rd-aux have Rd: C.reachable (ci,ni,tsi) by simp

from CS have t<N by cases auto

have [simp]: tk<N if ts t = WAIT tk for tk
using B.invar-reachable[OF is-invar4 Rc] that t<N
by (auto simp: invar4-def)
have HOLD-AUX: tk=c if ts t = HOLD tk for tk
using b-holds-only-current[OF Rc t t<N, of tk] that by auto
have REL-AUX: tk=c if ts t = REL tk t t<N for tk
using b-holds-only-current[OF Rc t t<N, of tk] that by auto
have \([\text{simp}]: c < N \text{ using } B \text{ invar-reachable[}\text{OF is-invar4 } R_c]\] by (auto simp: invar4-def)

have \([\text{simp}]:\) replicate \(N\) False \(\neq\) replicate \(N\) False\([c := True]\]
replicate \(N\) False\([c := True]\) \(\neq\) replicate \(N\) False
apply (auto simp: list-eq-iff-nth-eq nth-list-update)
using \(c < N\) by blast+

have \([\text{simp}]:\) replicate \(N\) False\([c := True]\] \(! d \leftarrow d = c \text{ if } d < N \text{ for } d\]
using that by (auto simp: list-eq-iff-nth-eq nth-list-update)

have \([\text{simp}]:\) replicate \(N\) False\([tk := False]\) = replicate \(N\) False \text{ for } \(tk\]
by (auto simp: list-eq-iff-nth-eq nth-list-update')

from SIM CS have \(\exists bs'. \text{ blstep } t \text{ bs bs'} \land \text{ sim-rel2 bs' cs'}\]
apply simp
apply (subst (asm) sim-rel2-def)
apply (erule clstep.cases)
apply (erule cstep-sng.cases)
apply clarsimp-all
subgoal
apply (intro exI conjI)
apply (rule blstep.intros)
apply (simp)
applyclarsimp
apply (rule bstep-sng.enter-wait)
apply (auto simp: sim-rel2-def split: if-splits)
done
subgoal for \(tk'\)
apply (intro exI conjI)
apply (rule blstep.intros)
apply (simp)
applyclarsimp
apply (rule bstep-sng.loop-wait)
subgoal
apply (clarsimp simp: sim-rel2-def split: if-splits)
apply (frule (2) is-REL-state-aux1[\text{OF } R_c])
by simp
subgoal by (auto simp: sim-rel2-def split: if-splits)
done
subgoal
apply (intro exI conjI)
apply (rule blstep.intros)
apply (simp)
apply (clarsimp simp: split: if-splits)
3.2. SOLUTION

```plaintext
apply (rule bstep-sng.exit-wait)
apply (auto simp: sim-rel2-def split: if-splits)
done

subgoal
  apply (intro exI conjI)
  apply (rule blstep.intros)
  apply (simp)
  apply clarsimp
  apply (rule bstep-sng.start-release)
  apply (auto simp: sim-rel2-def dest: HOLD-AUX split: if-splits)
done

subgoal
  apply (intro exI conjI)
  apply (rule blstep.intros)
  apply (simp)
  apply clarsimp
  apply (rule bstep-sng.release)
  apply (auto
    simp: sim-rel2-def
    dest: is-REL-state-aux2[OF Rc]
    split: if-splits)
  by (metis fun-upd-triv is-REL-state-simps(1))
done

then show \( \exists bs'. \ sim-rel2 \ bs' \ cs' \land \ blstep \ t \ bs \ bs' \) by blast
```

next

fix bs cs l
assume B. reachable bs C. reachable cs sim-rel2 bs cs B. can-step l bs
then show C. can-step l cs using c-never-blocked b-never-blocked by simp
qed

3.2.5 Transfer Setup

We set up the final simulation relation, and the transfer of the concepts used in the correctness statements.

```plaintext
definition sim-rel ≡ sim-rel1 OO sim-rel2
interpretation sim: simulation as0 alstep cs0 clstep sim-rel
  unfolding sim-rel-def
  by (rule sim-trans) unfold-locales

lemma xfer-holds:
  assumes sim-rel s cs
  shows is-HOLD (tts cs t) \iff \ is-HOLD (tts s t)
  using assms unfolding sim-rel-def sim-rel1-def sim-rel2-def
  by (cases tts cs t) auto

lemma xfer-waits:
```
assumes \( \text{sim-rel } s \; cs \)
shows \( \text{is-WAIT} \; (\text{tt} \; cs \; t) \iff \text{is-WAIT} \; (\text{tt} \; s \; t) \)
using assms unfolding sim-rel-def sim-rel1-def sim-rel2-def
by (cases \( \text{tt} \; cs \; t \)) auto

**Lemma:** \text{xfer-init:}

assumes \( \text{sim-rel } s \; cs \)
shows \( \text{tt} \; cs \; t = \text{INIT} \iff \text{tt} \; s \; t = \text{INIT} \)
using assms unfolding sim-rel-def sim-rel1-def sim-rel2-def
by auto

### 3.2.6 Main Theorems

#### Trusted Code Base

Note that the trusted code base for these theorems is only the formalization of the concrete system as defined in Section 3.2.4. The simulation setup and the abstract systems are only auxiliary constructions for the proof.

For completeness, we display the relevant definitions of reachability, runs, and fairness here:

\[
C.\; \text{step } s \; s' = (\exists l. \; \text{clstep } l \; s \; s')
\]

\[
C.\; \text{reachable} \equiv C.\; \text{step}^* \; cs_0
\]

\[
C.\; \text{is-lrun } l \; s \equiv s_0 = cs_0 \land (\forall i. \; \text{clstep } (l \; i) \; (s \; i) \; (s (\text{Suc } i)))
\]

\[
C.\; \text{is-run } s \equiv \exists l. \; C.\; \text{is-lrun } l \; s
\]

\[
C.\; \text{is-lfair } ls \; ss \equiv \forall l \; i. \; \exists j \geq i. \; \neg C.\; \text{can-step } l \; (ss \; j) \lor ls \; j = l
\]

\[
C.\; \text{is-fair-run } s \equiv \exists l. \; C.\; \text{is-lrun } l \; s \land C.\; \text{is-lfair } l \; s
\]

#### Safety

We show that there is no reachable state in which two different threads hold the lock.

**Theorem** \text{final-mutual-exclusion:} \([C.\; \text{reachable } s; \]

\[
t < N; \; t' < N; \; t \neq t'; \; \text{is-HOLD } (\text{tt} \; s \; t); \; \text{is-HOLD } (\text{tt} \; s' \; t')\]

\[
] \implies \text{False}
\]

apply (erule sim.xfer-reachable)
apply (simp add: xfer-holds)
by (erule (5) mutual-exclusion)

#### Fairness

We show that, whenever a thread \( t \) draws a ticket, all other threads \( t' \) waiting for the lock will be granted the lock before \( t \).

**Theorem** \text{final-fair:}
3.2. SOLUTION

assumes RUN: C.is-run s
assumes ACQ: t < N and tts (s i) t = INIT and is-WAIT (tts (s (Suc i)) t)
— Thread t draws ticket in step i
assumes HOLD: i < j and is-HOLD (tts (s j) t)
— Thread t holds lock in step j
assumes WAIT: t' < N and is-WAIT (tts (s i) t')
— Thread t' waits for lock at step i
obtains l where i < l and l < j and is-HOLD (tts (s l) t')
— Then, t' gets lock earlier
using RUN
proof (rule sim.xfer-run)
fix as
assume Ra: A.is-run as and SIM[rule-format]: ∀ i. sim-rel (as i) (s i)

note XFER = xfer-init[OF SIM] xfer-holds[OF SIM] xfer-waits[OF SIM]

show ?thesis
using assms
apply (simp add: XFER)
apply (erule (6) fair[OF Ra])
apply (erule (1) that)
apply (simp add: XFER)
done
qed

Liveness

We show that, for a fair run, every thread that waits for the lock will eventually
hold the lock.

theorem final-progress:
assumes FRUN: C.is-fair-run s
assumes WAIT: t < N and is-WAIT (tts (s i) t)
shows ∃ j > i. is-HOLD (tts (s j) t)
using FRUN
proof (rule sim.xfer-fair-run)
fix as
assume Ra: A.is-fair-run as
and SIM[rule-format]: ∀ i. sim-rel (as i) (s i)

note XFER = xfer-init[OF SIM] xfer-holds[OF SIM] xfer-waits[OF SIM]

show ?thesis
using assms
apply (simp add: XFER)
apply (erule (1) progress[OF Ra])
done
qed

end