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
Gap Buffer

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 \texttt{left()} and \texttt{right()} move the cursor by one character; \texttt{insert()} places a character at the beginning of the gap \( a[l] \); \texttt{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 insert(x: char)
    if l == r then
        // see extended task
        grow()
    end-if
    a[l] := x
    l := l + 1
end-procedure

procedure right()
    // your task: similar to left()
    // but pay attention to the
    // order of statements
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.
CHAPTER 1. GAP BUFFER

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

**Extended verification task.** Implement the operation \texttt{grow()}, specify its behavior in a way that lets you verify \texttt{insert()} in a modular way (i.e. not by referring to the implementation of \texttt{grow()}), and verify that \texttt{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 \texttt{System.arraycopy()} in Java), consider using these primitives instead of the loops in the pseudo-code below.

```plaintext
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 \( \text{′a gap-buffer} = \text{nat} \times \text{nat} \times \text{′a list} \)

Abstraction Relation

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

- definition \( \text{gap-α} \equiv \lambda (l,r,buf). \text{BUF} \ (\text{take } l \text{ buf}) \ (\text{drop } r \text{ buf}) \)
- definition \( \text{gap-invar} \equiv \lambda (l,r,buf). l\leq r \land r\leq \text{length buf} \)
- abbreviation \( \text{gap-rel} \equiv \text{br gap-α} \text{ gap-invar} \)

Empty

- definition \( \text{empty1} \equiv \text{RETURN} \ (0,0,[]) \)
- lemma \( \text{empty1-correct}: (\text{empty1}, \text{RETURN empty}) \in \langle \text{gap-rel} \rangle \text{nres-rel} \)

Left

- definition \( \text{move-left1} \equiv \lambda (l,r,buf). \text{doN} \{ \)
  
  \hspace{1em} \text{if} \ l\neq 0 \ \text{then} \ \text{doN} \{ \)
  
  \hspace{2em} \text{ASSERT} (r-1<\text{length buf} \land 1-1<\text{length buf}); \)
  
  \hspace{2em} \text{RETURN} \ (l-1,r-1,\text{buf}[r-1::=\text{buf}![l-1]])

  \}

  \}

  \}

  \}

lemma \( \text{move-left1-correct}: (\text{move-left1}, \text{RETURN o move-left}) \in \text{gap-rel} \to \langle \text{gap-rel} \rangle \text{nres-rel} \)

by (auto simp: in-br-conv gap-α-def gap-invar-def)

Right

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

\text{lemma} move-right1-correct: 
(move-right1.\text{RETURN} o move-right) \in \text{gap-rel} \to (\text{gap-rel})\text{nres-rel} 
\text{apply clarsimp} 
\text{unfolding} move-right1-def 
\text{apply refine-vcg} 
\text{unfolding} gap-\alpha\text{-def} gap-invar-def 
\text{apply} \ (\text{auto} 
  \text{simp: in-br-conv hd-drop-conv-nth take-update-last} 
  \text{split: prod.split}) 
\text{by} \ (\text{simp add: drop-Suc tl-drop})

\text{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} \{ 
  \text{let} \ b = \text{op-array-replicate} (\text{length buf} + K) \text{ default}; 
  b \leftarrow \text{mop-list-blit} \text{ buf} 0 b 0 l; 
  b \leftarrow \text{mop-list-blit} \text{ buf} r b (r+K) (\text{length buf} - r); 
  \text{RETURN} (l, r+K, b) 
\} 

\text{lemma} grow1-correct: \text{THEN} SPEC-trans, refine-vcg]: 
\text{assumes} \ \text{gap-invar gb} 
\text{shows} \ \text{grow1 K gb} \leq \text{SPEC} (\lambda 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\text{-def} gap-invar-def can-insert-def 
\text{apply} \ (\text{auto simp: op-list-blit-def}) 
\text{done}

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

\text{lemma} insert1-correct:
(insert1.RETURN oo insert) ∈ Id → gap-rel → (gap-rel)nres-rel
apply clarsimp
unfolding insert1-def
apply refine-vcg
unfolding insert-def gap-α-def gap-invar-def can-insert-def
apply (auto simp: in-br-conv take-update-last split: prod.split)
done

Delete

definition delete1
≡ λ(l,r,buf). if l>0 then RETURN (l−1,r,buf) else RETURN (l,r,buf)
lemma delete1-correct:
(delete1.RETURN o delete) ∈ gap-rel → (gap-rel)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)

1.2.4 Imperative Arrays and Executable Code

abbreviation gap-impl-assn ≡ nat-assn ×ₖ nat-assn ×ₖ array-assn id-assn
definition gap-assn A
≡ hr-comp (hr-comp gap-impl-assn gap-rel) ((the-pure A)textbuffer-rel)

context
notes gap-assn-def[symmetric fcomp-norm-unfold]
begin
sepref-definition empty-impl
is uncurry0 empty1 :: unit-assn ×ₖ gap-impl-assn
unfolding empty1-def array.fold-custom-empty
by sepref

sepref-definition move-left-impl
is move-left1 :: gap-impl-assn₁ →ₖ gap-impl-assn
unfolding move-left1-def by sepref

sepref-definition move-right-impl
is move-right1 :: gap-impl-assn₁ →ₖ gap-impl-assn
unfolding move-right1-def by sepref

sepref-definition insert-impl
is uncurry insert1 :: id-assn ×ₖ gap-impl-assn₁ →ₖ gap-impl-assn
unfolding insert1-def grow1-def by sepref
We inline `grow1` here

```text
```

```text
sepref-definition delete-impl
  is delete1 :: gap-impl-assn → gap-impl-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)) ∈ unit-assn → gap-assn ?A
(move-left-impl, RETURN o move-left) ∈ (gap-assn ?A) → gap-assn ?A
(move-right-impl, RETURN o move-right) ∈ (gap-assn ?A) → gap-assn ?A
CONSTRANIT is-pure ?A ⇒
(uncurry Challenge1.insert-impl, uncurry (RETURN o Challenge1.insert)) ∈ ?A → gap-assn ?A
(delete-impl, RETURN o delete) ∈ (gap-assn ?A) → gap-assn ?A
```

```text
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
```

### 1.2.5 Simple Client

```text
definition client ≡ RETURN (fold (λf. 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)
```

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

```text
sepref-definition client-impl
  is uncurry0 client :: unit-assn → gap-assn id-assn
  unfolding client-def fold.simps id-def comp-def
  by sepref
```
1.3. SHORTER SOLUTION

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 Is-
abelle, 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
CHAPTER 1. GAP BUFFER

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.

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

lemma move-left1-correct:
\((\text{move-left1}, \text{RETURN} \circ \text{move-left}) \in \text{gap-rel} \rightarrow (\text{gap-rel})nres-rel\)
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)

by (smt Cons-nth-drop-Suc Suc-pred append assoc append-Cons append-Nil
  diff-Suc-less drop-update-cancel hd-drop-com-nth list-update
  less-le-trans nth-list-update-eq take-hd-drop)

Right

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

lemma move-right1-correct:
\((\text{move-right1}, \text{RETURN} \circ \text{move-right}) \in \text{gap-rel} \rightarrow (\text{gap-rel})nres-rel\)
apply clarsimp
unfolding move-right1-def
apply refine-vcg
apply (auto simp: in-br-conv gap-α-def gap-invar-def
  rule nth-equalityI)
apply (smt-all add: Cons-nth-drop-Suc take-update-last)
done

Insert and Grow

definition can-insert ≡ \( \lambda (l, r, buf). \) \( l < r \)
**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 \( \text{buf}[l := \text{buf} ! r] \), 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 ≤ (SPEC (λgb′.
  gap-invar gb′
  ∧ gap-α gb′ = gap-α gb
  ∧ (K>0 → can-insert gb′)))
unfolding grow1-def
apply refine-vcg
using assms
unfolding gap-α-def gap-invar-def can-insert-def
apply (auto simp: op-list-blit-def)
done

**Definition**  insert1 \( x \equiv \lambda (l,r,buf). \) doN {
  (l,r,buf) ← if (l=0) then grow1 (length buf + 1) \( l,r,buf \) else RETURN \( l,r,buf \);
  ASSERT (l<length buf);
  RETURN \( l+1,r,buf[:l:=x] \)
}

**Lemma**  insert1-correct:
( insert1, RETURN oo insert ) ∈ Id → gap-rel → (gap-rel)nres-rel
apply clarsimp
unfolding insert1-def
apply refine-vcg — VCG knows the rule for grow1 already
unfolding insert-def gap-α-def gap-invar-def can-insert-def
apply (auto simp: in-br-conv take-update-last split: prod split)
done

Delete

**Definition**  delete1
≡ \( \lambda (l,r,buf). \) if l>0 then RETURN \( \langle l-1,r,buf \rangle \) else RETURN \( l,r,buf \)

**Lemma** delete1-correct:
(delete1, RETURN o delete) ∈ gap-rel → (gap-rel)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-assnd →, a gap-impl-assn
unfolding move-left1-def by sepref

sepref-definition move-right-impl
is move-right1 :: gap-impl-assnd →, a gap-impl-assn
unfolding move-right1-def by sepref

sepref-definition insert-impl
is uncurry insert1 :: id-assn k∗ a gap-impl-assnd →, a gap-impl-assn
unfolding insert1-def grow1-def by sepref
— We inline grow1 here

sepref-definition delete-impl
is delete1 :: gap-impl-assnd →, 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 k∗ a gap-assn →, a gap-assn

(delete-impl, RETURN ◦ delete) ∈ gap-assn →, a gap-assn
1.3. SHORTER SOLUTION

1.3.5 Executable Code

Isabelle/HOL can generate code in various target languages.

```plaintext
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
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:

```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 implies 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)
nvalid' xs  valid' (B # xs)
nvalid' 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 n'. l = \text{replicate } n \ R @ l')\text{ if } n < \text{length } l
using that by (auto simp: list-eq-iff-nth-eq nth-append intro: ext[where x = drop n l])

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

Main Case Analysis on @term valid

lemma valid-split:
valid l \iff
l = [] \lor
(l!0 = B \land 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 valid (drop (j + 1) l))
unfolding valid-eq-valid'[symmetric]
apply standard
subgoal
by (erule valid'.cases) (auto simp: nth-append nth-Cons 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 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 ∧ valid l \} = \{ [R] \}
  by (auto intro: valid.intros elim: valid.cases simp: replicate-Cons-eq)

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

lemma lc3-aux: \{ l. length l = 3 ∧ valid l \} = \{ [B,B,B], [B,R,R] \}
  by (auto 4 4 intro: valid.intros valid-3R 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 lc3-aux unfolding lcount-def by simp-all

The Recursion Case

lemma finite-valid-length:
  finite \{ l. length l = n ∧ valid l \} (is finite ?S)
proof –
  have ?S ⊆ lists \{ R, B \} ∩ \{ 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 ∧ valid l \} ≠ \{ \} (is ?S ≠ \{ \})
  using valid-line-just-B[of n] by force

lemma replicate-unequal-aux:
  replicate x R @ B # l ≠ replicate y R @ B # 1' (is ?l ≠ ?r) if \( x < y \) for 1'
proof –
  have ?l ! x = B ?r ! x = R
    using that by (auto simp: nth-append)
  then show ?thesis
    by auto
qed
2.2. SOLUTION

\textbf{lemma valid-prepend-B-iff:}
valid (B # xs) \leftrightarrow valid xs
by \(\text{auto intro: valid.intros elim: valid.cases simp: Cons-replicate-eq Cons-eq-append-conv}\)

\textbf{lemma lcrec: lcount n = lcount \((n-1) + 1 + (\sum_{i=3..<n} \text{lcount} \((n-i-1))\) if \(n>3\)
proof —
have \(\{\text{l. length } l = n \land \text{valid } l\}\)
  \(= \{\text{l. length } l = n \land \text{valid } (tl \ l) \land \text{!}0=B\}\)
  \cup \{\text{l. length } l = n \land 
  (\exists \ i. i < n \land i \geq 3 \land (\forall k < i. \text{!}k = R) \land \text{!}i = B \land \text{valid } (\text{drop } (i + 1) \ l))\}
  \cup \{\text{l. length } l = n \land (\forall i<n. \text{!}i=R)\}\)
  \(\text{is } \text{?A = ?B or ?D or ?C}\)
using \(\text{on > 3 by } (\text{subst valid-split}) \text{ auto}\)

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

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

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

unfolding inj-on-def by auto

have *:
\((\lambda l. \text{replicate } x \ R @ B \ # l) \cdot \{l. \text{length } l = n - \text{Suc } x \land \text{valid } l\} \cap
(\lambda l. \text{replicate } y \ R @ B \ # l) \cdot \{l. \text{length } l = n - \text{Suc } y \land \text{valid } l\} = \{\}\)
if \(3 \leq x < y < n\) for \(x y\)

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

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

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

  case 1
  show ?case
  unfolding pairwise-def disjoint-def
  proof (clarsimp, goal-cases)
    case prems: \((1 x y)\)
    from prems show ?case
    apply
    apply (rule linorder-cases[of x y])
    apply (simp; assumption)
    apply (simp; fail)
    done
  qed

  next
  case 3
  show ?case
  proof (subst sum.reindex, unfold inj-on-def, clarsimp, goal-cases)
    case prems: \((1 x y)\)
    with *[of x y] *[of x y] valid-line-aux[of n - \text{Suc } x] show ?case
    by (rule linorder-cases[of x y], auto)

  next
  case 2
  then show ?case
  by (simp add: lcount-def card-image[of inj])
  qed

  qed (auto intro: finite-subset[OF \(<\text{finite-valid-length}>\)])

show ?thesis
apply (subst lcount-def)
unfolding \(?A = \rightarrow \ ?B = \rightarrow \ ?C = \rightarrow \ D-eq\)
apply (subst card-Un-disjoint)

  apply (blast intro: finite-subset[OF \(<\text{finite-valid-length}>\)])

subgoal
using Cons-replicate-eq[of B - n R] replicate-unequal-aux by fastforce
apply (subst card-Un-disjoint)
apply (blast intro: finite-subset[OF - finite-valid-length]) +

unfolding 1 2 3
by (auto simp: Cons-replicate-eq Cons-eq-append-conv)
qd

2.2.3 Verification of Program

Inner Loop: Summation

definition sum-prog Φ l u f ≡
nfoldli [l..<u] (λ-. True) (λ i s. doN {
  ASSERT (Φ i);
  RETURN (s+f i)
}) 0

lemma sum-spec[THEN SPEC-trans, refine-vcg]:
assumes l\less u
assumes \And i. l\less i \implies i\less u \implies Φ i
shows sum-prog Φ l u f \less SPEC (λ r. r=(\sum i=l..<u. f i))
unfolding sum-prog-def
supply nfoldli-upt-rule[where l=λ j s. s=(\sum i=l..<j. f i), refine-vcg]
apply refine-vcg
using assms
apply auto
done

Main Program

definition icount M ≡ 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←nfoldli [4..<M+1] (λ-. True) (λ n c. doN {
    ASSERT (M<length c);
    RETURN (c!M)
  }) c;
  ASSERT (\forall i\leq M. c!i = lcount i);
}

ASSERT (M< length c);
RETURN (c!M)
Abstract Correctness Statement

theorem icount-correct: \( M > 2 \implies icount M \leq SPEC (\lambda r. r = lc\text{count} M) \)
unfolding icount-def
thm nfoldli-upt-rule
supply nfoldli-upt-rule[where
\[ I = \lambda n c. \text{length} c = M + 1 \land (\forall i < n. c!i = lc\text{ount} i), \text{refine-vcg} \]
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 \( ^k \rightarrow \text{nat-assn} \)
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 (lc\text{ount} \( M \)).

theorem icount-impl-correct:
\[ M > 2 \implies \{ \text{emp} \} icount-impl M \{ \lambda r. \uparrow (r = lc\text{ount} M) \} \]
proof –
  note \( A = \text{icount-impl}\text{.refine[to-hnr, THEN hn-refineD]} \)
  note \( A = A[\text{unfolded autoref-tag-defs}] \)
  note \( A = A[\text{unfolded hn-ctxt-def pure-def, of M M, simplified}] \)
  note \( \text{[sep-heap-rules]} = A \)

  assume \( M > 2 \)

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

Code Export

- `export-code icount-impl in SML-imp module-name Tiling`
- `export-code icount-impl in OCaml-imp module-name Tiling`
- `export-code icount-impl in Haskell module-name Tiling`
- `export-code icount-impl in Scala-imp module-name Tiling`

2.2.5 Alternative Problem Specification

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

```plaintext
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-1) = R; l!i = R; l!(i+1) = R ] \[ valid-point i
| [ 2 <= i; i < length l; l!i-2 = R; l!(i-1) = R; l!i = R ] \[ valid-point i
| [ i < length l; l!=B ] \[ valid-point i

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

lemma valid-lineI:
  assumes \[ \forall i. i < length l \[ 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 (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)
using that unfolding valid-line-def
proof (safe, goal-cases)
case (1 i)
with valid-drop-B[of xs i + 1] show ?case
  by auto
qed

lemma valid-line-prepend-B-iff:
valid-line (B # xs) ←→ valid-line xs
using valid-line-prepend-B valid-line-drop-B
by metis

lemma cases-valid-line:
assumes
  l = [] ∨
  (|0 = B ∧ valid-line (tl l)) ∨
  length l ≥ 3 ∧ (∀ i < length l. l ! i = R) ∨
  (∃ j < length l. j ≥ 3 ∧ (∀ i < j. l ! i = R) ∧ l ! j = B ∧ valid-line (drop (j + 1) l))
  (is ?a ∨ ?b ∨ ?c ∨ ?d)
shows valid-line l
proof
  from assms consider (empty) ?a | (B) ¬ ?a ∧ ?b | (all-red) ?c | (R-B) ?d
  by blast
  then show ?thesis
  proof cases
    case empty
    then show ?thesis
      by (simp add: valid-line-def)
  next
    case B
    then show ?thesis
      by (cases l) (auto simp: valid-line-prepend-B-iff)
  next
    case prems: all-red
    show ?thesis
    proof (rule valid-lineI)
      fix i assume i < length l
      consider i = 0 | i = l | i > l
      by atomize_elim auto
      then show valid-point l i
        using i < -> prems by cases (auto 4 4 intro: valid-point.intros)
    qed
  next
    case R-B
    then obtain j where j:
2.2. SOLUTION

\[ j < \text{length } l \leq j (\forall i < j. l! i = R) l! j = B \text{ valid-line (drop } (j + 1) l) \]
by blast
show ?thesis
proof (rule valid-lineI)
  fix i
  assume \[ i < \text{length } l \]
  with \[ j \geq 3 \]
  consider \[ i \leq j - 3 \]
  have \[ l! i \neq R \]
  by atomize_elim auto
then show \[ \text{valid-point } l i \]
proof cases
  case 5
  with \[ \text{valid-line } \rightarrow \ a < \text{length } b \]
  have \[ \text{valid-point (drop } (j + 1) l) (i - j - 1) \]
  unfolding valid-line-def by auto
  then show ?thesis
  using \[ j > j \]
  by cases (auto intro: valid-point.intros)
qed (use \( j \) in \( \text{auto intro: valid-point.intros} \))
qed

lemma valid-line-cases:
\[ l = [] \lor \]
\( (\| 0 = B \land \text{valid-line (tl } l)) \lor \]
\( \text{length } l \geq 3 \land (\forall i < \text{length } l. l! i = R) \lor \]
\( (\exists j < \text{length } l. j \geq 3 \land (\forall i < j. l! i = R) \land l! j = B \land \text{valid-line (drop } (j + 1) l)) \)
if \[ \text{valid-line } l \]
proof (cases \( l = [] \))
  case True
  then show ?thesis
  by (simp add: valid-line-def)
next
  case False
  from \[ \text{valid-line } l \land l \neq [] \]
  have \[ l = B \# tl l \]
  by (cases \( l \))
  with \[ \text{valid-line } b \]
  True show ?thesis
  by (metis valid-line-prepend-B-iff)
next
  case False
  from \[ \text{valid-line } b \land l \neq [] \]
  have \[ \text{valid-point } l \]
  unfolding valid-line-def by auto
  with \[ \text{False have } \text{red-start: } \text{length } l \geq 3 \land 0 = R \land 1 = R \land 2 = R \]
  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 \text{B-ge-3: } i \geq 3 \text{ if } l ! i = B \text{ for } i
  proof
    consider i = 0 | i = 1 | i = 2 | i \geq 3
    by \text{atomize-elim auto}
    then show i \geq 3
      using \text{red-start } l ! i = B \text{ 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 \text{Min-in, auto}
  have \forall i < \?j. l ! i = R
  proof
    {  
      fix i assume i < \?j l ! i = B
      then have i \geq 3
        by (auto intro: B-ge-3)
      with \< i < \?j: l ! i = B \text{ red-start } \?j \in ?S \text{ have } i \in ?S
        by auto
      then have \?j \leq i
        by (auto intro: Min-le)
      with \< i < \?j: have False
        by \text{simp}
    }
    then show \?thesis
      by (auto intro: color.exhaust)
  qed
  with \< ?j \in ?S\ obtain j where j: j < \text{length } l j \geq 3 \forall i < j. l ! i = R l ! j = B
  by blast
  moreover have \text{valid-line } (\text{drop } (j + 1) l)
  proof (rule valid-lineI)
    fix i assume i < \text{length } (\text{drop } (j + 1) l)
    with j \text{ valid-line } l \text{ have valid-point } l (j + i + 1)
    unfolding \text{valid-line-def} \text{ by auto}
    then show valid-point (\text{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 | i = 1 | i > 1
        by \text{atomize-elim auto}
        then show \?thesis
          using j prems by cases (auto intro: valid-point.intros)
  qed (auto intro: valid-point.intros)
2.2. SOLUTION

\begin{verbatim}
qed
ultimately show \textit{thesis}
by auto
qed
lem
lem
lem
lem

lemma valid-line-split:
valid-line \( l \) if
\( l = [] \lor (l!0 = B \land valid-line (tl l)) \lor \)
\( \text{length } l \geq 3 \land (\forall i < \text{length } l. l!i = R) \lor \)
\( (\exists j < \text{length } l. j \geq 3 \land (\forall i < j. l!i = R) \land l!j = B \land valid-line (drop (j + 1) l)) \)
using valid-line-cases cases-valid-line by blast

Connection to the easier definition given above

lemma valid-valid-line:
valid \( l \) if
valid-line \( l \)
by (induction \( l \) rule: length-induct, subst valid-line-split, subst valid-split, auto)
\end{verbatim}
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 \( \text{pass[]} \) 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} \text{ }\text{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 \( \text{pass[]} \) are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of \( \text{pass[]} \) 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:

```plaintext
procedure abql_init()
    for i = 1 to N - 1 do
        pass[i] := false
    end-for
    pass[0] := true
    next := 0
end-procedure

function abql_acquire()
    var my_ticket := fetch_and_add(next,1) mod N
    while not pass[my_ticket] do
        end-while
    return my_ticket
end-function

procedure abql_release(my_ticket)
    pass[my_ticket] := false
    pass[(my_ticket + 1) mod N] := true
end-procedure
```

Each thread that acquires the lock must eventually release it by calling \text{abql_release(my_ticket)},

[^33]:
where `my_ticket` is the return value of the earlier call of `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 `abql_release()` can be moved at the end of `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 $N$ 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 \textit{mod \(N\)} and \textit{mod \((k \times N)\)} 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

\[
\text{consts} \quad N :: \text{nat}
\]

\[
\text{specification (} N \text{)} \quad \text{N-not0[simp, intro!]}: N \neq 0 \text{ by auto}
\]

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

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

\[
\text{datatype thread =}
\]

\[
\text{INIT}
\]

| \text{is-WAIT: WAIT (ticket: nat)}
| \text{is-HOLD: HOLD (ticket: nat)}
| \text{is-REL: REL (ticket: nat)}
3.2.2 Refinement 1: Ticket Lock with Unbounded Counters

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

\textbf{type-synonym} \textit{astate} = \textit{nat} \times \textit{nat} \times (\textit{nat} \Rightarrow \textit{thread})

\textbf{abbreviation} \textit{cc} ≡ \textit{fst}

\textbf{abbreviation} \textit{nn s} ≡ \textit{fst} (\textit{snd s})

\textbf{abbreviation} \textit{tts s} ≡ \textit{snd} (\textit{snd s})

The step relation of a single thread

\textbf{inductive} \textit{astep-sng} \textbf{where}

\textit{enter-wait}: \textit{astep-sng} (\textit{c}, \textit{n}, \textit{INIT}) (\textit{c}, (\textit{n}+1), \textit{WAIT} \textit{n})

| \textit{loop-wait}: \textit{c} \neq \textit{k} \implies \textit{astep-sng} (\textit{c}, \textit{WAIT} \textit{k}) (\textit{c}, \textit{n}, \textit{WAIT} \textit{k})

| \textit{exit-wait}: \textit{astep-sng} (\textit{c}, \textit{n}, \textit{HOLD} \textit{c}) (\textit{c}, \textit{n}, \textit{HOLD} \textit{c})

| \textit{start-release}: \textit{astep-sng} (\textit{c}, \textit{n}, \textit{HOLD} \textit{k}) (\textit{c}, \textit{n}, \textit{REL} \textit{k})

| \textit{release}: \textit{astep-sng} (\textit{c}, \textit{n}, \textit{REL} \textit{k}) (\textit{k}+1, \textit{n}, \textit{INIT})

The step relation of the system

\textbf{inductive} \textit{alstep} \textbf{for} \textit{t} \textbf{where}

\[ \begin{array}{l}
\text{$t$<}\text{N;} \text{astep-sng} (\textit{c}, \textit{n}, \textit{ts} \textit{t}) (\textit{c'}, \textit{n}', \textit{ts'}) \\
\implies \text{alstep} \textit{t} (\textit{c}, \textit{n}, \textit{ts}) (\textit{c'}, \textit{n}', \textit{ts}(t:=s'))
\end{array} \]

Initial state of the system

\textbf{definition} \textit{aS0} ≡ (0, 0, λ-. \textit{INIT})

\textbf{interpretation} \textit{A: system aS0 alstep}.

In our system, each thread can always perform a step

\textbf{lemma} \textit{never-blocked}: \textit{A.can-step} \textit{l s} \longleftrightarrow \textit{l}<\text{N}

\textbf{apply} (\textit{cases s; cases tts s l; simp})

\textbf{unfolding} \textit{A.can-step-def}

\textbf{apply} (\textit{clarsimp simp; alstep.simps astep-sng.simps; blast})+

\textbf{done}

Thus, our system is in particular deadlock free

\textbf{interpretation} \textit{A: df-system aS0 alstep}

\textbf{apply} \textit{unfold-locales}

\textbf{subgoal for} \textit{s}

\textbf{using} \textit{never-blocked[of 0 s]}

\textbf{unfolding} \textit{A.can-step-def}

\textbf{by} \textit{auto}

\textbf{done}

\textbf{Safety: Mutual Exclusion}

Predicates to express that a thread uses or holds a ticket

\textbf{definition} \textit{has-ticket} \textit{s} \textit{k} ≡ \textit{s}=\textit{WAIT} \textit{k} \lor \textit{s}=\textit{HOLD} \textit{k} \lor \textit{s}=\textit{REL} \textit{k}
3.2. SOLUTION

lemma has-ticket-simps[simp]:

\neg has-ticket INIT k
has-ticket (WAIT k) k' \rightarrow k'=k
has-ticket (HOLD k) k' \rightarrow k'=k
has-ticket (REL k) k' \leftarrow k'=k
unfolding has-ticket-def by auto

definition locks-ticket s k \equiv s=\text{HOLD} k \lor s=\text{REL} k
lemma locks-ticket-simps[simp]:

\neg locks-ticket INIT k
\neg locks-ticket (WAIT k) k'
locks-ticket (HOLD k) k' \leftarrow k'=k
locks-ticket (REL k) k' \rightarrow 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).
c \leq n 
\land (\forall k. t<N \land has-ticket (ts t) k \rightarrow 
c \leq k \land k < n
\land (\forall t'. k', t'<N \land has-ticket (ts t') k' \land t' \neq t' \rightarrow k \neq k')
\land (\forall k. k \neq c \rightarrow \neg locks-ticket (ts t) k)
)

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 astep-sng.cases)
apply (clarsimp-all simp: invar1-def)
apply fastforce
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

\textbf{theorem} mutual-exclusion: \[ A.\text{reachable } s; \]
\[ t<N; t'<N; t\neq t'; \text{is-HOLD (tts } s \text{ t);} \text{is-HOLD (tts } s \text{ t'}) \]
\[ \implies \text{False} \]
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))

\textbf{lemma} mutual-exclusion': \[ A.\text{reachable } s; \]
\[ t<N; t'<N; t\neq t'; \]
\[ \text{locks-ticket (tts } s \text{ t) tk; locks-ticket (tts } s \text{ t') tk'} \]
\[ \implies \text{False} \]
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

\textbf{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 \( j \).

\textbf{lemma} fair-aux:
\textbf{assumes} R: A.is-run \( s \)
\textbf{assumes} A: \( i\leq j \) cc (s i) \( k \) cc (s j) \( t\leq N \) tts (s i) \( t=\text{WAIT } k \)
\textbf{shows} \( \exists l. i \leq l \land l < j \land \text{tts (s l) } t = \text{HOLD } k \)
\textbf{proof} –
interpret A: run as0 alstep s by unfold-locales fact
from A show ?thesis
proof (induction j \( \sim i \) arbitrary: i)
case 0
then show ?case by auto
next
case (Suc i')

hence [simp]; \( i' = j - \text{ Suc } i \) by auto
\textbf{note} IH=Suc.invar1(1)[OF this]


obtain \( t' \) where \( \text{alstep} t' (s\ i) (s\ (\text{Suc}\ i)) \) by (rule \text{A.stepE})
then show \(?\text{case using Suc.prems}\)
proof cases
  case \((c\ e\ n\ ts\ c'\ n'\ s')\)
  note [simp] = \(I(1,2,3)\)
  from \(\text{A.run-invar[OF is-invar1, of i]}\) have \(\text{invar1\ (c,n,ts)}\) by auto
note \(II = \text{this}\)\[\text{unfolded\ invar1-def,\ simplified}\]

from \(I(4)\) show \(?\text{thesis}\)
proof (cases rule: \text{astep-sng,cases})
  case enter-wait
  then show \(?\text{thesis}\)
    using \(\text{IH Suc.prems}\) apply (auto)
    by (metis \(1(2)\ \text{Suc-leD Suc-lessI fst-conv leD thread.distinct(1)}\))
next
  case \((\text{loop-wait}\ k)\)
  then show \(?\text{thesis}\)
    using \(\text{IH Suc.prems}\) apply (auto)
    by (metis \(1(2)\ \text{Suc-leD Suc-lessI fst-conv leD}\))
next
  case exit-wait
  then show \(?\text{thesis}\)
    apply (cases \(t'=t\))
    subgoal
    using \(\text{Suc.prems}\) apply clarsimp
    by (metis \(1(2)\ \text{Suc-leD Suc-lessI fst-conv fun-upd-same leD less-or-eq-imp-le snd-conv}\))
    subgoal
    using \(\text{Suc.prems\ IH}\)
    apply auto
    by (metis \(1(2)\ \text{Suc-leD Suc-lessI fst-conv leD}\))
done
next
  case \((\text{start-release}\ k)\)
  then show \(?\text{thesis}\)
    using \(\text{IH Suc.prems}\) apply (auto)
    by (metis \(1(2)\ \text{Suc-leD Suc-lessI fst-conv leD thread.distinct(7)}\))
next
  case \((\text{release}\ k)\)
  then show \(?\text{thesis}\)
    apply (cases \(t'=t\))
    using \(II\ \text{IH Suc.prems}\) apply (auto)
    by (metis \(1(2)\ 1(3)\ \text{Suc-leD Suc-leI Suc-lessI fst-conv locks-ticket-simps(4) le-antisym not-less-eq-eq has-ticket-simps(2) has-ticket-simps(4)}\))
qed
qed
**Lemma s-case-expand:**

\[(\text{case } s \text{ of } (c, n, ts) \Rightarrow P \ c \ n \ ts) = P (\text{cc } s) (\text{nn } s) (\text{ttts } s)\]

by \((\text{auto split: prod.splits})\)

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

**Lemma fair-aux2:**

assumes \(\text{RUN: } A.\text{is-run } s\)
assumes \(\text{ACQ: } t < N \text{ ts } (s \text{ (Suc } i)) \Rightarrow \text{INIT ts } (s \text{ (Suc } i)) \Rightarrow \text{WAIT } k\)
assumes \(\text{HOLD: } i < j \text{ ts } (s \text{ (Suc } i)) \Rightarrow \text{HOLD } k\)
assumes \(\text{WAIT: } t' < N \text{ ts } (s \text{ (Suc } i)) \Rightarrow \text{WAIT } k'\)

obtains \(l\) where \(i < l < j \text{ ts } (s \text{ (Suc } i)) \Rightarrow \text{HOLD } k'\)

**Proof:**

interpret \(A: \text{run } as_0\) alstep \(s\) by unfold-locales fact

from \(\text{ACQ WAIT have [simp]: } t \neq t' \neq t\) by auto

from \(\text{ACQ have [simp]: } nn (s \text{ (Suc } i)) = \text{Suc } k\)
\(\land \text{cc } (s \text{ (Suc } i)) = \text{cc } (s \text{ i}) \land \text{ttts } (s \text{ (Suc } i)) = (\text{ttts } (s \text{ i}))(t := \text{WAIT } k)\)
apply (rule-tac A.alstepE[of \(i\)])
apply (erule alstep.cases)
apply (erule astep-sng.cases)
by (auto simp: nth-list-update split: if-splits)

from \(A.\text{run-invar}[\text{OF is-invar1, of } i]\) have \(\text{invar1 } (s \text{ i})\) by auto
note \(I_1 = \text{this[unfolded invar1-def, unfolded s-case-expand, simplified]}\)

from \(\text{ACQ HOLD have Suc } i \neq j\) by auto with \(\text{HOLD have Suc } i < j\) by auto

have \(X_1: \text{cc } (s \text{ i}) \leq k'\) using \(I_1\) \(\text{WAIT}\) by fastforce

have \(X_2: k' < \text{cc } (s \text{ j})\)
using \(A.\text{run-invar}[\text{OF is-invar1, of } j, \text{unfolded invar1-def s-case-expand}]\)
using \(k' < k \cdot t < N \Rightarrow \text{HOLD}(2)\)
apply clarsimp
by (metis locks-ticket-simps \(3\) has-ticket-simps \(3\))

from \(\text{fair-aux}[\text{OF RUN } \text{Suc } i < j, \text{of } k' \text{ t'}, \text{simplified}]\) obtain \(l\) where
\(l \geq \text{Suc } i  l < j \text{ ts } (s \text{ l}) t' = \text{HOLD } k'\)
using \(\text{WAIT } X_1 X_2\) by auto

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

qed
3.2. SOLUTION

**Lemma** find-hold-position:

- **Assumes** `RUN`:
  - `A.is-run s`
- **Assumes** `WAIT`:
  - `t<N tss (s i) t = WAIT tk`
- **Assumes** `NWAIT`:
  - `i<j tss (s j) t ≠ WAIT tk`
- **Obtains** `l` where
  - `i<l l≤j tss (s l) t = HOLD tk`

**Proof** —

- interpret `A`: run as `0 alstep s by unfold-locales fact from WAIT(2) NWAIT`

  have `∃ l. i<l ∧ l≤j ∧ tss (s l) t = HOLD tk`

- **Proof** (induction `j—i arbitrary`):
  - case `0`
    - then show `?case` by `auto`
  - next
    - case `(Suc i)`
      - hence `[simp]: i' = 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)`
          - 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 tss (s i) t=INIT is-WAIT (tss (s (Suc i)) t)`
    - Thread `t` calls `acquire` in step `i`
- **Assumes** `HOLD`:
  - `i<j is-HOLD (tss (s j) t)`
    - Thread `t` holds lock in step `j`
- **Assumes** `WAIT`:
  - `t'<N is-WAIT (tss (s i) t')`
    - Thread `t'` waits for lock at step `i`
- **Obtains** `l` where
  - `i<l l<j is-HOLD (tss (s l) t')`
    - Then, `t'` gets lock earlier
proof

obtain \( k \) where \( W_k : \text{tts} (s \text{ (Suc } i)) \ t = \text{WAIT} \ k \) using \( ACQ \)
  by (cases \( \text{tts} (s \text{ (Suc } i)) \ t \) auto)

obtain \( k' \) where \( W_k' : \text{tts} (s \ i) \ t' = \text{WAIT} \ k' \) using \( \text{WAIT} \)
  by (cases \( \text{tts} (s \ i) \ t' \) auto)

from \( ACQ \ HOLD \) have \( \text{Suc } i \neq j \) by auto with \( HOLD \) have \( \text{Suc } i < j \) by auto

obtain \( j' \) where \( H'_1 : \text{Suc } i < j' \) \& \( j' \leq j \) \& \( \text{tts} (s \ j') \ t = \text{HOLD} \ k \)
  apply (rule \( \text{find}-\text{hold}-\text{position}\) [OF \( \text{RUN} \ a < N \) \& \( W_k < \text{Suc } i < j \)])
  using \( HOLD(2) \) by auto

show \( ?\text{thesis} \)
  apply (rule \( \text{fair}-\text{aux2}\) [OF \( \text{RUN} \ \text{ACQ}(1,2) \ W_k - H'(3) \ \text{WAIT}(1) \ W_k'])
  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

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

\[ \text{lemma} \ is-invar2: A.\text{is-invar} \ invar2 \]
  apply \( \text{rule} \)
  subgoal by (auto simp: invar2-def as0-def)
  subgoal for \( s \ s' \)
    apply (clarsimp simp: invar2-def)
    apply (erule astep.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.\text{invar-reachable}[OF \text{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

\[ \text{corollary} \ \text{current-lock-used}: \]
3.2. SOLUTION

assumes $R$: $A$ reachable $(c, n, ts)$
assumes $\text{WAIT} : t < N$ $ts$ $t = \text{WAIT} k$

obtains $t'$ where $t' < N$ has-ticket $(ts t') c$
using $A$ invar-reachable[OF is-invar2 $R$]
and $A$ invar-reachable[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
has-ticket-simps(2))

Used tickets are unique (Corollary from invariant 1)

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

We define the thread that holds a specified ticket

definition tkt-thread $\equiv \lambda ts k. \text{THE} t. t < N \land \text{has-ticket} (ts t) k$

lemma tkt-thread-eq:
assumes $R$: $A$ reachable $(c, n, ts)$
assumes $A$: $t < N$ has-ticket $(ts t) k$
shows $tkt-thread ts k = t$
using has-ticket-unique[OF $R$]
unfolding tkt-thread-def
using $A$ by auto

lemma holds-only-current:
assumes $R$: $A$ reachable $(c, n, ts)$
assumes $A$: $t < N$ locks-ticket $(ts t) k$
shows $k = c$
using $A$ invar-reachable[OF is-invar1 $R$] $A$ 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 $s \equiv$
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 $\text{FRUN}$: $A$ is-fair-run $s$
assumes $A$: $t < N$ is-WAIT $(tts (s i) t)$
shows $\exists j > i$. is-HOLD $(tts (s j) t)$
proof -
interpret $A$: fair-run $as_0$ alstep $s$ by unfold-locales fact

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

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

\[
\text{have } \text{wf lrel unfolding } \text{lrel-def by auto}
\]

\[
\text{then show } \forall \text{thesis using } A(I) \text{ Wk}
\]

\[
\text{proof (induction } i) \quad \text{case (less } i)
\]

We name the components of this and the next state

\[
\text{obtain } c n ts \text{ where } \text{simp]: } s i = (c,n,ts) \text{ by (cases } s i)
\]

\[
\text{from } A.\text{run-reachable[of } i] \text{ have } R: A.\text{reachable} (c,n,ts) \text{ by simp}
\]

\[
\text{obtain } c' n' ts' \text{ where } \text{simp]: } s (\text{Suc } i) = (c',n',ts')
\]

\[
\text{by (cases } s (\text{Suc } i))
\]

\[
\text{from } A.\text{run-reachable[of Suc } i] \text{ have } R': A.\text{reachable} (c',n',ts')
\]

\[
\text{by simp}
\]

\[
\text{from less.prems have } \text{WAIT[simp]: } ts t = \text{WAIT } k \text{ by simp}
\]

\[
\{\]

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

\[
\text{assume } ts' t \neq \text{WAIT } k
\]

\[
\text{hence } ts' t = \text{HOLD } k \text{ using less.prems}
\]

\[
\text{apply } (\text{rule-tac } A.\text{stepE[of } i])
\]

\[
\text{apply } (\text{auto elim!}: \text{alstep.cases astep-sng.cases split: if-splits})
\]

\[
\text{done}
\]

\[
\text{hence } \forall \text{case by auto}
\]

\[
\text{moreover } \{\]

\[
\text{assume [simp]: } ts' t = \text{WAIT } k
\]

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

\[
\text{obtain } tt \text{ where } UTT: tt<N \text{ has-ticket } (ts tt) c
\]

\[
\text{using current-lock-used[of } c n ts t k]\]
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] = «t< 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)
by (metis UTT WAIT `ts tt = REL c` le-neq-implies-less less.prems(1) thread.distinct(9) has-ticket-simps(2))

And we can apply the induction hypothesis

```isar
case False — tt has acquired the lock, or started releasing it
```

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

```isar
case False
```

```isar
And we can apply the IH
```

```isar
3.2.3 Refinement 2: Bounding the Counters
```

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

```isar
consts k::nat
```

```isar
specification (k) k-not0[simp]: k\neq0 by auto
```

```isar
lemma k-gt0[simp]: 0<k by (cases k) auto
```

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

```isar
type-synonym bstate = nat × nat × (nat ⇒ 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 \( \text{blstep} \) for \( t \) where

\[
\text{blstep} \begin{cases} \begin{array}{l} t<N; \text{bstep-sng } (c,n,t) \ (c',n',s') \\ \end{array} \end{cases}
\Rightarrow \text{blstep} \ (c,n,t) \ (c',n',s'\ (t:=s'))
\]

Initial state of the system

definition \( b_0 \equiv (0, 0, \lambda \cdot \text{INIT}) \)

interpretation \( B: \text{system } b_0 \text{ blstep .} \)

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

interpretation \( B: \text{df-system } b_0 \text{ blstep} \)
apply unfold-locales
subgoal for \( s \)
  using \( b\)-never-blocked[of \( 0 \) \( s \)]
  unfolding \( B\text{-can-step-def} \)
  by auto
done

Simulation

We show that the abstract system simulates the concrete one.

A few lemmas to ease the automation further below

lemma \( \text{nat-sum-gtZ-iff[simp]} \): finite \( s \Rightarrow \text{sum f s } \not\equiv (0::\text{nat}) \Longleftrightarrow (\exists x \in s. f x \neq 0) \)
by simp

lemma \( \text{n-eq-Suc-sub1-conv[simp]} \): \( n = \text{Suc } (n - \text{Suc } 0) \Longleftrightarrow n\neq0 \) by auto

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

lemma \( \text{mod-eq-imp-eq-aux[simp]} \): \( h \text{ mod } N = (a::\text{nat}) \mod N \Rightarrow a\leq b \Rightarrow b<a+N \Rightarrow b=a \)
by (auto simp add: mod-eq-dvd-iff-nat le-imp-diff-is-add)
**chapter 3. Array-Based Queuing Lock**

**lemmas mod-eq-imp-eq:**

\[ [b \leq x; x < b + N; b \leq y; y < b + N; x \mod N = y \mod N] \implies x = y \]

**proof**

- **assume a1:** \( b \leq y \)
- **assume a2:** \( y < b + N \)
- **assume a3:** \( b \leq x \)
- **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**

\[
\begin{align*}
\text{map-ticket } f \text{ INIT} & = \text{INIT} \\
\text{map-ticket } f \text{ (WAIT } tk) & = \text{WAIT } (f \text{ tk)} \\
\text{map-ticket } f \text{ (HOLD } tk) & = \text{HOLD } (f \text{ tk)} \\
\text{map-ticket } f \text{ (REL } tk) & = \text{REL } (f \text{ tk)}
\end{align*}
\]

**lemma** map-ticket-addsimp[simp]:

\[
\begin{align*}
\text{map-ticket } f \text{ t} = \text{INIT} & \iff t = \text{INIT} \\
\text{map-ticket } f \text{ t} = \text{WAIT } tk & \iff (\exists tk'. tk = f tk' \land t = \text{WAIT } tk') \\
\text{map-ticket } f \text{ t} = \text{HOLD } tk & \iff (\exists tk'. tk = f tk' \land t = \text{HOLD } tk') \\
\text{map-ticket } f \text{ t} = \text{REL } tk & \iff (\exists tk'. tk = f tk' \land t = \text{REL } tk')
\end{align*}
\]

**by (cases t; auto)+**

We define the number of threads that use a ticket

**fun** ni-weight :: thread \( \Rightarrow \) nat **where**

\[
\text{ni-weight } \text{INIT} = 0 \mid \text{ni-weight } \text{- } 1
\]

**lemma** ni-weight-le1[simp]: \( \text{ni-weight } s \leq \text{Suc } 0 \)

**by (cases s) auto**

**definition** num-ni ts \( \equiv \sum_{i=0..<N} \text{ni-weight } (ts \ i) \)

**lemma** num-ni-init[simp]: \( \text{num-ni } (\lambda - \text{. INIT}) = 0 \) **by (auto simp: num-ni-def)**

**lemma** num-ni-upd:

\( t < N \implies \text{num-ni } (ts(t:=s)) = \text{num-ni } ts - \text{ni-weight } (ts \ t) + \text{ni-weight } s \)

**by (auto**

\[
\begin{align*}
\text{simp: num-ni-def if-distrib[of ni-weight] sum.If-cases algebra-simps simp: sum-diff1-nat}
\end{align*}
\]

**)****

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

**apply (cases ts t)**
3.2. SOLUTION

by (simp-all add: num-ni-def) force+

lemma num-ni-leN: num-ni ts ≤ N
  apply (clarsimp simp: num-ni-def)
  apply (rule order-trans)
  apply (rule sum-bounded-above[where K=1])
  apply auto
  done

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.

definition invar3 ≡ λ(c,n,ts). n = c + num-ni ts

lemma is-invar3: A.is-invar invar3
  apply (rule)
  subgoal by (auto simp: invar3-def as0-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
  done

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

definition sim-rel1 ≡ λ(c,n,ts) (ci,ni,tsi).
  ci = c mod N
  ∧ ni = n mod (k*N)
  ∧ tsi = (map-ticket (λt. t mod N)) o ts

lemma sraux:
  sim-rel1 (c,n,ts) (ci,ni,tsi) \implies ci = c mod N ∧ ni = n mod (k*N)
  by (auto simp: sim-rel1-def Let-def)

lemma sraux2: [sim-rel1 (c,n,ts) (ci,ni,tsi); t<N]
  \implies tsi t = map-ticket (λt. x mod N) (ts t)
  by (auto simp: sim-rel1-def Let-def)

interpretation sim1: simulationI as0 alstep bs0 bstep sim-rel1
proof unfold-locales
  show sim-rel1 as0 bs0
    by (auto simp: sim-rel1-def as0-def bs0-def)
next
fix as bs t bs'
assume Ra-aux: A.reachable as
  and Rc-aux: B.reachable bs
and SIM: \textit{sim-rel1} as bs
and CS: \textit{blstep} t bs bs'

\textbf{obtain} c n ts \textbf{where} \textbf{[simp]}: \textit{as} = (c,n,ts) \textbf{by} (cases as)
\textbf{obtain} ci ni tsi \textbf{where} \textbf{[simp]}: \textit{bs} = (ci,ni,tsi) \textbf{by} (cases bs)
\textbf{obtain} ci' ni' tsi' \textbf{where} \textbf{[simp]}: \textit{bs'} = (ci',ni',tsi') \textbf{by} (cases bs')

from Ra-aux \textbf{have} Ra: A.\textit{reachable} (c,n,ts) \textbf{by} simp
from Rc-aux \textbf{have} Rc: B.\textit{reachable} (ci,ni,tsi) \textbf{by} simp

from CS \textbf{have} t < N \textbf{by} cases auto

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

\textbf{have} AUX1: c \leq tk tk < c+N \textbf{if} ts t = \textit{WAIT} tk for tk
\textbf{using} that A.\textit{invar-reachable}[OF is-invar1 Ra]
\textbf{apply} (auto simp: invar1-def)
\textbf{using} t < N \textbf{apply} fastforce
\textbf{using} t < N; num-ni-leN[of ts] \textbf{by} fastforce

from SIM CS \textbf{have} \exists as'. \textit{alstep} t as as' \land \textit{sim-rel1} as' bs'
\textbf{apply} simp
\textbf{apply} (erule blstep.cases)
\textbf{apply} (erule bstep-sng.cases)
\textbf{apply} clarsimp-all
\textbf{subgoal}
\textbf{apply} (intro exI conjI)
\textbf{apply} (rule alstep.intros)
\textbf{apply} (simp add: sim-rel1-def Let-def)
\textbf{apply} (simp add: sraux sraux2)
\textbf{apply} (rule astep-sng.enter-wait)
\textbf{apply} (simp add: sim-rel1-def; intro conjI ext)
\textbf{apply} (auto simp: sim-rel1-def Let-def mod-simps)
\textbf{done}

\textbf{subgoal}
\textbf{apply} (clarsimp simp: sraux sraux2)
\textbf{apply} (intro exI conjI)
\textbf{apply} (rule astep.intros)
\textbf{apply} (simp add: sim-rel1-def Let-def)
\textbf{apply} sraux
\textbf{apply} (rule astep-sng.loop-wait)
\textbf{apply} (auto simp: sim-rel1-def Let-def mod-simps)
\textbf{done}

\textbf{subgoal}
\textbf{apply} (clarsimp simp: sraux sraux2)
\textbf{subgoal for} tk'
\textbf{apply} (subgoal-tac tk' = c)
\textbf{apply} (intro exI conjI)
\textbf{apply} (rule alstep.intros)
3.2. SOLUTION

apply \((\text{simp add: sim-rel1-def Let-def})\)
apply clarsimp
apply \(\text{(rule astep-sng,exit-wait)}\)
apply \(\text{(auto simp: sim-rel1-def Let-def mod-simps)}\) []
apply \(\text{(clarsimp simp: sim-rel1-def)}\)
apply \(\text{(erule mod-eq-imp-eq-aux)}\)
apply \(\text{(auto simp: AUX1)}\)
done

then show \(\exists a\.' \cdot \text{sim-rel1 as' \& alstep t as as'}\) by blast

next
fix as bs l
assume A. reachable as B. reachable bs sim-rel1 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 \((\lambda t \cdot \text{map-ticket} (\lambda t \cdot t \mod N) (ts t))\) 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 \(\text{(rule sim1.xfer-reachable, assumption)}\)
apply (clarsimp simp: sim-rel1-def)
apply (erule xfer-locks-ticket)+
using holds-only-current
by blast

lemma b-mutual-exclusion': \[B.\text{reachable } s; \]
  \[t < N; t' < N; t \neq t'; \text{locks-ticket } (ts\ s\ t)\ \text{tk}; \text{locks-ticket } (ts\ s\ t')\ \text{tk'}\]
  \[\implies False\]
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 \(\text{has-ticket}\ (\text{map-ticket} (\lambda t. t \mod N) (ts\ t))\ tki\)
  obtains \(tk\ \text{where } tki = tk \mod N\ \text{has-ticket } (ts\ t)\ tk\)
using assms unfolding has-ticket-def
by auto

lemma has-ticket-in-range:
  assumes Ra: \(A.\text{reachable } (c,n,ts)\ \text{and } t < N\ \text{and } U: \text{has-ticket } (ts\ t)\ tk\)
  shows \(c \leq tk \land tk < c+N\)
proof –
  have [simp]: \(n = c + \text{num-ni}\ ts\)
    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: \[B.\text{reachable } (ci,ni,tsi); \]
  \[t < N; \text{has-ticket } (tsi\ t)\ \text{tk}; t' < N; \text{has-ticket } (tsi\ t')\ \text{tk'}\]
  \[\implies t' = t\]
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
    apply (frule (4) has-ticket-unique, assumption)
    apply (frule (2) has-ticket-in-range[where \(tk = tk\)])
    apply (frule (2) has-ticket-in-range[where \(tk = tk'\)])
3.2. SOLUTION

3.2.4 Refinement 3: Using an Array

Finally, we use an array instead of a counter, thus obtaining the exact data structures from the challenge assignment.

Note that we model the array by a list of Booleans here.

System’s state: Current ticket array, next ticket, thread states
type-synonym cstate = bool list × nat × (nat ⇒ thread)

The step relation of a single thread

inductive cstep-sng where
  enter-wait: cstep-sng (p,n,INIT) (p,(n+1) mod (k*N),WAIT (n mod N))
  loop-wait: ¬p!tk ⇒ cstep-sng (p,n,WAIT tk) (p,n,WAIT tk)
  exit-wait: p!tk ⇒ cstep-sng (p,n,WAIT tk) (p,n,HOLD tk)
  start-release: cstep-sng (p,n,HOLD tk) (p[tk:=False],n,REL tk)
  release: cstep-sng (p,n,REL tk) (p[(tk+1) mod N := True],n,INIT)

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

inductive clstep for \( t \) where
  \[ t<N; cstep-sng (c,n,ts t) (c',n',s') \]
  \[ \implies clstep t (c,n,ts) (c',n',ts(t:=s')) \]

Initial state of the system

definition cs0 ≡ ((replicate N False)[0:=True], 0, λ-. INIT)

interpretation C: system cs0 clstep .

lemma c-never-blocked: C.can-step l s \( \iff \) \( l<N \)
  apply (cases s; cases tss s l; simp)
  unfolding C.can-step-def
  apply (clarsimp-all simp: clstep.simps cstep-sng.simps)
  by metis

interpretation C: df-system cs0 clstep
  apply unfold-locales
  subgoal for \( s \)
    using c-never-blocked[of 0 s]
    unfolding C.can-step-def
    by auto
  done

We establish another invariant that states that the ticket numbers are bounded.
CHAPTER 3. ARRAY-BASED QUEUING LOCK

**definition** invar4
\[ \equiv \lambda (c,n,ts). c<N \land (\forall t<N. \forall tk. \text{has-ticket} (ts t) tk \rightarrow tk<N) \]

**lemma** is-invar4: B.is-invar invar4

**apply** (rule)
**subgoal by** (auto simp: invar4-def bs0-def)
**subgoal for** s s'
  **apply** clarify
  **apply** (erule blstep.cases, erule bstep-sng.cases)
  **unfolding** invar4-def
  **apply** safe
  **apply** (metis N-gt0 fun-upd-apply has-ticket-simps(2) mod-less-divisor)
  **apply** (metis fun-upd-triv)
  **apply** (metis fun-upd-other fun-upd-same has-ticket-simps(3))
  **apply** (metis fun-upd-other fun-upd-same has-ticket-def has-ticket-simps(4))
  **using** mod-less-divisor **apply** blast
  **apply** (metis fun-upd-apply thread.distinct(1) thread.distinct(3)
    thread.distinct(5) has-ticket-def)
  **done**
  **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** is-REL-state \[ \equiv \lambda ts. \exists t<N. \exists tk. ts t = REL tk \]

**lemma** is-REL-state-simps[simp]:
\[ t<N \Rightarrow \text{is-REL-state} (ts(t:=REL tk)) \]
\[ t<N \Rightarrow \neg \text{is-REL} (ts t) \Rightarrow \neg \text{is-REL} s' \]
\[ \Rightarrow \text{is-REL-state} (ts(t:=s')) \leftrightarrow \text{is-REL-state} ts \]
**unfolding** is-REL-state-def
**apply** (auto; fail) []
**apply** auto []
**by** (metis thread.distinct(12))

**lemma** is-REL-state-aux1:
**assumes** R: B.reachable (c,n,ts)
**assumes** REL: is-REL-state ts
**assumes** t<N and [simp]: ts t = WAIT tk
**shows** tk\(\neq\)c
**using** REL **unfolding** is-REL-state-def
**apply** clarify
**subgoal for** t' tk'
  **using** b-has-ticket-unique[OF R t<N, of tk t']
  **using** b-holds-only-current[OF R, of t' tk']
  **by** (auto)
  **done**

**lemma** is-REL-state-aux2:
3.2. SOLUTION

assumes R: B. reachable (c,n,ts)
assumes A: t< N ts t = REL tk
shows ~is-REL-state (ts(t:=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: Creachable 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: Creachable (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< N, of tk] that by auto
have REL-AUX: tk=c if ts t = REL tk t< N for tk
  using b-holds-only-current[OF Rc t< N, of tk] that by auto
have [simp]: c< N using B.invar-reachable[OF is-invar4 Rc]
  by (auto simp: invar4-def)
have [simp]:
  replicate N False ≠ (replicate N False)[c := True]
  (replicate N False)[c := True] ≠ replicate N False
apply (auto simp: list-eq-iff-nth-eq nth-list-update)
using c < N by blast+

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

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

from SIM CS have ∃bs'. blstep t bs bs' ∧ 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 exl conjI)
  apply (rule blstep.intros)
  apply (simp)
  apply clarsimp
  apply (rule blstep-sng.enter-wait)
  apply (auto simp: sim-rel2-def split: if-splits)
  done

subgoal for tk'
  apply (intro exl conjI)
  apply (rule blstep.intros)
  apply (simp)
  apply clarsimp
  apply (rule blstep-sng.loop-wait)
subgoal
  apply (clarsimp simp: sim-rel2-def split: if-splits)
  apply (frule (2) is-REL-state-aux1[OF Rc])
  by simp
subgoal by (auto simp: sim-rel2-def split: if-splits)
  done

subgoal
  apply (intro exl conjI)
  apply (rule blstep.intros)
  apply (simp)
  apply (clarsimp split: if-splits)
  apply (rule blstep-sng.exit-wait)
  apply (auto simp: sim-rel2-def split: if-splits)
  done

subgoal
3.2. SOLUTION

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.

definition sim-rel \equiv\ sim-rel1 \ OO \ sim-rel2
interpretation sim: simulation a0 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) \ \longleftrightarrow\ is-HOLD \ (tts \ s \ t)
using assms unfolding sim-rel1-def sim-rel2-def
by (cases tts cs t) auto

lemma xfer-waits:
assumes sim-rel s cs
shows is-WAIT \ (tts \ cs \ t) \ \longleftrightarrow\ is-WAIT \ (tts \ s \ t)
using assms unfolding sim-rel1-def sim-rel2-def
by (cases tts cs t) auto
CHAPTER 3. ARRAY-BASED QUEUING LOCK

lemma xfer-init:
  assumes sim-rel s cs
  shows \( \text{tts } cs t = \text{INIT} \iff \text{tts } s t = \text{INIT} \)
  using asms 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:

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

\[
C.\text{reachable} \equiv C.\text{step}^{\ast\ast} c_{0}
\]

\[
C.\text{is-lrun } l s \equiv s 0 = c_{0} \land (\forall i. \text{clstep } (l i) (s i) (s (Suc i)))
\]

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

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

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

Safety

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

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

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 final-fair:
  assumes RUN: C.is-run s
  assumes ACQ: \( t < N \) and \( \text{tts } (s \ i) t = \text{INIT} \) and \( \text{is-WAIT } (\text{tts } (s (Suc i)) t) \)
  — Thread \( t \) draws ticket in step \( i \)
  assumes HOLD: \( i < j \) and \( \text{is-HOLD } (\text{tts } (s j) t) \)
3.2. SOLUTION

— Thread \( t \) holds lock in step \( j \)
assumes \( \text{WAIT}: t' \leq N \) and \( \text{is-WAIT} (\text{tts} (s \ i \ t')) \)
— Thread \( t' \) waits for lock at step \( i \)
obtains \( \text{where} i < l \) and \( l < j \) and \( \text{is-HOLD} (\text{tts} (s \ l \ t')) \)
— Then, \( t' \) gets lock earlier
using \( \text{RUN} \)
proof (rule sim.xfer-run)
fix as
assume \( R_a: A.\text{is-run as} \) and \( \text{SIM}[(\text{rule-format}]: \forall i. \text{sim-rel (as i) (s i)} \)

note \( XFER = \text{xfer-init[OF SIM]} \text{xfer-holds[OF SIM]} \text{xfer-waits[OF SIM]} \)

show \(?\text{thesis}\)
using \( \text{assms} \)
apply (simp add: \( XFER \))
apply (erule (6) \( \text{fair[OF Ra]} \))
apply (erule (1) \( \text{that} \))
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 \( \text{FRUN}: C.\text{is-fair-run s} \)
assumes \( \text{WAIT}: t < N \) and \( \text{is-WAIT} (\text{tts} (s \ i \ t)) \)
shows \( \exists j > i. \text{is-HOLD} (\text{tts} (s \ j \ t)) \)
using \( \text{FRUN} \)
proof (rule sim.xfer-fair-run)
fix as
assume \( R_a: A.\text{is-fair-run as} \)
and \( \text{SIM}[(\text{rule-format}]: \forall i. \text{sim-rel (as i) (s i)} \)

note \( XFER = \text{xfer-init[OF SIM]} \text{xfer-holds[OF SIM]} \text{xfer-waits[OF SIM]} \)

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

end