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

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 .......... 10
   1.2.5 Simple Client .................................. 11
1.3 Shorter Solution ........................................ 12
   1.3.1 Abstract Specification ............................. 12
   1.3.2 Refinement 1: List with Gap ..................... 12
   1.3.3 Implementation on List-Level .................... 12
   1.3.4 Imperative Arrays ............................... 14
   1.3.5 Executable Code ................................ 15

## 2 Colored Tiles ............................................ 17

2.1 Challenge .................................................. 17
2.2 Solution .................................................. 18
   2.2.1 Problem Specification ............................... 18
   2.2.2 Derivation of Recursion Equations ............... 18
   2.2.3 Verification of Program ............................ 20
   2.2.4 Refinement to Imperative Code ................... 21
   2.2.5 Alternative Problem Specification ............... 22

## 3 Array-Based Queuing Lock ................................ 25

3.1 Challenge .................................................. 25
3.2 Solution .................................................. 27
   3.2.1 General Definitions ............................... 27
   3.2.2 Refinement 1: Ticket Lock with Unbounded Counters 28
   3.2.3 Refinement 2: Bounding the Counters ............ 31
   3.2.4 Refinement 3: Using an Array ..................... 35
   3.2.5 Transfer Setup .................................. 36
   3.2.6 Main Theorems .................................. 37
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 `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.

```
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
```

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

**Extended verification task.** Implement the operation \( \text{grow()} \), specify its behavior in a way that lets you verify \( \text{insert()} \) in a modular way (i.e., not by referring to the implementation of \( \text{grow()} \)), and verify that \( \text{grow()} \) 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.

```
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
⟨proof⟩
lemma empty-text[simp]: get-text empty = []
⟨proof⟩
lemma move-left-pos[simp]: get-pos (move-left b) = get-pos b – 1
— Note that 0 – 1 = 0 in Isabelle
⟨proof⟩
lemma move-left-text[simp]: get-text (move-left b) = get-text b
⟨proof⟩
lemma move-right-pos[simp]:
get-pos (move-right b) = min (get-pos b +1) (length (get-text b))

(\textit{proof})

\textbf{lemma move-right-text}[simp]: get-text (move-right b) = get-text b

(\textit{proof})

\textbf{lemma insert-pos}[simp]: get-pos (insert x b) = get-pos b +1

(\textit{proof})

\textbf{lemma insert-text}: get-text (insert x b)

= take (get-pos b) (get-text b)@x@drop (get-pos b) (get-text b)

(\textit{proof})

\textbf{lemma delete-pos}[simp]: get-pos (delete b) = get-pos b -1

(\textit{proof})

\textbf{lemma delete-text}: get-text (delete b)

= take (get-pos b -1) (get-text b)@drop (get-pos b) (get-text b)

(\textit{proof})

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

\textbf{lemma delete-text0}[simp]: get-pos b=0 =⇒ get-text (delete b) = get-text b

(\textit{proof})

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 [to-relAPP]:

textbuffer-rel A ≡ \{ BUF a b, BUF a’ b’ | a b a’ b’. \}
(a,a’)(\in\langle A\rangle list-rel) ∧ (b,b’)(\in\langle A\rangle list-rel)\}

\textbf{lemma [param]}: \langle BUF, BUF \rangle \in \langle A\rangle list-rel =⇒ \langle A\rangle list-rel =⇒ \langle A\rangle textbuffer-rel

(\textit{proof})

\textbf{lemma [param]}: \langle rec-textbuffer, rec-textbuffer \rangle

\in \langle A\rangle list-rel =⇒ \langle A\rangle list-rel =⇒ B =⇒ \langle A\rangle textbuffer-rel =⇒ B

(\textit{proof})

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

\textbf{sepref-decl-op} (no-def) empty :: \langle A\rangle textbuffer-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) get-text :: \langle A\rangle textbuffer-rel =⇒ \langle A\rangle list-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) get-pos :: \langle A\rangle textbuffer-rel =⇒ nat-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) move-left :: \langle A\rangle textbuffer-rel =⇒ \langle A\rangle textbuffer-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) move-right :: \langle A\rangle textbuffer-rel =⇒ \langle A\rangle textbuffer-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) insert :: A =⇒ \langle A\rangle textbuffer-rel =⇒ \langle A\rangle textbuffer-rel \langle proof \rangle

\textbf{sepref-decl-op} (no-def) delete :: \langle A\rangle textbuffer-rel =⇒ \langle A\rangle textbuffer-rel \langle proof \rangle

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

Empty

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

Left

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

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

Right

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

lemma move-right1-correct: (move-right1, \text{RETURN o move-right}) \in \langle \text{gap-rel} \rangle \text{nres-rel}
(proof)

Insert and Grow

definition can-insert \(\equiv \lambda (l,r,buf). l<r\)
**Definition** 

\[
grow K \equiv \lambda (l, r, buf). \text{doN} \{
\]

\[
\text{let } b = \text{op-array-replicate} \ (\text{length} \ \text{buf} + K) \ \text{default}; \]

\[
b \leftarrow \text{mop-list-blit} \ \text{buf} \ 0 \ b \ 0 \ l; \]

\[
b \leftarrow \text{mop-list-blit} \ \text{buf} \ r \ b \ (r + K) \ (\text{length} \ \text{buf} - r); \]

\[
\text{RETURN} \ (l, r + K, b) \}
\]

**Lemma** 

\[
grow-correct \quad \text{THEN SPEC-trans, refine-vcg}:
\]

**Assumptions**

\[
\text{gap-invar} \ gb
\]

**Shows**

\[
\text{grow} K \ gb \leq (\text{SPEC} \ (\lambda \ gb'.
\]

\[
\text{gap-invar} \ gb'
\]

\[
\land \ \text{gap-\alpha} \ gb' = \text{gap-\alpha} \ gb
\]

\[
\land (K > 0 \rightarrow \text{can-insert} \ gb'))\)
\]

**Proof**

\[
\]

**Definition** 

\[
\text{insert} x \equiv \lambda (l, r, buf). \text{doN} \{
\]

\[
(l, r, buf) \leftarrow
\]

\[
\text{if } (l = r) \text{ then grow1} \ (\text{length} \ \text{buf} + 1) \ (l, r, buf) \text{ else RETURN} \ (l, r, buf);
\]

\[
\text{ASSERT} \ (l < \text{length} \ \text{buf});
\]

\[
\text{RETURN} \ (l + 1, r, \text{buf}[l := x])
\]

**Lemma** 

\[
\text{insert1-correct}:
\]

\[
\text{insert1, RETURN oo insert} \in \text{Id} \to \text{gap-rel} \to \langle \text{gap-rel} \rangle \text{nres-rel}
\]

**Delete**

**Definition** 

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

**Lemma** 

\[
\text{delete1-correct}:
\]

\[
\text{delete1, RETURN o delete} \in \text{gap-rel} \to \langle \text{gap-rel} \rangle \text{nres-rel}
\]

**1.2.4 Imperative Arrays and Executable Code**

**Abbreviation** 

\[
\text{gap-impl-assn} \equiv \text{nat-assn} \times_a \text{nat-assn} \times_a \text{array-assn} \ \text{id-assn}
\]

**Definition** 

\[
\text{gap-assn} \ A \equiv \text{hr-comp} \ (\text{hr-comp} \ \text{gap-impl-assn} \ \text{gap-rel}) \ (\text{the-pure} \ A) \text{textbuffer-rel}
\]

**Context**

**Notes**

\[
\text{gap-assn-def [symmetric, fcomp-norm-unfold]}
\]

**Begin**

**Sepref-definition** 

\[
\text{empty-impl}
\]

**Is**

\[
\text{uncurry0 empty1 :: unit-assn} \to_a \text{gap-impl-assn}
\]

**Proof**

**Sepref-decl-impl** 

\[
\text{empty-impl}: \text{empty-impl} \ \text{refine} \text{FCOMP empty1-correct} \ (\text{proof})
\]


sepref-definition move-left-impl
  is move-left1 :: gap-impl-assn → gap-impl-assn
  ⟨proof⟩

sepref-decl-impl move-left-impl. move-left-impl.refine[FCOMP move-left1-correct] ⟨proof⟩

sepref-definition move-right-impl
  is move-right1 :: gap-impl-assn → gap-impl-assn
  ⟨proof⟩

sepref-decl-impl move-right-impl. move-right-impl.refine[FCOMP move-right1-correct] ⟨proof⟩

sepref-definition insert-impl
  is uncurry insert1 :: id-assn → gap-impl-assn
  ⟨proof⟩

sepref-decl-impl insert-impl. insert-impl.refine[FCOMP insert1-correct] ⟨proof⟩

sepref-definition delete-impl
  is delete1 :: gap-impl-assn → gap-impl-assn
  ⟨proof⟩

sepref-decl-impl delete-impl. delete-impl.refine[FCOMP delete1-correct] ⟨proof⟩

end

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

\[(\text{uncurry0 Challenge1.empty-impl, uncurry0 (RETURN Challenge1.empty)}) \in \text{unit-assn} \Rightarrow \text{a gap-assn} ?A\]

\[(\text{move-left-impl, RETURN } \circ \text{ move-left}) \in (\text{gap-assn} ?A)^d \Rightarrow \text{a gap-assn} ?A\]

\[(\text{move-right-impl, RETURN } \circ \text{ move-right}) \in (\text{gap-assn} ?A)^d \Rightarrow \text{a gap-assn} ?A\]

\[\text{CONSTRAINT is-pure} ?A \Rightarrow (\text{uncurry Challenge1.insert-impl, uncurry (RETURN } \circ \text{ Challenge1.insert)}) \in (\text{gate-assn} ?A)^k \Rightarrow \text{a gap-assn} ?A\]

\[(\text{delete-impl, RETURN } \circ \text{ delete}) \in (\text{gate-assn} ?A)^d \Rightarrow \text{a gap-assn} ?A\]

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

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)

lemma client ≤ SPEC (λr. get-text r=\{1,2,3,4,5\})
(proof)

sepref-definition client-impl
is uncurry0 client :: unit-assn \rightarrow gap-assn id-assn
(proof)

⟨ML⟩
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
1.3. SHORTER SOLUTION

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.

\[
gap-\alpha \equiv \lambda (l, r, \text{buf}). \text{BUF } l \text{@ take } l \text{ buf } @ \text{ drop } r \text{ buf}
\]

\[
gap-invar \equiv \lambda (l, r, \text{buf}). \text{ } l \leq r \land r \leq \text{length buf}
\]

\[
gap-rel \equiv \text{br gap-} \alpha \text{ gap-invar}
\]

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.

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

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

Right

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

\[
\text{lemma move-right1-correct: } \\
\text{(move-right1,RETURN o move-right) } \in \text{gap-rel } \rightarrow \text{(gap-rel)nres-rel }
\]

Insert and Grow

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

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

Proof:

1. Use the definition of can-insert to show that \( l < r \).
2. Use the definition of grow1 to replicate the buffer and then blit it.
3. Verify that the new buffer is correctly aligned and shifted.
4. Prove the invariant gap-invar for the new buffer.
5. Establish the relation gap-rel between the old and new buffers.
6. Use the nres-rel relation to show that the new buffer is correct.

Conclusion:

The shorter solution for the gap-buffer problem is validated through the definitions and proofs provided.
— 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[l 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'))))
⟨proof⟩

definition insert1 x ≡ λ(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<length buf);
RETURN (l+1,r,buf[l:=x])
}

lemma insert1-correct:
(insert1,RETURN oo insert) ∈ Id → gap-rel → (gap-rel)nres-rel
⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

sepref-definition move-right-impl

is move-right1 :: gap-impl-assn → a gap-impl-assn
⟨proof⟩
1.3. SHORTER SOLUTION

sepref-definition insert-impl
is uncurry insert1 :: id-assn \times_a gap-impl-assn \rightarrow_a gap-impl-assn
(proof)

sepref-definition delete-impl
is delete1 :: gap-impl-assn \rightarrow_a gap-impl-assn
(proof)

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 \circ move-left) ∈ gap-assn \rightarrow_a gap-assn

(move-right-impl, RETURN \circ move-right) ∈ gap-assn \rightarrow_a gap-assn

(uncurry Challenge1-short.insert-impl,
uncurry (RETURN \circ Challenge1-short.insert))
∈ id-assn \times_a gap-assn \rightarrow_a gap-assn

(delete-impl, RETURN \circ delete) ∈ gap-assn \rightarrow_a gap-assn

end

1.3.5 Executable Code

Isabelle/HOL can generate code in various target languages.

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:

```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
  (proof)

lemmas valid-red = valid.intros(3)[OF valid.intros(1), simplified]
2.2. SOLUTION

\textbf{lemma valid'-valid:}
\[
\text{valid'} l \implies \text{valid} l
\]
\textbf{⟨proof⟩}

\textbf{lemma valid-eq-valid':}
\[
\text{valid'} l = \text{valid} l
\]
\textbf{⟨proof⟩}

\section*{Additional Facts on Replicate}

\textbf{lemma replicate-iff:}
\[
(\forall i < \text{length} l. l!i = R) \iff (\exists n. l = \text{replicate} n R)
\]
\textbf{⟨proof⟩}

\textbf{lemma replicate-iff2:}
\[
(\forall i < n. l!i = R) \iff (\exists l'. l = \text{replicate} n R \at l') \text{ if } n < \text{length} l
\]
\textbf{⟨proof⟩}

\textbf{lemma replicate-Cons-eq:}
\[
\text{replicate} n x = y \# y s \iff (\exists n'. n = \text{Suc} n' \land x = y \land \text{replicate} n' x = y s)
\]
\textbf{⟨proof⟩}

\section*{Main Case Analysis on \texttt{@term valid}}

\textbf{lemma valid-split:}
\[
\text{valid} l \iff
\begin{align*}
  l = [] \lor \\
  (l|0 = B \land \text{valid} (tl l)) \lor \\
  \text{length} l \geq 3 \land (\forall i < \text{length} l. l!i = R) \lor \\
  (\exists j < \text{length} l. j \geq 3 \land (\forall i < j. l!i = R) \land j = B \land \text{valid} (\text{drop} (j + 1) l))
\end{align*}
\]
\textbf{⟨proof⟩}

\section*{Base cases}

\textbf{lemma lc0-aux:}
\[
\{ l. l = [] \land \text{valid} l \} = \{ [] \}
\]
\textbf{⟨proof⟩}

\textbf{lemma lc0: lcount 0 = 1}
\textbf{⟨proof⟩}

\textbf{lemma lc1aux: \{ l. length l=1 \land valid l \} = \{ [B] \}}
\textbf{⟨proof⟩}

\textbf{lemma lc2aux: \{ l. length l=2 \land valid l \} = \{ [B,B] \}}
\textbf{⟨proof⟩}

\textbf{lemma valid-3R: valid [R, R, R]}
2.2.3 Verification of Program

Inner Loop: Summation

definition sum-prog \( \Phi \) \( l \) \( u \) \( f \) ≡
\[ \text{nfoldli} \ [l..<u] (\lambda - \ True) (\lambda i \ s. \ doN \ { \begin{align*} &\text{ASSERT} (\Phi \ i); \\ &\text{RETURN} (s+f \ i) \end{align*} }) 0 \]

lemma \( \text{sum-spec} \) [\( \text{THEN SPEC-trans, refine-vcg} \)]:
\begin{align*}
&\text{assumes} \ l \leq u \\
&\text{assumes} \ \forall i, l \leq i \Longrightarrow i < u \Longrightarrow \Phi \ i \\
&\text{shows} \ \text{sum-prog} \ \Phi \ l \ u \ f \ \leq \ SPEC \ (\lambda r. \ r = (\sum i=l..<u. \ f \ i))
\end{align*}
\( (\text{proof}) \)
2.2. SOLUTION

Main Program

definition icount M ≡ doN {
    ASSERT (M > 2);
    let c = ap-array-replicate (M + 1) 0;
    let c = c[0:=1, 1:=1, 2:=1, 3:=2];
    ASSERT (∀ i < 4. c[i] = lcount i);
    c ← nfoldli [4..<M+1] (λ. True) (λ n c. doN {
        sum ← sum-prog (λ i. n−i−1 < length c) 3 n (λ i. c[(n−i−1)]);
        ASSERT (n−1 < length c ∧ n < length c);
        RETURN (c[n := c!(n−1) + 1 + sum])
    }) c;
    ASSERT (∀ i ≤ M. c[i] = lcount i);
    RETURN (c!M)
}

Abstract Correctness Statement

theorem icount-correct: M > 2 ⇒ icount M ≤ SPEC (λ r. r = lcount M)
⟨proof⟩

2.2.4 Refinement to Imperative Code

sepref-definition icount-impl is icount :: nat-assn[^] →_a nat-assn
⟨proof⟩

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).

theorem icount-impl-correct:
M > 2 ⇒ <emp > icount-impl M <λ r. ↑(r = lcount M)>,
⟨proof⟩

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

context fixes l :: color list begin

inductive valid-point where

\[ \begin{align*}
\{ i+2 &< \text{length } l; \; l!i = R; \; l!(i+1) = R; \; l!(i+2) = R \} \implies \text{valid-point } i \\
\{ 1 &\leq i + 1 < \text{length } l; \; l!(i-1) = R; \; l!(i) = R; \; l!(i+1) = R \} \implies \text{valid-point } i \\
\{ 2 &\leq i; \; i < \text{length } l; \; l!(i-2) = R; \; l!(i-1) = R; \; l!(i) = R \} \implies \text{valid-point } i \\
\{ i &< \text{length } l; \; l![i = B] \implies \text{valid-point } i
\end{align*} \]

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

lemma valid-line1:
assumes \( \land i. \; i < \text{length } l \implies \text{valid-point } l \; i \)
shows valid-line l
(\( \langle \text{proof} \rangle \))

lemma valid-B-first:
valid-point xs i \( \implies i < \text{length } xs \implies \text{valid-point } (B \# xs) (i + 1) \)
(\( \langle \text{proof} \rangle \))

lemma valid-line-prepend-B:
valid-line (B \# xs) if valid-line xs
(\( \langle \text{proof} \rangle \))

lemma valid-drop-B:
valid-point xs (i - 1) if valid-point (B \# xs) i i > 0
(\( \langle \text{proof} \rangle \))

lemma valid-line-drop-B:
valid-line xs if valid-line (B \# xs)
(\( \langle \text{proof} \rangle \))

lemma valid-line-prepend-B-iff:
valid-line (B \# xs) \longleftrightarrow valid-line xs
(\( \langle \text{proof} \rangle \))

lemma cases-valid-line:
assumes 
\( 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 } (\text{drop } (j + 1) \; l)) \\
(\text{is } ?a \lor ?b \lor ?c \lor ?d)
shows valid-line l
(\( \langle \text{proof} \rangle \))
2.2. SOLUTION

**lemma** valid-line-cases:
\[
\begin{align*}
l &= \emptyset \lor \\
(l!0 &= B \land \text{valid-line } (tl \ l)) \lor \\
\text{length } l &\geq 3 \land (\forall i < \text{length } l, l!i = R) \lor \\
\exists j < \text{length } l, j &\geq 3 \land (\forall i < j, l!i = R) \land l!j = B \land \text{valid-line } (\text{drop } (j + 1) l))
\end{align*}
\]
\begin{proof}
\end{proof}

**lemma** valid-line-split:
\[
\begin{align*}
l &= \emptyset \lor \\
(l!0 &= B \land \text{valid-line } (tl \ l)) \lor \\
\text{length } l &\geq 3 \land (\forall i < \text{length } l, l!i = R) \lor \\
\exists j < \text{length } l, j &\geq 3 \land (\forall i < j, l!i = R) \land l!j = B \land \text{valid-line } (\text{drop } (j + 1) l))
\end{align*}
\]
\begin{proof}
\end{proof}

Connection to the easier definition given above

**lemma** valid-valid-line:
\[
\begin{align*}
\text{valid } l &\leftrightarrow \text{valid-line } l
\end{align*}
\]
\begin{proof}
\end{proof}

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 $\text{pass}[\cdot]$ 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 $\text{pass}[\cdot]$ are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of $\text{pass}[\cdot]$ 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 $N - 1$
    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 $\text{abql_release(my_ticket)}$. 25
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/VTCmp 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 \( \text{mod } (k\times N) \) 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

```plaintext
consts N :: nat

specification (N) N-not0[simp, intro!]: N\neq0 (proof)
lemma N-gt0[simp, intro!]: 0<N (proof)
```

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

```plaintext
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**
```
astate = nat × nat × (nat ⇒ thread)
```

**Abbreviation**
```
cc ≡ fst
```
```
ccs ≡ fst (snd s)
```
```
tss ≡ snd (snd s)
```

The step relation of a single thread

**Inductive**
```
astep-sng where
| enter-wait: astep-sng (c,n,INIT) (c,(n+1),WAIT n)
| loop-wait: c ≠ k ⇒ astep-sng (c,n,WAIT k) (c,n,WAIT k)
| exit-wait: astep-sng (c,n,WAIT c) (c,n,HOLD c)
| start-release: astep-sng (c,n,HOLD k) (c,n,REL k)
| release: astep-sng (c,n,REL k) (k+1,n,INIT)
```

The step relation of the system

**Inductive**
```
alstep t where
[ t < N ; astep-sng (c,n,ts t) (c′,n′,s′) ]
⇒ alstep t (c,n,ts (t:=s′))
```

Initial state of the system

**Definition**
```
as0 ≡ (0, 0, λ-. INIT)
```

**Interpretation**
```
A: system as0 alstep ⟨proof⟩
```

In our system, each thread can always perform a step

**Lemma**
```
never-blocked: A.can-step l s ⟷ l < N
⟨proof⟩
```

Thus, our system is in particular deadlock free

**Interpretation**
```
A: df-system as0 alstep ⟨proof⟩
```

### Safety: Mutual Exclusion

Predicates to express that a thread uses or holds a ticket

**Definition**
```
has-ticket s k ≡ s=WAIT k ∨ s=HOLD k ∨ s=REL k
```

**Lemma**
```
has-ticket-simps[simp]:
¬ has-ticket INIT k
has-ticket (WAIT k) k′⟼ k′=k
has-ticket (HOLD k) k′⟼ k′=k
has-ticket (REL k) k′⟼ k′=k
⟨proof⟩
```

**Definition**
```
locks-ticket s k ≡ s=HOLD k ∨ s=REL k
```
3.2. SOLUTION

lemma locks-ticket-simps[simp];
¬locks-ticket INIT k
¬locks-ticket (WAIT k) k'
locks-ticket (HOLD k) k'\rightarrow k'=k
locks-ticket (REL k) k'\rightarrow k'=k
(proof)

lemma holds-imp-uses: locks-ticket s k \implies has-ticket s k
(proof)

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).
\quad c \leq n
\land (\forall t k. t<N \land has-ticket (ts t) k \rightarrow
\quad 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
(proof)

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

theorem mutual-exclusion: [A.reachable s;
\quad t<N; t'<N; t\neq t'; is-HOLD (ts s t); is-HOLD (ts s t')
\] \implies False
(proof)

lemma mutual-exclusion': [A.reachable s;
\quad t<N; t'<N; t\neq t';
locks-ticket (ts s t) tk; locks-ticket (ts s t') tk'
\] \implies False
(proof)

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.
lemmas 3.1

Lemma 3.1

assumes \( R: A.\text{is-run}\ s \)

assumes \( A: i<j \text{ cc } (s\ i) \leq k \ k < \text{ cc } (s\ j) \ t < N \ t t s\ (s\ i) \ t = \text{WAIT } k \)

shows \( \exists l. i \leq l \wedge l < j \wedge t t s\ (s\ l) \ t = \text{HOLD } k \)

proof

Lemma 3.2

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

proof

A version of the fairness lemma which is very detailed on the actual ticket numbers.

We will weaken this later.

Lemma 3.3

assumes \( RUN: A.\text{is-run}\ s \)

assumes \( ACQ: t < N \ t t s\ (s\ i) \ t = \text{INIT } t t s\ (s\ (\text{Suc } i)) \ t = \text{WAIT } k \)

assumes \( HOLD: i < j \text{ tt s}\ (s\ j) \ t = \text{HOLD } k \)

assumes \( WAIT: t' < N \ t t s\ (s\ i) \ t' = \text{WAIT } k' \)

obtains \( l \) where \( i < l < j \text{ tt s}\ (s\ l) \ t' = \text{HOLD } k' \)

proof

Lemma 3.4

assumes \( RUN: A.\text{is-run}\ s \)

assumes \( WAIT: t < N \ t t s\ (s\ i) \ t = \text{WAIT } tk \)

assumes \( N W A I T: i < j \text{ tt s}\ (s\ j) \ t \neq \text{WAIT } tk \)

obtains \( l \) where \( i < l \leq j \text{ tt s}\ (s\ l) \ t = \text{HOLD } tk \)

proof

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 3.5

assumes \( RUN: A.\text{is-run}\ s \)

assumes \( ACQ: t < N \ t t s\ (s\ i) \ t = \text{INIT } t t s\ (s\ (\text{Suc } i)) \ t \)

— Thread \( t \) calls \( \text{acquire} \) in step \( i \)

assumes \( HOLD: i < j \text{ is-HOLD } t t s\ (s\ j) \ t \)

— Thread \( t \) holds lock in step \( j \)

assumes \( WAIT: t' < N \text{ is-WAIT } t t s\ (s\ i) \ t' \)

— Thread \( t' \) waits for lock at step \( i \)

obtains \( l \) where \( i < l \leq j \text{ is-HOLD } t t s\ (s\ l) \ t' \)

— Then, \( t' \) gets lock earlier

proof

Liveness

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

Definition 3.6

\[ \lambda (c, n, ts). \forall k. c \leq k \wedge k < n \rightarrow (\exists t < N. \text{has-ticket } (ts\ t)\ k) \]
3.2. SOLUTION

**Lemma** is-invar2: $A.is-invar \text{ invar2}$

\[
\text{(proof)}
\]

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

**Corollary** current-lock-used:

\[
\text{assumes } R: A.\text{reachable (c,n,ts)} \\
\text{assumes } \text{WAIT: } t < N \text{ ts } t = \text{WAIT } k \\
\text{obtains } t' \text{ where } t' < N \text{ has-ticket (ts } t') c \\
\text{(proof)}
\]

Used tickets are unique (Corollary from invariant 1)

**Lemma** has-ticket-unique:

\[
\begin{align*}
&\text{assumes } R: A.\text{reachable (c,n,ts)}; \\
&\text{assumes } \text{A: } t < N \text{ has-ticket (ts } t) k; \\
&\text{obtains } t' < N \text{ has-ticket (ts } t') k \\
\end{align*}
\]

\[
\Rightarrow t' = t \\
\text{(proof)}
\]

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:

\[
\text{assumes } R: A.\text{reachable (c,n,ts)} \\
\text{assumes } A: t < N \text{ has-ticket (ts } t) k \\
\text{shows } \text{tkt-thread ts } k = t \\
\text{(proof)}
\]

**Lemma** holds-only-current:

\[
\text{assumes } R: A.\text{reachable (c,n,ts)} \\
\text{assumes } A: t < N \text{ locks-ticket (ts } t) k \\
\text{shows } k = c \\
\text{(proof)}
\]

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

**Definition** tweight $s \equiv$

\[
\begin{align*}
&\text{case } s \text{ of WAIT } - \Rightarrow 3: \text{nat} \mid \text{HOLD } - \Rightarrow 2 \mid \text{REL } - \Rightarrow 1 \mid \text{INIT } \Rightarrow 0 \\
\end{align*}
\]

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

**Theorem** progress:

\[
\text{assumes } \text{FRUN: } A.\text{is-fair-run } s \\
\text{assumes } A: t < N \text{ is-WAIT (tts } (s i) t) \\
\text{shows } \exists j > i. \text{is-HOLD (tts } (s j) t) \\
\text{(proof)}
\]

3.2.3 Refinement 2: Bounding the Counters

We fix the $k$ from the task description, which must be positive

**Consts** $k::\text{nat}$
CHAPTER 3. ARRAY-BASED QUEUING LOCK

specification \( (k) \) \( k\neq0 \) (proof)

lemma \( kg0 \) (simp); \( 0<k \) (proof)

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

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

The step relation of a single thread

inductive bstep-sng where

\[
\begin{align*}
\text{enter-wait: } & \hspace{1em} \text{bstep-sng (c,n,\text{INIT}) (c,(n+1) \mod (k*N),\text{WAIT (n mod N)}) } \\
\text{loop-wait: } & \hspace{1em} c\neq tk \Rightarrow \text{bstep-sng (c,n,\text{WAIT tk}) (c,n,\text{WAIT tk}) } \\
\text{exit-wait: } & \hspace{1em} \text{bstep-sng (c,n,\text{WAIT c}) (c,n,\text{HOLD c}) } \\
\text{start-release: } & \hspace{1em} \text{bstep-sng (c,n,\text{HOLD tk}) (c,n,\text{REL tk}) } \\
\text{release: } & \hspace{1em} \text{bstep-sng (c,n,\text{REL tk}) ((tk+1) \mod N,n,\text{INIT}) }
\end{align*}
\]

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

inductive blstep for \( t \) where

\[
\begin{align*}
\text{[ } t<N; \text{ bstep-sng (c,n,ts t) (c’,n’,s’) ] } \\
\Rightarrow & \hspace{1em} \text{blstep t (c,n,ts) (c’,n’,ts(t:=s’))}
\end{align*}
\]

Initial state of the system

definition \( bs_0 \equiv (0, 0, \lambda - \text{INIT}) \)

interpretation B: system \( bs_0 \) blstep (proof)

lemma b-never-blocked: \( B.\text{can-step l s} \iff l < N \) (proof)

interpretation B: df-system \( bs_0 \) blstep (proof)

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]:
\[
\text{finite s} \Rightarrow \text{sum f s \neq (0::nat) \iff (\exists x \in s. f x \neq 0)}
\]
(proof)

lemma n-eq-Suc-sub1-conv [simp]: \( n = \text{Suc (n - Suc 0)} \iff n \neq 0 \) (proof)

lemma mod-mult-mod-eq [mod-simps]: \( \text{x mod (k * N) mod N = x mod N} \) (proof)

lemma mod-eq-imp-eq-aux: \( b \mod N = (a::\text{nat}) \mod N \Rightarrow a \leq b \Rightarrow b < a+N \Rightarrow b = a \) (proof)

lemma 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)

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 \iff \( t \) = INIT
map-ticket \( f \) \( t \) = WAIT \( tk \) \iff (∃tk'. \( tk = f \) tk' ∧ \( t \) = WAIT \( tk' \))
map-ticket \( f \) \( t \) = HOLD \( tk \) \iff (∃tk'. \( tk = f \) tk' ∧ \( t \) = HOLD \( tk' \))
map-ticket \( f \) \( t \) = REL \( tk \) \iff (∃tk'. \( tk = f \) tk' ∧ \( t \) = REL \( tk' \))

(proof)

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

(proof)

**definition** num-ni \( ts \) ≡ \( \sum \) i = 0..<N. ni-weight \( ts \) \( i \)

**lemma** num-ni-init [simp]: num-ni (\( λ \cdot \) INIT) = 0 (proof)

**lemma** num-ni-upd:

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

(proof)

**lemma** num-ni-nz-if [simp]: \( [t < N; \; ts \; t \neq INIT] \implies \) num-ni \( ts \) ≠ 0

(proof)

**lemma** num-ni-leN: num-ni \( ts \) ≤ N

(proof)

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

(proof)

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

**definition** sim-rel1 ≡ \( λ(c,n,ts). \) (\( ci,ni,tsi \)).
\( ci = c \mod N \)
\( \land ni = n \mod (k*N) \)
\( \land tsi = (\text{map-ticket } (\lambda t. t \mod N)) \circ ts \)

**Lemma sraux:**

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

**Lemma sraux2:**

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

**Interpretation sim1:**

\( \text{simulationI as0 alstep bs0 blstep sim-rel1} \)

**Transfer of Properties**

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

**Lemma xfer-locks-ticket:**

\( \text{assumes locks-ticket } (\text{map-ticket } (\lambda t. t \mod N)) \ (ts \ t) \ tki \)
\( \text{obtains } tk \text{ where } tki = tk \mod N \text{ locks-ticket } (ts \ t) \ tk \)

**Lemma b-holds-only-current:**

\( [B.\text{reachable } (c, n, ts); t < N; \text{locks-ticket } (ts \ t) \ tk] \implies tk = c \)

**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' \]
\( \implies False \)

**Lemma xfer-has-ticket:**

\( \text{assumes has-ticket } (\text{map-ticket } (\lambda t. t \mod N)) \ (ts \ t) \ tki \)
\( \text{obtains } tk \text{ where } tki = tk \mod N \text{ has-ticket } (ts \ t) \ tk \)

**Lemma has-ticket-in-range:**

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

**Lemma b-has-ticket-unique:**

\( B.\text{reachable } (ci, ni, tsi); t < N; \text{has-ticket } (tsi \ t) \ tki; t' < N; \text{has-ticket } (tsi \ t') \ tk' \]
\( \implies t' = t \)
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.

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

```
type-synonym cstate = bool list \times nat \times (nat \Rightarrow thread)
```

The step relation of a single thread

```
inductive cstep-sng where
  enter-wait: cstep-sng (p.n.INIT) (p.(n+1) mod (k\times N).WAIT (n mod N))
  | loop-wait: p!tk \Rightarrow cstep-sng (p.n.WAIT tk) (p.n.WAIT tk)
  | exit-wait: p!tk \Rightarrow 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+t) 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
  \[ t<N; cstep-sng (c.n.ts t) (c'.n'.s') \]
  \Rightarrow clstep t (c.n.ts) (c'.n'.ts(t:=s'))
```

Initial state of the system

```
definition cs0 \equiv (\text{replicate } N \text{ False})[0:=True], 0, \lambda -. INIT)
```

```
interpretation C: system cs0 clstep ⟨proof⟩
```

```
lemma c-never-blocked: C.can-step I s \iff I<N
⟨proof⟩
```

```
interpretation C: df-system cs0 clstep ⟨proof⟩
```

We establish another invariant that states that the ticket numbers are bounded.

```
definition invar4
  \equiv \lambda(c.n.ts). c<N \land (\forall t<N. \forall tk. \text{has-ticket (ts t) tk} \longrightarrow tk<N)
```

```
lemma is-invar4: B.is-invar invar4
⟨proof⟩
```

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(\( t := \text{REL} \ tk \)))} \]
\[ t < N \Rightarrow \neg \text{is-REL (ts t)} \Rightarrow \neg \text{is-REL s'} \]
\[ \Rightarrow \text{is-REL-state (ts(\( t := s' \)))} \leftrightarrow \text{is-REL-state ts} \]
(proof)

lemma is-REL-state-aux1:
assumes \( R: B.\text{reachable} \) \((c,n,ts)\)
assumes \( REL: \text{is-REL-state ts} \)
assumes \( t < N \) and \( \text{simp}: ts t = \text{WAIT} \ tk \)
shows \( tk \neq c \)
(proof)

lemma is-REL-state-aux2:
assumes \( R: B.\text{reachable} \) \((c,n,ts)\)
assumes \( A: t < N \) ts t = \( \text{REL} \ tk \)
shows \( \neg \text{is-REL-state (ts(\( t := \text{INIT} \))} \)
(proof)

Simulation relation that implements current ticket by array

\[
\text{definition sim-rel2} \equiv \lambda (c,n,ts) (ci,ni,tsi).
\begin{align*}
\text{if is-REL-state ts then} & \\
\text{ci} &= \text{replicate N False} \\
\text{else} & \\
\text{ci} &= (\text{replicate N False})[c:=True] \\
\end{align*}
\]
\[\wedge ni = n \]
\[\wedge tsi = ts\]

\[
\text{interpretation sim2: simulationI bs_0 blstep cs_0 clstep sim-rel2}
\]
(proof)

3.2.5 Transfer Setup

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

\[
\text{definition sim-rel} \equiv \text{sim-rel1 OO sim-rel2}
\]

\[
\text{interpretation sim: simulation as_0 alstep cs_0 clstep sim-rel}
\]
(proof)

lemma xfer-holds:
assumes sim-rel s cs
shows is-HOLD (tts cs t) \leftrightarrow is-HOLD (tts s t)
(proof)

lemma xfer-waits:
assumes sim-rel \( s \) \( cs \)
shows is-WAIT \((tts \ cs \ t) \leftarrow \) is-WAIT \((tts \ s \ t)\)
(proof)

lemma xfer-init:
assumes sim-rel \( s \) \( cs \)
shows \( tts \ cs \ t = INIT \leftarrow \) tts \( s \) \( t \) = INIT
(proof)

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 fair-
ness here:

\[
C.\text{step } s \ s' = (\exists l. \text{clstep } l \ s \ s')
\]
\[
C.\text{reachable} \equiv C.\text{step}^* \ c_{s_0}
\]
\[
C.\text{is-run } l \ s \equiv s \ 0 = c_{s_0} \land (\forall i. \text{clstep } (l \ i) \ (s \ i) \ (s \ (\text{Suc } i)))
\]
\[
C.\text{is-run } s \equiv \exists l. C.\text{is-lrun } l \ s
\]
\[
C.\text{is-lrun } l \ ss \equiv \forall i. \exists j \geq i. \neg C.\text{can-step } l \ (ss \ j) \lor ls \ j = l
\]
\[
C.\text{is-fair-run } s \equiv \exists l. C.\text{is-lrun } l \ s \land C.\text{is-lfair } l \ s
\]

Safety

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

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

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 \( t \).

\begin{align*}
\text{theorem final-fair: } & \text{assumes RUN: } C.\text{is-run } s \\
& \text{assumes ACQ: } t < N \text{ and } tts (s \ i) t = \text{INIT and is-WAIT } (tts (s (\text{Suc } i)) \ t) \\
& \quad \text{— Thread } t \text{ draws ticket in step } i \\
& \text{assumes HOLD: } i < j \text{ and is-HOLD } (tts (s \ j) \ t)
\end{align*}
— Thread \( t \) holds lock in step \( j \)
\textbf{assumes} \( \text{WAIT: } t' < N \text{ and } \text{is-WAIT} \ (\text{tts} \ (s \ i) \ t') \)
— Thread \( t' \) waits for lock at step \( i \)
\textbf{obtains} \( l \) where \( i < l \text{ and } t < j \text{ and } \text{is-HOLD} \ (\text{tts} \ (s \ l) \ t') \)
— Then, \( t' \) gets lock earlier
\( \langle \text{proof} \rangle \)

\section*{Liveness}

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

\textbf{theorem} final-progress:
\textbf{assumes} \( \text{FRUN: } C.\text{is-fair-run} \ s \)
\textbf{assumes} \( \text{WAIT: } t < N \text{ and } \text{is-WAIT} \ (\text{tts} \ (s \ i) \ t) \)
\textbf{shows} \( \exists j > i. \text{is-HOLD} \ (\text{tts} \ (s \ j) \ t) \)
\( \langle \text{proof} \rangle \)

\textbf{end}