VerifyThis 2018 - Polished Isabelle Solutions

Peter Lammich Simon Wimmer

March 17, 2025

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

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 $a[0],\ldots,a[l-1],a[r],...,a[n-1]$.
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 `insert()` has the precondition $l < r$ and remove the call to `grow()`. Alternatively, assume a contract for `grow()` that ensures that this call does not change the abstract representation.

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

```
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

- https://en.wikipedia.org/wiki/Gap_buffer
- http://scienceblogs.com/goodmath/2009/02/18/gap-buffers-or-why-bother-with-1

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*))
⟨*proof*⟩
lemma *move-right-text*[*simp*]: *get-text* (*move-right b*) = *get-text b*
⟨*proof*⟩

lemma *insert-pos*[*simp*]: *get-pos* (*insert x b*) = *get-pos b* + *1*
⟨*proof*⟩
lemma *insert-text*: *get-text* (*insert x b*)
= *take* (*get-pos b*) (*get-text b*)@*x*#*drop* (*get-pos b*) (*get-text b*)
⟨*proof*⟩

lemma *delete-pos*[*simp*]: *get-pos* (*delete b*) = *get-pos b* − *1*
⟨*proof*⟩
lemma *delete-text*: *get-text* (*delete b*)
= *take* (*get-pos b*−*1*) (*get-text b*)@*drop* (*get-pos b*) (*get-text b*)
⟨*proof*⟩

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*
⟨*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′*)∈⟨*A*⟩*list-rel* ∧ (*b*,*b′*)∈⟨*A*⟩*list-rel*}

lemma [*param*]: (*BUF*,*BUF*) ∈ ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel* → ⟨*A*⟩*textbuffer-rel*
⟨*proof*⟩
lemma [*param*]: (*rec-textbuffer*,*rec-textbuffer*)
∈ (⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel*→*B*) → ⟨*A*⟩*textbuffer-rel* → *B*
⟨*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
sepref-decl-op (*no-def*) *empty* :: ⟨*A*⟩*textbuffer-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *get-text* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*list-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *get-pos* :: ⟨*A*⟩*textbuffer-rel* → *nat-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *move-left* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *move-right* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *insert* :: *A*→⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* ⟨*proof*⟩
sepref-decl-op (*no-def*) *delete* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* ⟨*proof*⟩
end

1.2.2 Refinement 1: List with Gap

1.2.3 Implementation on List-Level

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

Abstraction Relation

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

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

Empty

definition *empty1* ≡ *RETURN* (*0*,*0*,[])
lemma *empty1-correct*: (*empty1*, *RETURN empty*) ∈ ⟨*gap-rel*⟩*nres-rel*
⟨*proof*⟩

Left

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

lemma *move-left1-correct*:
(*move-left1*, *RETURN o move-left*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
⟨*proof*⟩

Right

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

lemma *move-right1-correct*:
(*move-right1*,*RETURN o move-right*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
⟨*proof*⟩

Insert and Grow

definition *can-insert* ≡ λ(*l*,*r*,*buf*). *l*<*r*

definition *grow1 K* ≡ λ(*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*)
}

lemma *grow1-correct*[*THEN SPEC-trans*, *refine-vcg*]:
assumes *gap-invar gb*
shows *grow1 K gb* ≤ (*SPEC* (λ*gb*′.
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*⟩

1.2.4 Imperative Arrays and Executable Code

abbreviation *gap-impl-assn* ≡ *nat-assn* $\times_a$ *nat-assn* $\times_a$ *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}$\rightarrow_a$*gap-impl-assn*
⟨*proof*⟩
sepref-decl-impl *empty-impl*: *empty-impl*.*refine*[*FCOMP empty1-correct*] ⟨*proof*⟩

sepref-definition *move-left-impl*
is *move-left1* :: *gap-impl-assn*^{d}$\rightarrow_a$*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*^{d}$\rightarrow_a$*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*^{k}$*_a$*gap-impl-assn*^{d}$\rightarrow_a$*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*^{d}$\rightarrow_a$*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:

(*uncurry0 Challenge1.empty-impl*, *uncurry0* (*RETURN Challenge1.empty*))
∈ *id-assn*^{k} $\rightarrow_a$ *gap-assn ?A*
(*move-left-impl*, *RETURN* ∘ *move-left*) ∈ (*gap-assn ?A*)^d $\rightarrow_a$ *gap-assn ?A*
(*move-right-impl*, *RETURN* ∘ *move-right*) ∈ (*gap-assn ?A*)^d $\rightarrow_a$ *gap-assn ?A*
CONSTRAINT is-pure ?A ⟹
(*uncurry Challenge1.insert-impl*, *uncurry* (*RETURN* ∘∘ *Challenge1.insert*))
∈ *?A*^{k} $*_a$ (*gap-assn ?A*)^d $\rightarrow_a$ *gap-assn ?A*
(*delete-impl*, *RETURN* ∘ *delete*) ∈ (*gap-assn ?A*)^d $\rightarrow_a$ *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* ($\lambda f. f$) [
insert (*1*::*int*),
insert (*2*::*int*),
insert (*3*::*int*),
insert (*5*::*int*),
move-left,
insert (*4*::*int*),

move-right,
insert (*6*::*int*),
delete
] *empty*)

lemma *client* $\leq$ *SPEC* (λr. *get-text r*=[*1*,*2*,*3*,*4*,*5*])
⟨*proof*⟩

sepref-definition *client-impl*
is *uncurry0 client* :: *unit-assn*^{k} $\rightarrow_a$ *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*

Abstraction Relation

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

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

Left

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

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

lemma *move-left1-correct*:
(*move-left1*, *RETURN o move-left*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
⟨*proof*⟩

Right

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

lemma *move-right1-correct*:
(*move-right1*,*RETURN o move-right*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
⟨*proof*⟩

Insert and Grow

definition *can-insert* ≡ λ(*l*,*r*,*buf*). *l*<*r*

```
definition grow1 K ≡ λ(l,r,buf). doN {
  let b = op-array-replicate (length buf + K) default;
  b ← mop-list-blit buf 0 b 0 l;
  b ← mop-list-blit buf r b (r+K) (length buf − r);
  RETURN (l,r+K,b)
```

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

lemma *grow1-correct*[*THEN SPEC-trans*, *refine-vcg*]:
— Declares this as a rule to be used by the VCG
assumes *gap-invar gb*
shows *grow1 K gb* ≤ (*SPEC* (λ*gb*′.
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* $\times_a$ *nat-assn* $\times_a$ *array-assn id-assn*

sepref-definition *move-left-impl*
is *move-left1* :: *gap-impl-assn*^{d}$\rightarrow_a$*gap-impl-assn*
⟨*proof*⟩

sepref-definition *move-right-impl*
is *move-right1* :: *gap-impl-assn*^{d}$\rightarrow_a$*gap-impl-assn*
⟨*proof*⟩

sepref-definition *insert-impl*
is *uncurry insert1* :: $id\text{-}assn^k *_a gap\text{-}impl\text{-}assn^d \rightarrow_a gap\text{-}impl\text{-}assn$
⟨*proof*⟩

sepref-definition *delete-impl*
is *delete1* :: $gap\text{-}impl\text{-}assn^d \rightarrow_a gap\text{-}impl\text{-}assn$
⟨*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:

$(move\text{-}left\text{-}impl, RETURN \circ move\text{-}left) \in gap\text{-}assn^d \rightarrow_a gap\text{-}assn$

$(move\text{-}right\text{-}impl, RETURN \circ move\text{-}right) \in gap\text{-}assn^d \rightarrow_a gap\text{-}assn$

$(uncurry\ Challenge1\text{-}short.insert\text{-}impl,$
$uncurry\ (RETURN \circ\circ Challenge1\text{-}short.insert))$
$\in id\text{-}assn^k *_a gap\text{-}assn^d \rightarrow_a gap\text{-}assn$

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

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:

```
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
count[3] := 2   // [BBB] or [RRR]
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*) |
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*]

lemma *valid′-valid*:
$valid'\ l \Longrightarrow valid\ l$
⟨*proof*⟩

lemma *valid-eq-valid′*:
$valid'\ l = valid\ l$
⟨*proof*⟩

Additional Facts on Replicate

lemma *replicate-iff*:
$(\forall i<length\ l.\ l\ !\ i = R) \longleftrightarrow (\exists\ n.\ l = replicate\ n\ R)$
⟨*proof*⟩

lemma *replicate-iff2*:
$(\forall i<n.\ l\ !\ i = R) \longleftrightarrow (\exists\ l'.\ l = replicate\ n\ R\ @\ l')$ **if** $n < length\ l$
⟨*proof*⟩

lemma *replicate-Cons-eq*:
$replicate\ n\ x = y\ \#\ ys \longleftrightarrow (\exists\ n'.\ n = Suc\ n' \wedge x = y \wedge replicate\ n'\ x = ys)$
⟨*proof*⟩

Main Case Analysis on @*term valid*

lemma *valid-split*:
$valid\ l \longleftrightarrow$
$l = [] \vee$
$(l!0 = B \wedge valid\ (tl\ l)) \vee$
$length\ l \geq 3 \wedge (\forall\ i < length\ l.\ l\ !\ i = R) \vee$
$(\exists\ j < length\ l.\ j \geq 3 \wedge (\forall\ i < j.\ l\ !\ i = R) \wedge l\ !\ j = B \wedge valid\ (drop\ (j + 1)\ l))$
⟨*proof*⟩

Base cases

lemma *lc0-aux*:
$\{l.\ l = [] \wedge valid\ l\} = \{[]\}$
⟨*proof*⟩

lemma *lc0*: $lcount\ 0 = 1$
⟨*proof*⟩

lemma *lc1aux*: $\{l.\ length\ l{=}1 \wedge valid\ l\} = \{[B]\}$
⟨*proof*⟩

lemma *lc2aux*: $\{l.\ length\ l{=}2 \wedge valid\ l\} = \{[B,B]\}$
⟨*proof*⟩

lemma *valid-3R*: ‹$valid\ [R, R, R]$›
⟨*proof*⟩

lemma *lc3-aux*: $\{l.\ length\ l{=}3 \wedge valid\ l\} = \{[B,B,B], [R,R,R]\}$
⟨*proof*⟩

lemma *lcounts-init*: $lcount\ 0 = 1\ lcount\ 1 = 1\ lcount\ 2 = 1\ lcount\ 3 = 2$
⟨*proof*⟩

The Recursion Case

lemma *finite-valid-length*:
$finite\ \{l.\ length\ l = n \wedge valid\ l\}$ (**is** *finite ?S*)
⟨*proof*⟩

lemma *valid-line-just-B*:
$valid\ (replicate\ n\ B)$
⟨*proof*⟩

lemma *valid-line-aux*:
$\{l.\ length\ l = n \wedge valid\ l\} \neq \{\}$ (**is** $?S \neq \{\}$)
⟨*proof*⟩

lemma *replicate-unequal-aux*:
$replicate\ x\ R\ @\ B\ \#\ l \neq replicate\ y\ R\ @\ B\ \#\ l'$ (**is** $?l \neq ?r$) **if** ‹$x < y$› **for** $l\ l'$
⟨*proof*⟩

lemma *valid-prepend-B-iff*:
$valid\ (B\ \#\ xs) \longleftrightarrow valid\ xs$
⟨*proof*⟩

lemma *lcrec*: $lcount\ n = lcount\ (n{-}1) + 1 + (\sum i{=}3..{<}n.\ lcount\ (n{-}i{-}1))$ **if** ‹$n{>}3$›
⟨*proof*⟩

2.2.3 Verification of Program

Inner Loop: Summation

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

lemma *sum-spec*[*THEN SPEC-trans*, *refine-vcg*]:
assumes $l{\le}u$
assumes $\bigwedge i.\ l{\le}i \Longrightarrow i{<}u \Longrightarrow \Phi\ i$
shows *sum-prog* $\Phi\ l\ u\ f \le SPEC\ (\lambda r.\ r{=}(\sum i{=}l..{<}u.\ f\ i))$
⟨*proof*⟩

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 (∀*i*<*4*. *c*!*i* = *lcount i*);

  c←*nfoldli* [*4*..<*M*+*1*] (λ-. *True*) (λ*n c*. *doN* {
    ~~*let sum* = (∑*i*=*3*..<*n*. *c*!(*n*−*i*−*1*));~~
    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*);

  ASSERT (*M* < *length c*);
  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* :: $nat\text{-}assn^k \rightarrow_a nat\text{-}assn$
  ⟨*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*:
  $M>2 \Longrightarrow$ <*emp*> *icount-impl M* <λ*r*. ↑(*r* = *lcount M*)>$_t$
⟨*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**
$[\![i+2<length\ l;\ l!i=R;\ l!(i+1) = R;\ l!(i+2) = R\ ]\!] \Longrightarrow valid\text{-}point\ i$
$|\ [\![1\leq i;i+1<length\ l;\ l!(i-1)=R;\ l!(i) = R;\ l!(i+1) = R\ ]\!] \Longrightarrow valid\text{-}point\ i$
$|\ [\![2\leq i;\ i<length\ l;\ l!(i-2)=R;\ l!(i-1) = R;\ l!(i) = R\ ]\!] \Longrightarrow valid\text{-}point\ i$
$|\ [\![\ i<length\ l;\ l!i=B]\!] \Longrightarrow valid\text{-}point\ i$

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

lemma *valid-lineI*:
assumes $\bigwedge i.\ i < length\ l \Longrightarrow valid\text{-}point\ l\ i$
shows *valid-line l*
⟨*proof*⟩

lemma *valid-B-first*:
$valid\text{-}point\ xs\ i \Longrightarrow i < length\ xs \Longrightarrow valid\text{-}point\ (B\ \#\ xs)\ (i + 1)$
⟨*proof*⟩

lemma *valid-line-prepend-B*:
valid-line (*B* # *xs*) **if** *valid-line xs*
⟨*proof*⟩

lemma *valid-drop-B*:
valid-point xs $(i - 1)$ **if** *valid-point* (*B* # *xs*) *i* $i > 0$
⟨*proof*⟩

lemma *valid-line-drop-B*:
valid-line xs **if** *valid-line* (*B* # *xs*)
⟨*proof*⟩

lemma *valid-line-prepend-B-iff*:
valid-line (*B* # *xs*) $\longleftrightarrow$ *valid-line xs*
⟨*proof*⟩

lemma *cases-valid-line*:
assumes
$l = [] \vee$
$(l!0 = B \wedge valid\text{-}line\ (tl\ l)) \vee$
$length\ l \geq 3 \wedge (\forall\ i < length\ l.\ l\ !\ i = R) \vee$
$(\exists\ j < length\ l.\ j \geq 3 \wedge (\forall\ i < j.\ l\ !\ i = R) \wedge l\ !\ j = B \wedge valid\text{-}line\ (drop\ (j + 1)\ l))$
(**is** *?a* ∨ *?b* ∨ *?c* ∨ *?d*)
shows *valid-line l*
⟨*proof*⟩

lemma *valid-line-cases*:
$l = [] \lor$
$(l!0 = B \land \textit{valid-line}\ (tl\ l)) \lor$
$\textit{length}\ l \geq 3 \land (\forall\ i < \textit{length}\ l.\ l\ !\ i = R) \lor$
$(\exists\ j < \textit{length}\ l.\ j \geq 3 \land (\forall\ i < j.\ l\ !\ i = R) \land l\ !\ j = B \land \textit{valid-line}\ (\textit{drop}\ (j + 1)\ l))$
if *valid-line l*
⟨*proof*⟩

lemma *valid-line-split*:
$\textit{valid-line}\ l \longleftrightarrow$
$l = [] \lor$
$(l!0 = B \land \textit{valid-line}\ (tl\ l)) \lor$
$\textit{length}\ l \geq 3 \land (\forall\ i < \textit{length}\ l.\ l\ !\ i = R) \lor$
$(\exists\ j < \textit{length}\ l.\ j \geq 3 \land (\forall\ i < j.\ l\ !\ i = R) \land l\ !\ j = B \land \textit{valid-line}\ (\textit{drop}\ (j + 1)\ l))$
⟨*proof*⟩

Connection to the easier definition given above

lemma *valid-valid-line*:
$\textit{valid}\ l \longleftrightarrow \textit{valid-line}\ l$
⟨*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 `pass[]` of length `N`. We also allocate a shared integer value `next`. In practice, `next` is an unsigned bounded integer that wraps to 0 on overflow, and we assume that the maximal value of `next` is of the form $k\texttt{N}-1$. Finally, we assume at our disposal an atomic `fetch_and_add` instruction, such that `fetch_and_add(next,1)` increments the value of `next` by 1 and returns the original value of `next`.
The elements of `pass[]` are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of `pass[]` is set to `false`, except `pass[0]` which is set to `true`, allowing the first coming thread to acquire the lock. Variable `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:

```
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 `abql_release(my_ticket)`,

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 *mod N* and *mod* $(k*N)$ *respectively*

3. We implement the current counter by an array, yielding exactly the algorithm described in the challenge.

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

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

3.2.1 General Definitions

We fix a positive number *N* of threads

consts *N* :: *nat*
specification (*N*) *N-not0*[*simp*, *intro*!]: *N*≠*0* ⟨*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.

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 $astate = nat \times nat \times (nat \Rightarrow thread)$

abbreviation $cc \equiv fst$
abbreviation $nn\ s \equiv fst\ (snd\ s)$
abbreviation $tts\ s \equiv snd\ (snd\ s)$

The step relation of a single thread

inductive *astep-sng* **where**
enter-wait: $astep\text{-}sng\ (c,n,INIT)\ (c,(n+1),WAIT\ n)$
| *loop-wait*: $c \neq k \Longrightarrow astep\text{-}sng\ (c,n,WAIT\ k)\ (c,n,WAIT\ k)$
| *exit-wait*: $astep\text{-}sng\ (c,n,WAIT\ c)\ (c,n,HOLD\ c)$
| *start-release*: $astep\text{-}sng\ (c,n,HOLD\ k)\ (c,n,REL\ k)$
| *release*: $astep\text{-}sng\ (c,n,REL\ k)\ (k+1,n,INIT)$

The step relation of the system

inductive *alstep* **for** *t* **where**
$[\![\ t<N;\ astep\text{-}sng\ (c,n,ts\ t)\ (c',n',s')\ ]\!]$
$\Longrightarrow alstep\ t\ (c,n,ts)\ (c',n',ts(t:=s'))$

Initial state of the system

definition $as_0 \equiv (0,\ 0,\ \lambda\text{-}.\ INIT)$

interpretation *A*: $system\ as_0\ alstep$ ⟨*proof*⟩

In our system, each thread can always perform a step

lemma *never-blocked*: $A.can\text{-}step\ l\ s \longleftrightarrow l<N$
⟨*proof*⟩

Thus, our system is in particular deadlock free

interpretation *A*: $df\text{-}system\ as_0\ alstep$
⟨*proof*⟩

Safety: Mutual Exclusion

Predicates to express that a thread uses or holds a ticket

definition $has\text{-}ticket\ s\ k \equiv s=WAIT\ k \lor s=HOLD\ k \lor s=REL\ k$
lemma *has-ticket-simps*[*simp*]:
$\neg has\text{-}ticket\ INIT\ k$
$has\text{-}ticket\ (WAIT\ k)\ k' \longleftrightarrow k'=k$
$has\text{-}ticket\ (HOLD\ k)\ k' \longleftrightarrow k'=k$
$has\text{-}ticket\ (REL\ k)\ k' \longleftrightarrow k'=k$
⟨*proof*⟩

definition $locks\text{-}ticket\ s\ k \equiv s=HOLD\ k \lor s=REL\ k$

lemma *locks-ticket-simps*[*simp*]:
$\neg$*locks-ticket INIT k*
$\neg$*locks-ticket* (*WAIT k*) *k'*
locks-ticket (*HOLD k*) *k'* $\longleftrightarrow$ *k'*=*k*
locks-ticket (*REL k*) *k'* $\longleftrightarrow$ *k'*=*k*
⟨*proof*⟩

lemma *holds-imp-uses*: *locks-ticket s k* $\Longrightarrow$ *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* $\equiv \lambda(c,n,ts)$.
$c \leq n$
$\wedge$ ($\forall t\ k.\ t<N \wedge$ *has-ticket* (*ts t*) *k* $\longrightarrow$
$c \leq k \wedge k < n$
$\wedge$ ($\forall t'\ k'.\ t'<N \wedge$ *has-ticket* (*ts t'*) $k' \wedge t \neq t' \longrightarrow k \neq k'$)
$\wedge$ ($\forall k.\ k \neq c \longrightarrow \neg$*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*;
$t<N$; $t'<N$; $t \neq t'$; *is-HOLD* (*tts s t*); *is-HOLD* (*tts s t'*)
⟧ $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *mutual-exclusion'*: ⟦*A.reachable s*;
$t<N$; $t'<N$; $t \neq t'$;
locks-ticket (*tts s t*) *tk*; *locks-ticket* (*tts s t'*) *tk'*
⟧ $\Longrightarrow$ *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*.

lemma *fair-aux*:
assumes *R*: *A.is-run s*
assumes *A*: $i{<}j$ $cc\ (s\ i) \le k$ $k < cc\ (s\ j)$ $t{<}N$ $tts\ (s\ i)\ t{=}WAIT\ k$
shows $\exists l.\ i{\le}l \wedge l{<}j \wedge tts\ (s\ l)\ t = HOLD\ k$
⟨*proof*⟩

lemma *s-case-expand*:
$(case\ s\ of\ (c, n, ts) \Rightarrow P\ c\ n\ ts) = P\ (cc\ s)\ (nn\ s)\ (tts\ s)$
⟨*proof*⟩

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

lemma *fair-aux2*:
assumes *RUN*: *A.is-run s*
assumes *ACQ*: $t{<}N$ $tts\ (s\ i)\ t{=}INIT$ $tts\ (s\ (Suc\ i))\ t{=}WAIT\ k$
assumes *HOLD*: $i{<}j$ $tts\ (s\ j)\ t = HOLD\ k$
assumes *WAIT*: $t'{<}N$ $tts\ (s\ i)\ t' = WAIT\ k'$
obtains *l* **where** $i{<}l$ $l{<}j$ $tts\ (s\ l)\ t' = HOLD\ k'$
⟨*proof*⟩

lemma *find-hold-position*:
assumes *RUN*: *A.is-run s*
assumes *WAIT*: $t{<}N$ $tts\ (s\ i)\ t = WAIT\ tk$
assumes *NWAIT*: $i{<}j$ $tts\ (s\ j)\ t \ne WAIT\ tk$
obtains *l* **where** $i{<}l$ $l{\le}j$ $tts\ (s\ l)\ t = HOLD\ tk$
⟨*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 *fair*:
assumes *RUN*: *A.is-run s*
assumes *ACQ*: $t{<}N$ $tts\ (s\ i)\ t{=}INIT$ $is\text{-}WAIT\ (tts\ (s\ (Suc\ i))\ t)$
— Thread t calls *acquire* in step i
assumes *HOLD*: $i{<}j$ $is\text{-}HOLD\ (tts\ (s\ j)\ t)$
— Thread t holds lock in step j
assumes *WAIT*: $t'{<}N$ $is\text{-}WAIT\ (tts\ (s\ i)\ t')$
— Thread t' waits for lock at step i
obtains *l* **where** $i{<}l$ $l{<}j$ $is\text{-}HOLD\ (tts\ (s\ l)\ t')$
— 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 *invar2*
$\equiv \lambda(c,n,ts).\ \forall k.\ c{\le}k \wedge k{<}n \longrightarrow (\exists t{<}N.\ has\text{-}ticket\ (ts\ t)\ k)$

lemma *is-invar2*: *A.is-invar invar2*
⟨*proof*⟩

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

corollary *current-lock-used*:
assumes *R*: *A.reachable* (c,n,ts)
assumes *WAIT*: $t<N$ $ts\ t = WAIT\ k$
obtains t' **where** $t'<N$ *has-ticket* $(ts\ t')\ c$
⟨*proof*⟩

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* $(ts\ t')\ k$
⟧ $\Longrightarrow t'=t$
⟨*proof*⟩

We define the thread that holds a specified ticket

definition *tkt-thread* $\equiv \lambda ts\ k.$ *THE* $t.\ t<N \wedge$ *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* $ts\ k = t$
⟨*proof*⟩

lemma *holds-only-current*:
assumes *R*: *A.reachable* (c,n,ts)
assumes *A*: $t<N$ *locks-ticket* $(ts\ t)\ k$
shows $k=c$
⟨*proof*⟩

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 WAIT - $\Rightarrow$ *3::nat* | *HOLD* - $\Rightarrow$ *2* | *REL* - $\Rightarrow$ *1* | *INIT* $\Rightarrow$ *0*

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

theorem *progress*:
assumes *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*⟩

3.2.3 Refinement 2: Bounding the Counters

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

consts *k::nat*

specification (k) *k-not0*[*simp*]: $k{\neq}0$ ⟨*proof*⟩
lemma *k-gt0*[*simp*]: $0{<}k$ ⟨*proof*⟩

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

type-synonym $bstate = nat \times nat \times (nat \Rightarrow thread)$

The step relation of a single thread

inductive *bstep-sng* **where**
enter-wait: $bstep\text{-}sng\ (c,n,INIT)\ (c,(n+1)\ mod\ (k*N),WAIT\ (n\ mod\ N))$
| *loop-wait*: $c{\neq}tk \Longrightarrow bstep\text{-}sng\ (c,n,WAIT\ tk)\ (c,n,WAIT\ tk)$
| *exit-wait*: $bstep\text{-}sng\ (c,n,WAIT\ c)\ (c,n,HOLD\ c)$
| *start-release*: $bstep\text{-}sng\ (c,n,HOLD\ tk)\ (c,n,REL\ tk)$
| *release*: $bstep\text{-}sng\ (c,n,REL\ tk)\ ((tk+1)\ mod\ N,n,INIT)$

The step relation of the system, labeled with the thread *t* that performs the step

inductive *blstep* **for** *t* **where**
$[\![\ t{<}N;\ bstep\text{-}sng\ (c,n,ts\ t)\ (c',n',s')\ ]\!]$
$\Longrightarrow blstep\ t\ (c,n,ts)\ (c',n',ts(t{:=}s'))$

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*: $B.can\text{-}step\ l\ s \longleftrightarrow l{<}N$
⟨*proof*⟩

interpretation *B*: $df\text{-}system\ bs_0\ blstep$
⟨*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*]:
$finite\ s \Longrightarrow sum\ f\ s \neq (0{::}nat) \longleftrightarrow (\exists x{\in}s.\ f\ x \neq 0)$
⟨*proof*⟩

lemma *n-eq-Suc-sub1-conv*[*simp*]: $n = Suc\ (n - Suc\ 0) \longleftrightarrow n{\neq}0$ ⟨*proof*⟩

lemma *mod-mult-mod-eq*[*mod-simps*]: $x\ mod\ (k * N)\ mod\ N = x\ mod\ N$
⟨*proof*⟩

lemma *mod-eq-imp-eq-aux*: $b\ mod\ N = (a{::}nat)\ mod\ N \Longrightarrow a{\le}b \Longrightarrow b{<}a{+}N \Longrightarrow b{=}a$
⟨*proof*⟩

lemma *mod-eq-imp-eq*:

$[\![b \leq x; x < b + N; b \leq y; y < b + N; x\ mod\ N = y\ mod\ N\ ]\!] \Longrightarrow x{=}y$
⟨*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-addsimps*[*simp*]:
map-ticket f t = INIT $\longleftrightarrow$ *t=INIT*
map-ticket f t = WAIT tk $\longleftrightarrow$ $(\exists tk'.\ tk{=}f\ tk' \wedge t{=}WAIT\ tk')$
map-ticket f t = HOLD tk $\longleftrightarrow$ $(\exists tk'.\ tk{=}f\ tk' \wedge t{=}HOLD\ tk')$
map-ticket f t = REL tk $\longleftrightarrow$ $(\exists tk'.\ tk{=}f\ tk' \wedge t{=}REL\ tk')$
⟨*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* $\leq$ *Suc 0*
⟨*proof*⟩

definition *num-ni ts* ≡ $\sum$ *i=0..<N. ni-weight (ts i)*
lemma *num-ni-init*[*simp*]: *num-ni* (λ-. *INIT*) = *0* ⟨*proof*⟩

lemma *num-ni-upd*:
t<N ⟹ *num-ni (ts(t:=s)) = num-ni ts − ni-weight (ts t) + ni-weight s*
⟨*proof*⟩

lemma *num-ni-nz-if*[*simp*]: $[\![t < N;\ ts\ t \neq INIT]\!] \Longrightarrow num\text{-}ni\ ts \neq 0$
⟨*proof*⟩

lemma *num-ni-leN*: *num-ni ts* $\leq$ *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*).

$ci = c \ mod \ N$
$\wedge \ ni = n \ mod \ (k*N)$
$\wedge \ tsi = (map\text{-}ticket \ (\lambda t. \ t \ mod \ N)) \ o \ ts$

lemma *sraux*:
sim-rel1 $(c,n,ts) \ (ci,ni,tsi) \Longrightarrow ci = c \ mod \ N \wedge ni = n \ mod \ (k*N)$
⟨*proof*⟩

lemma *sraux2*: $[\![$*sim-rel1* $(c,n,ts) \ (ci,ni,tsi); \ t<N]\!]$
$\Longrightarrow tsi \ t = map\text{-}ticket \ (\lambda x. \ x \ mod \ N) \ (ts \ t)$
⟨*proof*⟩

interpretation *sim1*: *simulationI* as_0 *alstep* bs_0 *blstep sim-rel1*
⟨*proof*⟩

Transfer of Properties

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

lemma *xfer-locks-ticket*:
assumes *locks-ticket* $(map\text{-}ticket \ (\lambda t. \ t \ mod \ N) \ (ts \ t))$ *tki*
obtains *tk* **where** $tki{=}tk \ mod \ N$ *locks-ticket* $(ts \ t)$ *tk*
⟨*proof*⟩

lemma *b-holds-only-current*:
$[\![B.reachable \ (c, \ n, \ ts); \ t < N; \ locks\text{-}ticket \ (ts \ t) \ tk]\!] \Longrightarrow tk = c$
⟨*proof*⟩

lemma *b-mutual-exclusion'*: $[\![B.reachable \ s;$
$t<N; \ t'<N; \ t\neq t'; \ locks\text{-}ticket \ (tts \ s \ t) \ tk; \ locks\text{-}ticket \ (tts \ s \ t') \ tk'$
$]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *xfer-has-ticket*:
assumes *has-ticket* $(map\text{-}ticket \ (\lambda t. \ t \ mod \ N) \ (ts \ t))$ *tki*
obtains *tk* **where** $tki{=}tk \ mod \ N$ *has-ticket* $(ts \ t)$ *tk*
⟨*proof*⟩

lemma *has-ticket-in-range*:
assumes *Ra*: $A.reachable \ (c,n,ts)$ **and** $t<N$ **and** *U*: *has-ticket* $(ts \ t)$ *tk*
shows $c\leq tk \wedge tk<c+N$
⟨*proof*⟩

lemma *b-has-ticket-unique*: $[\![B.reachable \ (ci,ni,tsi);$
$t<N; \ has\text{-}ticket \ (tsi \ t) \ tki; \ t'<N; \ has\text{-}ticket \ (tsi \ t') \ tki$
$]\!] \Longrightarrow t'{=}t$

⟨*proof*⟩

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*′) ⟧
⟹ *clstep t* (*c*,*n*,*ts*) (*c*′,*n*′,*ts*(*t*:=*s*′))

Initial state of the system

definition cs_0 ≡ ((*replicate N False*)[*0*:=*True*], *0*, λ-. *INIT*)

interpretation *C*: *system* cs_0 *clstep* ⟨*proof*⟩

lemma *c-never-blocked*: *C*.*can-step l s* ⟷ *l*<*N*
⟨*proof*⟩

interpretation *C*: *df-system* cs_0 *clstep*
⟨*proof*⟩

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

definition *invar4*
≡ λ(*c*,*n*,*ts*). *c*<*N* ∧ (∀*t*<*N*. ∀*tk*. *has-ticket* (*ts t*) *tk* ⟶ *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* ≡ λ*ts*. ∃*t*<*N*. ∃*tk*. *ts t* = *REL tk*

lemma *is-REL-state-simps*[*simp*]:
$t<N \Longrightarrow$ *is-REL-state* (*ts*(*t*:=*REL tk*))
$t<N \Longrightarrow \neg$*is-REL* (*ts t*) $\Longrightarrow \neg$*is-REL* s'
$\Longrightarrow$ *is-REL-state* (*ts*(*t*:=s')) $\longleftrightarrow$ *is-REL-state ts*
⟨*proof*⟩

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 $tk \neq c$
⟨*proof*⟩

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

Simulation relation that implements current ticket by array

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

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.

definition *sim-rel* $\equiv$ *sim-rel1 OO sim-rel2*
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*) $\longleftrightarrow$ *is-HOLD* (*tts s t*)
⟨*proof*⟩

lemma *xfer-waits*:

assumes *sim-rel s cs*
shows $is\text{-}WAIT\ (tts\ cs\ t) \longleftrightarrow is\text{-}WAIT\ (tts\ s\ t)$
⟨*proof*⟩

lemma *xfer-init*:
assumes *sim-rel s cs*
shows $tts\ cs\ t = INIT \longleftrightarrow tts\ s\ t = INIT$
⟨*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 fairness here:

$C.step\ s\ s' = (\exists l.\ clstep\ l\ s\ s')$

$C.reachable \equiv C.step^{**}\ cs_0$
$C.is\text{-}lrun\ l\ s \equiv s\ 0 = cs_0 \wedge (\forall i.\ clstep\ (l\ i)\ (s\ i)\ (s\ (Suc\ i)))$
$C.is\text{-}run\ s \equiv \exists l.\ C.is\text{-}lrun\ l\ s$
$C.is\text{-}lfair\ ls\ ss \equiv \forall l\ i.\ \exists j \geq i.\ \neg\ C.can\text{-}step\ l\ (ss\ j) \vee ls\ j = l$
$C.is\text{-}fair\text{-}run\ s \equiv \exists l.\ C.is\text{-}lrun\ l\ s \wedge C.is\text{-}lfair\ l\ s$

Safety

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

theorem *final-mutual-exclusion*: $[\![C.reachable\ s;$
$t{<}N;\ t'{<}N;\ t{\neq}t';\ is\text{-}HOLD\ (tts\ s\ t);\ is\text{-}HOLD\ (tts\ s\ t')$
$]\!] \Longrightarrow False$
⟨*proof*⟩

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

— Thread t holds lock in step j
assumes *WAIT*: $t' < N$ **and** *is-WAIT* $(tts\ (s\ i)\ t')$
— Thread t' waits for lock at step i
obtains l **where** $i < l$ **and** $l < j$ **and** *is-HOLD* $(tts\ (s\ l)\ t')$
— Then, t' gets lock earlier
⟨*proof*⟩

Liveness

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

theorem *final-progress*:
assumes *FRUN*: *C.is-fair-run s*
assumes *WAIT*: $t < N$ **and** *is-WAIT* $(tts\ (s\ i)\ t)$
shows $\exists j > i.$ *is-HOLD* $(tts\ (s\ j)\ t)$
⟨*proof*⟩

end