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

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

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

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

Gap Buffer

1.1 Challenge

A gap buffer is a data structure for the implementation of text editors, which can efficiently move the cursor, as well add and delete characters.
The idea is simple: the editor's content is represented as a character array a of length n, which has a gap of unused entries $a[l],\ldots,a[r-1]$, with respect to two indices $l \leq r$. The data it represents is composed as $a[0],\ldots,a[l-1],a[r],...,a[n-1]$.
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*
unfolding *empty-def* **by** *auto*
lemma *empty-text[simp]*: *get-text empty = []*
unfolding *empty-def* **by** *auto*
lemma *move-left-pos[simp]*: *get-pos (move-left b) = get-pos b − 1*
— Note that *0 − 1 = 0* in Isabelle
by *(cases b) auto*
lemma *move-left-text[simp]*: *get-text (move-left b) = get-text b*
by *(cases b) auto*

lemma *move-right-pos[simp]*:

get-pos (*move-right b*) = *min* (*get-pos b*+*1*) (*length* (*get-text b*))
by (*cases b*) *auto*
lemma *move-right-text*[*simp*]: *get-text* (*move-right b*) = *get-text b*
by (*cases b*) *auto*

lemma *insert-pos*[*simp*]: *get-pos* (*insert x b*) = *get-pos b* + *1*
by (*cases b*) *auto*
lemma *insert-text*: *get-text* (*insert x b*)
= *take* (*get-pos b*) (*get-text b*)@*x*#*drop* (*get-pos b*) (*get-text b*)
by (*cases b*) *auto*

lemma *delete-pos*[*simp*]: *get-pos* (*delete b*) = *get-pos b* − *1*
by (*cases b*) *auto*
lemma *delete-text*: *get-text* (*delete b*)
= *take* (*get-pos b*−*1*) (*get-text b*)@*drop* (*get-pos b*) (*get-text b*)
by (*cases b*) *auto*

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

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

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

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

context
notes[*simp*] =
empty-def get-text-def get-pos-def move-left-def move-right-def
insert-def delete-def conv-to-is-Nil
begin
sepref-decl-op (*no-def*) *empty* :: ⟨*A*⟩*textbuffer-rel* .
sepref-decl-op (*no-def*) *get-text* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*list-rel* .
sepref-decl-op (*no-def*) *get-pos* :: ⟨*A*⟩*textbuffer-rel* → *nat-rel* .
sepref-decl-op (*no-def*) *move-left* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* .
sepref-decl-op (*no-def*) *move-right* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* .
sepref-decl-op (*no-def*) *insert* :: *A*→⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* .
sepref-decl-op (*no-def*) *delete* :: ⟨*A*⟩*textbuffer-rel* → ⟨*A*⟩*textbuffer-rel* .
end

1.2.2 Refinement 1: List with Gap

1.2.3 Implementation on List-Level

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

Abstraction Relation

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

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

Empty

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

Left

```
definition move-left1 ≡ λ(l,r,buf). doN {
  if l≠0 then doN {
    ASSERT(r−1<length buf ∧ 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
  apply clarsimp
  unfolding move-left1-def
  apply refine-vcg
  apply (auto
    simp: in-br-conv gap-α-def gap-invar-def move-left1-def
    split: prod.splits)
  subgoal by (simp add: butlast-take)
  subgoal
    by (smt Cons-nth-drop-Suc One-nat-def Suc-pred diff-Suc-less
      drop-update-cancel last-take-nth-conv le-trans length-list-update
      less-le-trans neq0-conv nth-list-update-eq)
  done
```

Right

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

```
  if r<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*
  apply *clarsimp*
  unfolding *move-right1-def*
  apply *refine-vcg*
  unfolding *gap-α-def gap-invar-def*
  apply (*auto*
    simp: *in-br-conv hd-drop-conv-nth take-update-last*
    split: *prod.split*)
  by (*simp add*: *drop-Suc tl-drop*)

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*′)))
  unfolding *grow1-def*
  apply *refine-vcg*
  using *assms*
  unfolding *gap-α-def gap-invar-def can-insert-def*
  apply (*auto simp*: *op-list-blit-def*)
  done

definition *insert1 x* ≡ 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*
apply *clarsimp*
unfolding *insert1-def*
apply *refine-vcg*
unfolding *insert-def gap-α-def gap-invar-def can-insert-def*
apply (*auto simp*: *in-br-conv take-update-last split*: *prod.split*)
done

Delete

definition *delete1*
≡ λ (*l*,*r*,*buf*). *if l>0 then RETURN* (*l−1*,*r*,*buf*) *else RETURN* (*l*,*r*,*buf*)
lemma *delete1-correct*:
(*delete1*,*RETURN o delete*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
apply *clarsimp*
unfolding *delete1-def*
apply *refine-vcg*
unfolding *gap-α-def gap-invar-def*
by (*auto simp*: *in-br-conv butlast-take split*: *prod.split*)

1.2.4 Imperative Arrays and Executable Code

abbreviation *gap-impl-assn* ≡ *nat-assn* $\times_a$ *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*
unfolding *empty1-def array.fold-custom-empty*
by *sepref*
sepref-decl-impl *empty-impl*: *empty-impl.refine*[*FCOMP empty1-correct*] **.**

sepref-definition *move-left-impl*
is *move-left1* :: *gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *move-left1-def* **by** *sepref*
sepref-decl-impl *move-left-impl*: *move-left-impl.refine*[*FCOMP move-left1-correct*] **.**

sepref-definition *move-right-impl*
is *move-right1* :: *gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *move-right1-def* **by** *sepref*
sepref-decl-impl *move-right-impl*: *move-right-impl.refine*[*FCOMP move-right1-correct*]
.

sepref-definition *insert-impl*
is *uncurry insert1* :: *id-assn*$^k *_a$*gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *insert1-def grow1-def* **by** *sepref*

— We inline *grow1* here
sepref-decl-impl *insert-impl*: *insert-impl.refine*[*FCOMP insert1-correct*] **.**

sepref-definition *delete-impl*
is *delete1* :: $gap\text{-}impl\text{-}assn^d \rightarrow_a gap\text{-}impl\text{-}assn$
unfolding *delete1-def* **by** *sepref*
sepref-decl-impl *delete-impl*: *delete-impl.refine*[*FCOMP delete1-correct*] **.**

end

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

(*uncurry0 Challenge1.empty-impl*, *uncurry0* (*RETURN Challenge1.empty*))
$\in id\text{-}assn^k \rightarrow_a gap\text{-}assn\ ?A$
(*move-left-impl*, *RETURN* ∘ *move-left*) $\in (gap\text{-}assn\ ?A)^d \rightarrow_a gap\text{-}assn\ ?A$
(*move-right-impl*, *RETURN* ∘ *move-right*) $\in (gap\text{-}assn\ ?A)^d \rightarrow_a gap\text{-}assn\ ?A$
CONSTRAINT is-pure ?A $\Longrightarrow$
(*uncurry Challenge1.insert-impl*, *uncurry* (*RETURN* ∘∘ *Challenge1.insert*))
$\in ?A^k *_a (gap\text{-}assn\ ?A)^d \rightarrow_a gap\text{-}assn\ ?A$
(*delete-impl*, *RETURN* ∘ *delete*) $\in (gap\text{-}assn\ ?A)^d \rightarrow_a gap\text{-}assn\ ?A$

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* (λ*f*. *f*) [
insert (*1*::*int*),
insert (*2*::*int*),
insert (*3*::*int*),
insert (*5*::*int*),
move-left,
insert (*4*::*int*),
move-right,
insert (*6*::*int*),
delete
] *empty*)

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

sepref-definition *client-impl*
is *uncurry0 client* :: $unit\text{-}assn^k \rightarrow_a gap\text{-}assn\ id\text{-}assn$
unfolding *client-def fold.simps id-def comp-def*
by *sepref*

ML-val ‹
  @{*code client-impl*} ()
›

end

1.3 Shorter Solution

theory *Challenge1-short*
imports *lib/VTcomp*
begin

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

1.3.1 Abstract Specification

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

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

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

1.3.2 Refinement 1: List with Gap

1.3.3 Implementation on List-Level

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

Abstraction Relation

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

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

Left

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

```
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
  apply clarsimp
  unfolding move-left1-def
  apply refine-vcg
  apply (auto
    simp: in-br-conv gap-α-def gap-invar-def move-left1-def
    split: prod.splits)

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

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
  apply clarsimp
  unfolding move-right1-def
  apply refine-vcg
  unfolding gap-α-def gap-invar-def
  apply (auto simp: in-br-conv split: prod.split)
  apply (rule nth-equalityI)
   apply (simp-all add: Cons-nth-drop-Suc take-update-last)
  done
```

Insert and Grow

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

definition *grow1 K* $\equiv \lambda(l,r,buf)$. *doN* {
let b = op-array-replicate (*length buf* + *K*) *default*;
b ← *mop-list-blit buf 0 b 0 l*;
b ← *mop-list-blit buf r b* (*r+K*) (*length buf* − *r*);
RETURN (*l*,*r+K*,*b*)
}
— Note: Most operations have also a variant prefixed with *mop*. These are defined in the refinement monad and already contain the assertion of their precondition. The backside is that they cannot be easily used in as part of expressions, e.g., in *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* ($\lambda gb'$.
gap-invar gb'
∧ *gap*-α gb' = *gap*-α *gb*
∧ (*K>0* ⟶ *can-insert* gb')))
unfolding *grow1-def*
apply *refine-vcg*
using *assms*
unfolding *gap-α-def gap-invar-def can-insert-def*
apply (*auto simp*: *op-list-blit-def*)
done

definition *insert1 x* $\equiv \lambda(l,r,buf)$. *doN* {
(*l*,*r*,*buf*) ←
if (*l=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*
apply *clarsimp*
unfolding *insert1-def*
apply *refine-vcg* — VCG knows the rule for grow1 already
unfolding *insert-def gap-α-def gap-invar-def can-insert-def*
apply (*auto simp*: *in-br-conv take-update-last split*: *prod.split*)
done

Delete

definition *delete1*
$\equiv \lambda(l,r,buf)$. *if l>0 then RETURN* (*l−1*,*r*,*buf*) *else RETURN* (*l*,*r*,*buf*)
lemma *delete1-correct*:
(*delete1*,*RETURN o delete*) ∈ *gap-rel* → ⟨*gap-rel*⟩*nres-rel*
apply *clarsimp*

unfolding *delete1-def*
apply *refine-vcg*
unfolding *gap-α-def gap-invar-def*
by (*auto simp*: *in-br-conv butlast-take split*: *prod.split*)

1.3.4 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*
unfolding *move-left1-def* **by** *sepref*

sepref-definition *move-right-impl*
is *move-right1* :: *gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *move-right1-def* **by** *sepref*

sepref-definition *insert-impl*
is *uncurry insert1* :: *id-assn*$^k *_a$*gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *insert1-def grow1-def* **by** *sepref*
— We inline *grow1* here

sepref-definition *delete-impl*
is *delete1* :: *gap-impl-assn*$^d \rightarrow_a$*gap-impl-assn*
unfolding *delete1-def* **by** *sepref*

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

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

Proves:

(*move-left-impl*, *RETURN* ∘ *move-left*) ∈ *gap-assn*$^d \rightarrow_a$ *gap-assn*

(*move-right-impl*, *RETURN* ∘ *move-right*) ∈ *gap-assn*$^d \rightarrow_a$ *gap-assn*

(*uncurry Challenge1-short.insert-impl*,
uncurry (*RETURN* ∘∘ *Challenge1-short.insert*))
∈ *id-assn*$^k *_a$ *gap-assn*$^d \rightarrow_a$ *gap-assn*

(*delete-impl*, *RETURN* ∘ *delete*) ∈ *gap-assn*$^d \rightarrow_a$ *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:

```
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*
by (*induction rule*: *valid.induct*)
(*auto 4 4 intro*: *valid′.intros elim*: *valid′.cases*
simp: *replicate-add*[*symmetric*] *append-assoc*[*symmetric*]
)

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

lemma *valid′-valid*:
valid′ l ⟹ *valid l*
by (*induction rule*: *valid′.induct*) (*auto intro*: *valid.intros valid-red*)

lemma *valid-eq-valid′*:
valid′ l = *valid l*
using *valid-valid′ valid′-valid* **by** *metis*

Additional Facts on Replicate

lemma *replicate-iff*:
(∀*i*<*length l*. *l* ! *i* = *R*) ⟷ (∃ *n*. *l* = *replicate n R*)
by *auto* (*metis* (*full-types*) *in-set-conv-nth replicate-eqI*)

lemma *replicate-iff2*:
(∀*i*<*n*. *l* ! *i* = *R*) ⟷ (∃ *l′*. *l* = *replicate n R* @ *l′*) **if** *n* < *length l*
using *that* **by** (*auto simp*: *list-eq-iff-nth-eq nth-append intro*: *exI*[**where** *x* = *drop n l*])

lemma *replicate-Cons-eq*:
replicate n x = *y* # *ys* ⟷ (∃ *n′*. *n* = *Suc n′* ∧ *x* = *y* ∧ *replicate n′ x* = *ys*)
by (*cases n*) *auto*

Main Case Analysis on @*term valid*

lemma *valid-split*:
valid l ⟷
l = [] ∨
(*l*!*0* = *B* ∧ *valid* (*tl l*)) ∨
length l ≥ *3* ∧ (∀ *i* < *length l*. *l* ! *i* = *R*) ∨
(∃ *j* < *length l*. *j* ≥ *3* ∧ (∀ *i* < *j*. *l* ! *i* = *R*) ∧ *l* ! *j* = *B* ∧ *valid* (*drop* (*j* + *1*) *l*))
unfolding *valid-eq-valid′*[*symmetric*]
apply *standard*
subgoal
by (*erule valid′.cases*) (*auto simp*: *nth-append nth-Cons split*: *nat.splits*)
subgoal
by (*auto intro*: *valid′.intros simp*: *replicate-iff elim*!: *disjE1*)
(*fastforce intro*: *valid′.intros simp*: *neq-Nil-conv replicate-iff2 nth-append*)+
done

Base cases

lemma *lc0-aux*:
{*l*. *l* = [] ∧ *valid l*} = {[]}
by (*auto intro*: *valid.intros*)

lemma *lc0*: *lcount 0* = *1*

by (*auto simp*: *lc0-aux lcount-def*)

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

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

lemma *valid-3R*: ‹*valid* [*R*, *R*, *R*]›
using *valid.intros*(*3*) [*of* ‹[]› *3*] **by** (*simp add*: *numeral-eq-Suc valid.intros*)

lemma *lc3-aux*: {*l*. *length l*=*3* ∧ *valid l*} = {[*B*,*B*,*B*], [*R*,*R*,*R*]}
by (*auto 4 4 intro*: *valid.intros valid-3R elim*: *valid.cases*
simp: *replicate-Cons-eq*)

lemma *lcounts-init*: *lcount 0* = *1 lcount 1* = *1 lcount 2* = *1 lcount 3* = *2*
using *lc0 lc1aux lc2aux lc3-aux* **unfolding** *lcount-def* **by** *simp-all*

The Recursion Case

lemma *finite-valid-length*:
finite {*l*. *length l* = *n* ∧ *valid l*} (**is** *finite ?S*)
proof −
have *?S* ⊆ *lists* {*R*, *B*} ∩ {*l*. *length l* = *n*}
by (*auto intro*: *color.exhaust*)
moreover have *finite* ...
by (*auto intro*: *lists-of-len-fin1*)
ultimately show *?thesis*
by (*rule finite-subset*)
qed

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

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

lemma *replicate-unequal-aux*:
replicate x R @ *B* # *l* ≠ *replicate y R* @ *B* # *l′* (**is** *?l* ≠ *?r*) **if** ‹*x* < *y*› **for** *l l′*
proof −
have *?l* ! *x* = *B ?r* ! *x* = *R*
using *that* **by** (*auto simp*: *nth-append*)
then show *?thesis*
by *auto*
qed

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

lemma lcrec: lcount n = lcount (n−1) + 1 + (∑i=3..<n. lcount (n−i−1)) if ‹n>3›
proof −
  have {l. length l = n ∧ valid l}
      = {l. length l = n ∧ valid (tl l) ∧ l!0=B}
      ∪ {l. length l = n ∧
         (∃ i. i < n ∧ i ≥ 3 ∧ (∀ k < i. l!k = R) ∧ l!i = B ∧ valid (drop (i + 1) l))}
      ∪ {l. length l = n ∧ (∀i<n. l!i=R)}
       (is ?A = ?B ∪ ?D ∪ ?C)
    using ‹n > 3› by (subst valid-split) auto

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

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

  let ?D1=(⋃ i ∈ {3..<n}. (λ l. replicate i R @ B # l)‘ {l. length l = n − i − 1 ∧ valid
l})
  have ?D =
     (⋃i ∈ {3..<n}. {l. length l = n ∧ (∀ k < i. l!k = R) ∧ l!i = B ∧ valid (drop (i + 1)
l)})
    by auto
  have {l. length l = n ∧ (∀ k < i. l!k = R) ∧ l!i = B ∧ valid (drop (i + 1) l)}
         = (λ l. replicate i R @ B # l)‘ {l. length l = n − i − 1 ∧ valid l}
    if i < n 2 < i for i
    apply safe
    subgoal for l
      apply (rule image-eqI[where x = drop (i + 1) l])
       apply (rule nth-equalityI)
      using that
        apply (simp-all split: nat.split add: nth-Cons nth-append)
      using add-diff-inverse-nat apply fastforce
      done
    using that by (simp add: nth-append; fail)+

  then have D-eq: ?D = ?D1
    unfolding ‹?D = -› by auto
```

```
have inj: inj-on (λl. replicate x R @ B # l) {l. length l = n − Suc x ∧ valid l} for x
  unfolding inj-on-def by auto

have ∗:
  (λl. replicate x R @ B # l) ‘ {l. length l = n − Suc x ∧ valid l} ∩
    (λl. replicate y R @ B # l) ‘ {l. length l = n − Suc y ∧ valid l} = {}
  if 3 ≤ x x < y y < n for x y
  using that replicate-unequal-aux[OF ‹x < y›] by auto

have 3: card ?D1 = (∑i=3..<n. lcount (n−i−1))
proof (subst card-Union-disjoint, goal-cases)
  case 1
  show ?case
    unfolding pairwise-def disjnt-def
  proof (clarsimp, goal-cases)
    case prems: (1 x y)
    from prems show ?case
      apply −
      apply (rule linorder-cases[of x y])
        apply (rule ∗; assumption)
       apply (simp; fail)
      apply (subst Int-commute; rule ∗; assumption)
      done
  qed
next
  case 3
  show ?case
  proof (subst sum.reindex, unfold inj-on-def, clarsimp, goal-cases)
    case prems: (1 x y)
    with ∗[of y x] ∗[of x y] valid-line-aux[of n − Suc x] show ?case
      by − (rule linorder-cases[of x y], auto)
  next
    case 2
    then show ?case
      by (simp add: lcount-def card-image[OF inj])
  qed
qed (auto intro: finite-subset[OF - finite-valid-length])

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

     apply (blast intro: finite-subset[OF - finite-valid-length])+

  subgoal
    using Cons-replicate-eq[of B - n R] replicate-unequal-aux by fastforce
  apply (subst card-Un-disjoint)
```

```
    apply (blast intro: finite-subset[OF - finite-valid-length])+

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

2.2.3 Verification of Program

Inner Loop: Summation

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

lemma sum-spec[THEN SPEC-trans, refine-vcg]:
  assumes l≤u
  assumes ⋀i. l≤i ⟹ i<u ⟹ Φ i
  shows sum-prog Φ l u f ≤ SPEC (λr. r=(∑i=l..<u. f i))
  unfolding sum-prog-def
  supply nfoldli-upt-rule[where I=λj s. s=(∑i=l..<j. f i), refine-vcg]
  apply refine-vcg
  using assms
  apply auto
  done
```

Main Program

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

  ASSERT (∀i<4. c!i = lcount i);

  c←nfoldli [4..<M+1] (λ-. True) (λn c. doN {
    let sum = (∑i=3..<n. c!(n-i-1));
    sum ← sum-prog (λi. n−i−1 < length c) 3 n (λi. 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 (λr. r=lcount M)
  unfolding icount-def
  thm nfoldli-upt-rule
  supply nfoldli-upt-rule[where
   I=λn c. length c = M+1 ∧ ( ∀i<n. c!i = lcount i), refine-vcg]
  apply refine-vcg
  apply (auto simp:)
  subgoal for i
   apply (subgoal-tac i∈{0,1,2,3}) using lcounts-init
   by (auto)

  subgoal for i c j
   apply (cases j<i)
   apply auto
   apply (subgoal-tac i=j)
   apply auto
   apply (subst lcrec[where n=j])
   apply auto
   done
  done
```

2.2.4 Refinement to Imperative Code

```
sepref-definition icount-impl is icount :: nat-assn^k →_a nat-assn
  unfolding icount-def sum-prog-def
  by sepref
```

Main Correctness Statement

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

```
theorem icount-impl-correct:
  M>2 ⟹ <emp> icount-impl M <λr. ↑(r = lcount M)>_t
proof –
  note A = icount-impl.refine[to-hnr, THEN hn-refineD]
  note A = A[unfolded autoref-tag-defs]
  note A = A[unfolded hn-ctxt-def pure-def, of M M, simplified]
  note [sep-heap-rules] = A

  assume M>2

  show ?thesis
   using icount-correct[OF ‹M>2›]
   by (sep-auto simp: refine-pw-simps pw-le-iff)
qed
```

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 ⟧ ⟹ valid-point i
  | ⟦1≤i;i+1<length l; l!(i−1)=R; l!(i) = R; l!(i+1) = R ⟧ ⟹ valid-point i
  | ⟦2≤i; i<length l; l!(i−2)=R; l!(i−1) = R; l!(i) = R ⟧ ⟹ valid-point i
  | ⟦ i<length l; l!i=B⟧ ⟹ valid-point i


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

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

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

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

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

```
  apply (fastforce intro: valid-point.intros)
  done

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

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

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

```
    j<length l 3 ≤ j (∀i<j. l ! i = R) l ! j = B valid-line (drop (j + 1) l)
    by blast
  show ?thesis
  proof (rule valid-lineI)
    fix i assume i < length l
    with ‹j ≥ 3› consider i ≤ j − 3 | i = j − 2 | i = j − 1 | i = j | i > j
      by atomize-elim auto
    then show valid-point l i
    proof cases
      case 5
      with ‹valid-line -› ‹i < length l› have valid-point (drop (j + 1) l) (i − j − 1)
        unfolding valid-line-def by auto
      then show ?thesis
        using ‹i > j› by cases (auto intro: valid-point.intros)
    qed (use j in ‹auto intro: valid-point.intros›)
  qed
 qed
qed

lemma valid-line-cases:
 l = [] ∨
 (l!0 = B ∧ valid-line (tl l)) ∨
 length l ≥ 3 ∧ (∀ i < length l. l ! i = R) ∨
 (∃ j < length l. j ≥ 3 ∧ (∀ i < j. l ! i = R) ∧ l ! j = B ∧ valid-line (drop (j + 1) l))
 if valid-line l
proof (cases l = [])
 case True
 then show ?thesis
   by (simp add: valid-line-def)
next
 case False
 show ?thesis
 proof (cases l!0 = B)
   case True
   with ‹l ≠ []› have l = B # tl l
     by (cases l) auto
   with ‹valid-line l› True show ?thesis
     by (metis valid-line-prepend-B-iff)
 next
   case False
   from ‹valid-line l› ‹l ≠ []› have valid-point l 0
     unfolding valid-line-def by auto
   with False have red-start: length l ≥ 3 l!0 = R l!1 = R l!2 = R
     by (auto elim!: valid-point.cases simp: numeral-2-eq-2)
   show ?thesis
   proof (cases ∀i < length l. l ! i = R)
     case True
     with ‹length l ≥ 3› show ?thesis
       by auto
```

```
next
  case False
  let ?S = {j. j < length l ∧ j ≥ 3 ∧ l ! j = B} let ?j = Min ?S
  have B-ge-3: i ≥ 3 if l ! i = B for i
  proof −
    consider i = 0 | i = 1 | i = 2 | i ≥ 3
      by atomize-elim auto
    then show i ≥ 3
      using red-start ‹l ! i = B› by cases auto
  qed
  from False obtain i where l ! i = B i < length l i ≥ 3
    by (auto intro: B-ge-3 color.exhaust)
  then have ?j ∈ ?S
    by − (rule Min-in, auto)
  have ∀ i < ?j. l ! i = R
  proof −
    {
      fix i assume i < ?j l ! i = B
      then have i ≥ 3
        by (auto intro: B-ge-3)
      with ‹i < ?j› ‹l ! i = B› red-start ‹?j ∈ ?S› have i ∈ ?S
        by auto
      then have ?j ≤ i
        by (auto intro: Min-le)
      with ‹i < ?j› have False
        by simp
    }
    then show ?thesis
      by (auto intro: color.exhaust)
  qed
  with ‹?j ∈ ?S› obtain j where j: j < length l j ≥ 3 ∀ i < j. l ! i = R l ! j = B
    by blast
  moreover have valid-line (drop (j + 1) l)
  proof (rule valid-lineI)
    fix i assume i < length (drop (j + 1) l)
    with j ‹valid-line l› have valid-point l (j + i + 1)
      unfolding valid-line-def by auto
    then show valid-point (drop (j + 1) l) i
    proof cases
      case 2
      then show ?thesis
        using j by (cases i) (auto intro: valid-point.intros)
    next
      case prems: 3
      consider i = 0 | i = 1 | i > 1
        by atomize-elim auto
      then show ?thesis
        using j prems by cases (auto intro: valid-point.intros)
    qed (auto intro: valid-point.intros)
```

qed
ultimately show *?thesis*
by *auto*
qed
qed
qed

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

Connection to the easier definition given above

lemma *valid-valid-line*:
$valid\ l \longleftrightarrow valid\text{-}line\ l$
by (*induction l rule*: *length-induct*, *subst valid-line-split*, *subst valid-split*, *auto*)

end

Array-Based Queuing Lock

3.1 Challenge

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

We assume that there are `N` threads and we allocate a shared Boolean array `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 $k\mathtt{N}-1$. 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* **by** *auto*
lemma *N-gt0*[*simp*, *intro*!]: *0*<*N* **by** (*cases N*) *auto*

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

datatype *thread* =
INIT
| *is-WAIT*: *WAIT* (*ticket*: *nat*)
| *is-HOLD*: *HOLD* (*ticket*: *nat*)
| *is-REL*: *REL* (*ticket*: *nat*)

3.2.2 Refinement 1: Ticket Lock with Unbounded Counters

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

type-synonym *astate* = *nat* × *nat* × (*nat* ⇒ *thread*)

abbreviation *cc* ≡ *fst*
abbreviation *nn s* ≡ *fst* (*snd s*)
abbreviation *tts s* ≡ *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* **for** *t* **where**
⟦ *t*<*N*; *astep-sng* (*c*,*n*,*ts t*) (*c*′,*n*′,*s*′) ⟧
⟹ *alstep t* (*c*,*n*,*ts*) (*c*′,*n*′,*ts*(*t*:=*s*′))

Initial state of the system

definition as_0 ≡ (*0*, *0*, λ-. *INIT*)

interpretation *A*: *system* as_0 *alstep* **.**

In our system, each thread can always perform a step

lemma *never-blocked*: *A*.*can-step l s* ⟷ *l*<*N*
apply (*cases s*; *cases tts s l*; *simp*)
unfolding *A*.*can-step-def*
apply (*clarsimp simp*: *alstep*.*simps astep-sng*.*simps*; *blast*)+
done

Thus, our system is in particular deadlock free

interpretation *A*: *df-system* as_0 *alstep*
apply *unfold-locales*
subgoal for *s*
using *never-blocked*[*of 0 s*]
unfolding *A*.*can-step-def*
by *auto*
done

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*]:
$\neg$*has-ticket INIT k*
has-ticket (*WAIT k*) $k' \longleftrightarrow k'=k$
has-ticket (*HOLD k*) $k' \longleftrightarrow k'=k$
has-ticket (*REL k*) $k' \longleftrightarrow k'=k$
unfolding *has-ticket-def* **by** *auto*

definition *locks-ticket s k* $\equiv$ *s=HOLD k* $\vee$ *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*) $k' \longleftrightarrow k'=k$
unfolding *locks-ticket-def* **by** *auto*

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

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

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

definition *invar1* $\equiv \lambda(c,n,ts)$.
$c \leq n$
$\wedge$ ($\forall t\ k.\ t<N \wedge$ *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*
apply *rule*
subgoal by (*auto simp*: *invar1-def* as_0*-def*)
subgoal for $s\ s'$
apply (*clarify*)
apply (*erule alstep.cases*)
apply (*erule astep-sng.cases*)
apply (*clarsimp-all simp*: *invar1-def*)
apply *fastforce*
apply *fastforce*
apply *fastforce*

```
  apply fastforce
  by (metis Suc-le-eq holds-imp-uses locks-ticket-def le-neq-implies-less)
done
```

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

```
theorem mutual-exclusion: ⟦A.reachable s;
   t<N; t'<N; t≠t'; is-HOLD (tts s t); is-HOLD (tts s t')
 ⟧ ⟹ False
  apply (cases tts s t; simp)
  apply (cases tts s t'; simp)
  using A.invar-reachable[OF is-invar1, of s]
  apply (auto simp: invar1-def)
  by (metis locks-ticket-simps(3) has-ticket-simps(3))

lemma mutual-exclusion': ⟦A.reachable s;
   t<N; t'<N; t≠t';
   locks-ticket (tts s t) tk; locks-ticket (tts s t') tk'
 ⟧ ⟹ False
  apply (cases tts s t; simp; cases tts s t'; simp)
  apply (cases tts s t'; simp)
  using A.invar-reachable[OF is-invar1, of s]
  apply (clarsimp-all simp: invar1-def)
  unfolding locks-ticket-def has-ticket-def
  apply metis+
  done
```

Fairness: Ordered Lock Acquisition

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

```
lemma fair-aux:
  assumes R: A.is-run s
  assumes A: i<j cc (s i) ≤ k k < cc (s j) t<N tts (s i) t=WAIT k
  shows ∃l. i≤l ∧ l<j ∧ tts (s l) t = HOLD k
proof −
  interpret A: run as₀ alstep s by unfold-locales fact

  from A show ?thesis
  proof (induction j−i arbitrary: i)
    case 0
    then show ?case by auto
  next
    case (Suc i')

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

```
obtain t' where alstep t' (s i) (s (Suc i)) by (rule A.stepE)
then show ?case using Suc.prems
proof cases
 case (1 c n ts c' n' s')
 note [simp] = 1(1,2,3)

 from A.run-invar[OF is-invar1, of i] have invar1 (c,n,ts) by auto
 note I1 = this[unfolded invar1-def, simplified]

 from 1(4) show ?thesis
 proof (cases rule: astep-sng.cases)
  case enter-wait
  then show ?thesis
   using IH Suc.prems apply (auto)
   by (metis 1(2) Suc-leD Suc-lessI fst-conv leD thread.distinct(1))
 next
  case (loop-wait k)
  then show ?thesis
   using IH Suc.prems apply (auto)
   by (metis 1(2) Suc-leD Suc-lessI fst-conv leD)

 next
  case exit-wait
  then show ?thesis
   apply (cases t'=t)
   subgoal
    using Suc.prems apply clarsimp
    by (metis 1(2) Suc-leD Suc-lessI fst-conv fun-upd-same leD
     less-or-eq-imp-le snd-conv)
   subgoal
    using Suc.prems IH
    apply auto
    by (metis 1(2) Suc-leD Suc-lessI fst-conv leD)
   done
 next
  case (start-release k)
  then show ?thesis
   using IH Suc.prems apply (auto)
   by (metis 1(2) Suc-leD Suc-lessI fst-conv leD thread.distinct(7))
 next
  case (release k)
  then show ?thesis
   apply (cases t'=t)
   using I1 IH Suc.prems apply (auto)
   by (metis 1(2) 1(3) Suc-leD Suc-leI Suc-lessI fst-conv
    locks-ticket-simps(4) le-antisym not-less-eq-eq
    has-ticket-simps(2) has-ticket-simps(4))
 qed
qed
```

qed
qed

lemma *s-case-expand*:
(*case s of* (*c*, *n*, *ts*) ⇒ *P c n ts*) = *P* (*cc s*) (*nn s*) (*tts s*)
by (*auto split*: *prod.splits*)

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

lemma *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 −
interpret *A*: *run* as_0 *alstep s* **by** *unfold-locales fact*

from *ACQ WAIT* **have** [*simp*]: *t≠t′ t′≠t* **by** *auto*
from *ACQ* **have** [*simp*]:
nn (*s i*) = *k* ∧ *nn* (*s* (*Suc i*)) = *Suc k*
∧ *cc* (*s* (*Suc i*)) = *cc* (*s i*) ∧ *tts* (*s* (*Suc i*)) = (*tts* (*s i*))(*t*:=*WAIT k*)
apply (*rule-tac A.stepE*[*of i*])
apply (*erule alstep.cases*)
apply (*erule astep-sng.cases*)
by (*auto simp*: *nth-list-update split*: *if-splits*)

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

from *WAIT I1* **have** *k′* < *k* **by** *fastforce*
from *ACQ HOLD* **have** *Suc i* ≠ *j* **by** *auto* **with** *HOLD* **have** *Suc i* < *j* **by** *auto*

have *X1*: *cc* (*s i*) ≤ *k′* **using** *I1 WAIT* **by** *fastforce*
have *X2*: *k′* < *cc* (*s j*)
using *A.run-invar*[*OF is-invar1*, *of j*, *unfolded invar1-def s-case-expand*]
using ‹*k′* < *k*› ‹*t<N*› *HOLD*(*2*)
apply *clarsimp*
by (*metis locks-ticket-simps*(*3*) *has-ticket-simps*(*3*))

from *fair-aux*[*OF RUN* ‹*Suc i* < *j*›, *of k′ t′*, *simplified*] **obtain** *l* **where**
l≥Suc i l<j tts (*s l*) *t′* = *HOLD k′*
using *WAIT X1 X2* **by** *auto*

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

qed

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

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

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

    obtain t′ where alstep t′ (s i) (s (Suc i)) by (rule A.stepE)
    then show ?case
      apply −
      apply (cases t=t′;erule alstep.cases; erule astep-sng.cases)
      apply auto
      using IH Suc.prems Suc.hyps(2)
      apply (auto)
      apply (metis Suc-lessD Suc-lessI fun-upd-same snd-conv)
      apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
      apply (metis Suc.prems(1) Suc-lessD Suc-lessI fun-upd-triv)
      apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
      apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
      apply (metis Suc-lessD Suc-lessI fun-upd-other snd-conv)
      done
  qed
  thus ?thesis using that by blast
qed
```

Finally we can show fairness, which we state as follows: Whenever a thread *t* gets a ticket, all other threads *t′* waiting for the lock will be granted the lock before *t*.

```
theorem fair:
  assumes RUN: A.is-run s
  assumes ACQ: t<N tts (s i) t=INIT is-WAIT (tts (s (Suc i)) t)
    — Thread t calls acquire in step i
  assumes HOLD: i<j is-HOLD (tts (s j) t)
    — Thread t holds lock in step j
  assumes WAIT: t′<N is-WAIT (tts (s i) t′)
    — Thread t′ waits for lock at step i
  obtains l where i<l l<j is-HOLD (tts (s l) t′)
    — Then, t′ gets lock earlier
```

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

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

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

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

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

Liveness

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

definition *invar2*
$\equiv \lambda(c,n,ts).\ \forall k.\ c \leq k \wedge k < n \longrightarrow (\exists t < N.\ has\text{-}ticket\ (ts\ t)\ k)$

lemma *is-invar2*: *A.is-invar invar2*
apply *rule*
subgoal by (*auto simp*: *invar2-def* as_0*-def*)
subgoal for *s* s'
apply (*clarsimp simp*: *invar2-def*)
apply (*erule alstep.cases*; *erule astep-sng.cases*; *clarsimp*)
apply (*metis less-antisym has-ticket-simps*(*1*))
subgoal by (*metis has-ticket-simps*(*2*))
subgoal by (*metis has-ticket-simps*(*2*))
subgoal by (*metis has-ticket-simps*(*3*))
subgoal
apply (*frule A.invar-reachable*[*OF is-invar1*])
unfolding *invar1-def*
apply *clarsimp*
by (*metis Suc-leD locks-ticket-simps*(*4*)
not-less-eq-eq has-ticket-simps(*4*))
done
done

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

corollary *current-lock-used*:

```
  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
  using A.invar-reachable[OF is-invar2 R]
    and A.invar-reachable[OF is-invar1 R] WAIT
  unfolding invar1-def invar2-def apply auto
  by (metis (full-types) le-neq-implies-less not-le order-mono-setup.refl
    has-ticket-simps(2))
```

Used tickets are unique (Corollary from invariant 1)

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

We define the thread that holds a specified ticket

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

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

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

```
definition tweight s ≡
  case s of WAIT - ⇒ 3::nat | HOLD - ⇒ 2 | REL - ⇒ 1 | INIT ⇒ 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 ∃j>i. is-HOLD (tts (s j) t)
proof -
  interpret A: fair-run as0 alstep s by unfold-locales fact

  from A obtain k where Wk: tts (s i) t = WAIT k
```

by (*cases tts* (*s i*) *t*) *auto*

We use the following induction scheme:

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

define *lrel* **where** *lrel* ≡
inv-image (*measure id* <∗*lex*∗> *measure id* <∗*lex*∗> *measure id*) (λ*i*. (
k−*cc* (*s i*),
tweight (*tts* (*s i*) (*tkt-thread* (*tts* (*s i*)) (*cc* (*s i*)))),
A.*dist-step* (*tkt-thread* (*tts* (*s i*)) (*cc* (*s i*))) *i*
))

have *wf lrel* **unfolding** *lrel-def* **by** *auto*
then show *?thesis* **using** *A*(*1*) *Wk*
proof (*induction i*)
case (*less i*)

We name the components of this and the next state

obtain *c n ts* **where** [*simp*]: *s i* = (*c*,*n*,*ts*) **by** (*cases s i*)
from *A*.*run-reachable*[*of i*] **have** *R*: *A*.*reachable* (*c*,*n*,*ts*) **by** *simp*

obtain *c′ n′ ts′* **where** [*simp*]: *s* (*Suc i*) = (*c′*,*n′*,*ts′*)
by (*cases s* (*Suc i*))
from *A*.*run-reachable*[*of Suc i*] **have** *R′*: *A*.*reachable* (*c′*,*n′*,*ts′*)
by *simp*

from *less*.*prems* **have** *WAIT*[*simp*]: *ts t* = *WAIT k* **by** *simp*

{

If thread *t* left waiting state, we are done

assume *ts′ t* ≠ *WAIT k*
hence *ts′ t* = *HOLD k* **using** *less*.*prems*
apply (*rule-tac A*.*stepE*[*of i*])
apply (*auto elim*!: *alstep*.*cases astep-sng*.*cases split*: *if-splits*)
done
hence *?case* **by** *auto*
} moreover {
assume [*simp*]: *ts′ t* = *WAIT k*

Otherwise, we obtain the thread *tt* that holds the current lock

obtain *tt* **where** *UTT*: *tt*<*N has-ticket* (*ts tt*) *c*
using *current-lock-used*[*of c n ts t k*]

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

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

have *A.can-step tt* (*s i*) **by** (*simp add*: *never-blocked*)
hence *?case* **proof** (*cases rule*: *A.rstep-cases*)
case (*other t*′) — Another thread than *tt* makes a step.

The current ticket and *tt*'s state remain the same

hence [*simp*]: *c*′ = *c* ∧ *ts*′ *tt* = *ts tt*
using *has-ticket-unique*[*OF R UTT*, *of t*′]
unfolding *A.rstep-def*
using *holds-only-current*[*OF R*, *of t*′]
by (*force elim*!: *alstep.cases astep-sng.cases*)

Thus, *tt* is still using the current ticket

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

Only the distance to *tt*'s next step has decreased

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

And we can apply the induction hypothesis

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

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

Thus, current increases

have [*simp*]: *c*′ = *Suc c*
using *THIS* **apply** −
unfolding *A.rstep-def*
apply (*erule alstep.cases*, *erule astep-sng.cases*)
by *auto*

But is still less than *k*

from *A.invar-reachable*[*OF is-invar1 R*] **have** *k*>*c*
apply (*auto simp*: *invar1-def*)

by (*metis UTT WAIT* ‹*ts tt* = *REL c*› *le-neq-implies-less less.prems*(*1*) *thread.distinct*(*9*) *has-ticket-simps*(*2*))

And we can apply the induction hypothesis

hence (*Suc i*, *i*) ∈ *lrel*
unfolding *lrel-def* **by** *auto*
with *less.IH*[*of Suc i*] ‹*t*<*N*› **show** *?thesis*
apply (*auto*) **using** *Suc-lessD* **by** *blast*
next
case *False* — *tt* has acquired the lock, or started releasing it

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

hence [*simp*]:
$c' = c$
∧ *tweight* (*ts tt*) > *tweight* (*ts′ tt*)
∧ *has-ticket* (*ts′ tt*) *c*
using *THIS UTT* **apply** −
unfolding *A.rstep-def*
apply (*erule alstep.cases*, *erule astep-sng.cases*)
by (*auto simp*: *has-ticket-def tweight-def*)

tt still holds the current lock

have [*simp*]: *tkt-thread ts′ c* = *tt*
using *tkt-thread-eq*[*OF R′* ‹*tt*<*N*›, *of c*] **by** *simp*

And we can apply the IH

have (*Suc i*, *i*) ∈ *lrel* **unfolding** *lrel-def* **by** *auto*
with *less.IH*[*of Suc i*] ‹*t*<*N*› **show** *?thesis*
apply (*auto*) **using** *Suc-lessD* **by** *blast*
qed
qed
}
ultimately show *?case* **by** *blast*
qed
qed

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*≠*0* **by** *auto*
lemma *k-gt0*[*simp*]: *0*<*k* **by** (*cases k*) *auto*

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

type-synonym *bstate* = *nat* × *nat* × (*nat* ⇒ *thread*)

The step relation of a single thread

```
inductive bstep-sng where
  enter-wait: bstep-sng (c,n,INIT) (c,(n+1) mod (k∗N),WAIT (n mod N))
| loop-wait: c≠tk ⟹ bstep-sng (c,n,WAIT tk) (c,n,WAIT tk)
| exit-wait: bstep-sng (c,n,WAIT c) (c,n,HOLD c)
| start-release: bstep-sng (c,n,HOLD tk) (c,n,REL tk)
| release: bstep-sng (c,n,REL tk) ((tk+1) mod N,n,INIT)
```

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

```
inductive blstep for t where
  ⟦ t<N; bstep-sng (c,n,ts t) (c′,n′,s′) ⟧
    ⟹ blstep t (c,n,ts) (c′,n′,ts(t:=s′))
```

Initial state of the system

```
definition bs0 ≡ (0, 0, λ-. INIT)

interpretation B: system bs0 blstep .

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

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

Simulation

We show that the abstract system simulates the concrete one.

A few lemmas to ease the automation further below

```
lemma nat-sum-gtZ-iff[simp]:
  finite s ⟹ sum f s ≠ (0::nat) ⟷ (∃x∈s. f x ≠ 0)
  by simp

lemma n-eq-Suc-sub1-conv[simp]: n = Suc (n − Suc 0) ⟷ n≠0 by auto

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

lemma mod-eq-imp-eq-aux: b mod N = (a::nat) mod N ⟹ a≤b ⟹ b<a+N ⟹ b=a
  by (auto simp add: mod-eq-dvd-iff-nat le-imp-diff-is-add)
```

lemma *mod-eq-imp-eq*:
$[\![b \leq x; x < b + N; b \leq y; y < b + N; x \bmod N = y \bmod N]\!] \Longrightarrow x=y$
proof –
assume *a1*: $b \leq y$
assume *a2*: $y < b + N$
assume *a3*: $b \leq x$
assume *a4*: $x < b + N$
assume *a5*: $x \bmod N = y \bmod N$
have *f6*: $x < y + N$
using *a4 a1* **by** *linarith*
have $y < x + N$
using *a3 a2* **by** *linarith*
then show *?thesis*
using *f6 a5* **by** (*metis* (*no-types*) *mod-eq-imp-eq-aux nat-le-linear*)
qed

Map the ticket of a thread

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

lemma *map-ticket-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 $\longleftrightarrow$ $(\exists tk'.\ tk{=}f\ tk' \wedge t{=}REL\ tk')$
by (*cases t*; *auto*)+

We define the number of threads that use a ticket

fun *ni-weight* :: *thread* $\Rightarrow$ *nat* **where**
ni-weight INIT = 0 | *ni-weight - = 1*

lemma *ni-weight-le1*[*simp*]: *ni-weight s* $\leq$ *Suc 0*
by (*cases s*) *auto*

definition *num-ni ts* $\equiv \sum$ *i=0..<N. ni-weight* (*ts i*)
lemma *num-ni-init*[*simp*]: *num-ni* (λ-. *INIT*) = *0* **by** (*auto simp*: *num-ni-def*)

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

lemma *num-ni-nz-if*[*simp*]: $[\![t < N; ts\ t \neq INIT]\!] \Longrightarrow$ *num-ni ts* $\neq$ *0*
apply (*cases ts t*)

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

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

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

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

lemma is-invar3: A.is-invar invar3
  apply (rule)
  subgoal by (auto simp: invar3-def as₀-def)
  subgoal for s s'
    apply clarify
    apply (erule alstep.cases, erule astep-sng.cases)
    apply (auto simp: invar3-def num-ni-upd)
    using holds-only-current by fastforce
  done
```

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

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


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

lemma sraux2: ⟦sim-rel1 (c,n,ts) (ci,ni,tsi); t<N⟧
  ⟹ tsi t = map-ticket (λx. x mod N) (ts t)
  by (auto simp: sim-rel1-def Let-def)


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

```
  and SIM: sim-rel1 as bs
  and CS: blstep t bs bs′

obtain c n ts where [simp]: as=(c,n,ts) by (cases as)
obtain ci ni tsi where [simp]: bs=(ci,ni,tsi) by (cases bs)
obtain ci′ ni′ tsi′ where [simp]: bs′=(ci′,ni′,tsi′) by (cases bs′)
from Ra-aux have Ra: A.reachable (c,n,ts) by simp
from Rc-aux have Rc: B.reachable (ci,ni,tsi) by simp

from CS have t<N by cases auto

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

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

from SIM CS have ∃as′. alstep t as as′ ∧ sim-rel1 as′ bs′
 apply simp
 apply (erule blstep.cases)
 apply (erule bstep-sng.cases)
 apply clarsimp-all
subgoal
 apply (intro exI conjI)
 apply (rule alstep.intros)
 apply (simp add: sim-rel1-def Let-def)
 apply (simp add: sraux sraux2)
 apply (rule astep-sng.enter-wait)
 apply (simp add: sim-rel1-def; intro conjI ext)
 apply (auto simp: sim-rel1-def Let-def mod-simps)
 done
subgoal
 apply (clarsimp simp: sraux sraux2)
 apply (intro exI conjI)
 apply (rule alstep.intros)
 apply (simp add: sim-rel1-def Let-def)
 apply clarsimp
 apply (rule astep-sng.loop-wait)
 apply (auto simp: sim-rel1-def Let-def mod-simps)
 done
subgoal
 apply (clarsimp simp: sraux sraux2)
 subgoal for tk′
  apply (subgoal-tac tk′=c)
  apply (intro exI conjI)
  apply (rule alstep.intros)
```

```
      apply (simp add: sim-rel1-def Let-def)
      apply clarsimp
      apply (rule astep-sng.exit-wait)
      apply (auto simp: sim-rel1-def Let-def mod-simps) []
      apply (clarsimp simp: sim-rel1-def)
      apply (erule mod-eq-imp-eq-aux)
      apply (auto simp: AUX1)
      done
    done
  subgoal
    apply (clarsimp simp: sraux sraux2)
    apply (intro exI conjI)
    apply (rule alstep.intros)
    apply (simp add: sim-rel1-def Let-def)
    apply clarsimp
    apply (rule astep-sng.start-release)
    apply (auto simp: sim-rel1-def Let-def mod-simps)
    done
  subgoal
    apply (clarsimp simp: sraux sraux2)
    apply (intro exI conjI)
    apply (rule alstep.intros)
    apply (simp add: sim-rel1-def Let-def)
    apply clarsimp
    apply (rule astep-sng.release)
    apply (auto simp: sim-rel1-def Let-def mod-simps)
    done
  done
  then show ∃as'. sim-rel1 as' bs' ∧ alstep t as as' by blast
next
  fix as bs l
  assume A.reachable as B.reachable bs sim-rel1 as bs A.can-step l as
  then show B.can-step l bs using b-never-blocked never-blocked by simp
qed
```

Transfer of Properties

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

```
lemma xfer-locks-ticket:
  assumes locks-ticket (map-ticket (λt. t mod N) (ts t)) tki
  obtains tk where tki=tk mod N locks-ticket (ts t) tk
  using assms unfolding locks-ticket-def
  by auto


lemma b-holds-only-current:
  ⟦B.reachable (c, n, ts); t < N; locks-ticket (ts t) tk⟧ ⟹ tk = c
  apply (rule sim1.xfer-reachable, assumption)
```

```
  apply (clarsimp simp: sim-rel1-def)
  apply (erule xfer-locks-ticket)+
  using holds-only-current
  by blast

lemma b-mutual-exclusion′: ⟦B.reachable s;
    t<N; t′<N; t≠t′; locks-ticket (tts s t) tk; locks-ticket (tts s t′) tk′
  ⟧ ⟹ False
  apply (rule sim1.xfer-reachable, assumption)
  apply (clarsimp simp: sim-rel1-def)
  apply (erule xfer-locks-ticket)+
  apply (drule (3) mutual-exclusion′; simp)
  done

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

lemma has-ticket-in-range:
  assumes Ra: A.reachable (c,n,ts) and t<N and U: has-ticket (ts t) tk
  shows c≤tk ∧ tk<c+N
proof −

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

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

lemma b-has-ticket-unique: ⟦B.reachable (ci,ni,tsi);
    t<N; has-ticket (tsi t) tki; t′<N; has-ticket (tsi t′) tki
  ⟧ ⟹ t′=t
  apply (rule sim1.xfer-reachable, assumption)
  apply (auto simp: sim-rel1-def)
  subgoal for c n ts
    apply (erule xfer-has-ticket)+
    apply simp
    subgoal for tk tk′
      apply (subgoal-tac tk=tk′)
      apply simp
      apply (frule (4) has-ticket-unique, assumption)
      apply (frule (2) has-ticket-in-range[where tk=tk])
      apply (frule (2) has-ticket-in-range[where tk=tk′])
```

```
      apply (auto simp: mod-simps)
      apply (rule mod-eq-imp-eq; (assumption|simp))
      done
    done
  done
```

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* **.**

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

interpretation *C*: *df-system* cs_0 *clstep*
  apply *unfold-locales*
  subgoal for *s*
    using *c-never-blocked*[*of 0 s*]
    unfolding *C*.*can-step-def*
    by *auto*
  done

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

```
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
  apply (rule)
  subgoal by (auto simp: invar4-def bs₀-def)
  subgoal for s s'
    apply clarify
    apply (erule blstep.cases, erule bstep-sng.cases)
    unfolding invar4-def
        apply safe
    apply (metis N-gt0 fun-upd-apply has-ticket-simps(2) mod-less-divisor)
      apply (metis fun-upd-triv)
      apply (metis fun-upd-other fun-upd-same has-ticket-simps(3))
      apply (metis fun-upd-other fun-upd-same has-ticket-def has-ticket-simps(4))
      using mod-less-divisor apply blast
      apply (metis fun-upd-apply thread.distinct(1) thread.distinct(3)
        thread.distinct(5) has-ticket-def)
    done
  done
```

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

```
definition is-REL-state ≡ λts. ∃t<N. ∃tk. ts t = REL tk

lemma is-REL-state-simps[simp]:
  t<N ⟹ is-REL-state (ts(t:=REL tk))
  t<N ⟹ ¬is-REL (ts t) ⟹ ¬is-REL s'
    ⟹ is-REL-state (ts(t:=s')) ⟷ is-REL-state ts
  unfolding is-REL-state-def
  apply (auto;fail) []
  apply auto []
  by (metis thread.disc(12))

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

lemma is-REL-state-aux2:
```

```
  assumes R: B.reachable (c,n,ts)
  assumes A: t<N ts t = REL tk
  shows ¬is-REL-state (ts(t:=INIT))
  using b-holds-only-current[OF R] A
  using b-mutual-exclusion'[OF R]
  apply (clarsimp simp: is-REL-state-def)
  by fastforce
```

Simulation relation that implements current ticket by array

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


interpretation sim2: simulationI bs0 blstep cs0 clstep sim-rel2
proof unfold-locales
  show sim-rel2 bs0 cs0
    by (auto simp: sim-rel2-def bs0-def cs0-def is-REL-state-def)
next
  fix bs cs t cs'
  assume Rc-aux: B.reachable bs
     and Rd-aux: C.reachable cs
     and SIM: sim-rel2 bs cs
     and CS: clstep t cs cs'

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

  from CS have t<N by cases auto

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

  have [simp]: c<N using B.invar-reachable[OF is-invar4 Rc]
    by (auto simp: invar4-def)
```

```
have [simp]:
  replicate N False ≠ (replicate N False)[c := True]
  (replicate N False)[c := True] ≠ replicate N False
  apply (auto simp: list-eq-iff-nth-eq nth-list-update)
  using ‹c < N› by blast+

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

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

from SIM CS have ∃bs'. blstep t bs bs' ∧ sim-rel2 bs' cs'
  apply simp
  apply (subst (asm) sim-rel2-def)
  apply (erule clstep.cases)
  apply (erule cstep-sng.cases)
  apply clarsimp-all
  subgoal
    apply (intro exI conjI)
    apply (rule blstep.intros)
    apply (simp)
    apply clarsimp
    apply (rule bstep-sng.enter-wait)
    apply (auto simp: sim-rel2-def split: if-splits)
    done
  subgoal for tk'
    apply (intro exI conjI)
    apply (rule blstep.intros)
    apply (simp)
    apply clarsimp
    apply (rule bstep-sng.loop-wait)
    subgoal
      apply (clarsimp simp: sim-rel2-def split: if-splits)
      apply (frule (2) is-REL-state-aux1[OF Rc])
      by simp
    subgoal by (auto simp: sim-rel2-def split: if-splits)
    done
  subgoal
    apply (intro exI conjI)
    apply (rule blstep.intros)
    apply (simp)
    apply (clarsimp split: if-splits)
    apply (rule bstep-sng.exit-wait)
    apply (auto simp: sim-rel2-def split: if-splits)
    done
  subgoal
```

apply (*intro exI conjI*)
apply (*rule blstep.intros*)
apply (*simp*)
apply *clarsimp*
apply (*rule bstep-sng.start-release*)
apply (*auto simp*: *sim-rel2-def dest*: *HOLD-AUX split*: *if-splits*)
done
subgoal
apply (*intro exI conjI*)
apply (*rule blstep.intros*)
apply (*simp*)
apply *clarsimp*
apply (*rule bstep-sng.release*)
apply (*auto*
simp: *sim-rel2-def*
dest: *is-REL-state-aux2*[*OF Rc*]
split: *if-splits*)
by (*metis fun-upd-triv is-REL-state-simps*(*1*))
done
then show $\exists bs'$. *sim-rel2* bs' cs' $\wedge$ *blstep t bs* bs' **by** *blast*

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

3.2.5 Transfer Setup

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

definition *sim-rel* $\equiv$ *sim-rel1 OO sim-rel2*
interpretation *sim*: *simulation* as_0 *alstep* cs_0 *clstep sim-rel*
unfolding *sim-rel-def*
by (*rule sim-trans*) *unfold-locales*

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

lemma *xfer-waits*:
assumes *sim-rel s cs*
shows *is-WAIT* (*tts cs t*) $\longleftrightarrow$ *is-WAIT* (*tts s t*)
using *assms* **unfolding** *sim-rel-def sim-rel1-def sim-rel2-def*
by (*cases tts cs t*) *auto*

lemma *xfer-init*:
assumes *sim-rel s cs*
shows $tts\ cs\ t = INIT \longleftrightarrow tts\ s\ t = INIT$
using *assms* **unfolding** *sim-rel-def sim-rel1-def sim-rel2-def*
by *auto*

3.2.6 Main Theorems

Trusted Code Base

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

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

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

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

Safety

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

theorem *final-mutual-exclusion*: $[\![C.reachable\ s;$
$t{<}N;\ t'{<}N;\ t{\neq}t';\ is\text{-}HOLD\ (tts\ s\ t);\ is\text{-}HOLD\ (tts\ s\ t')$
$]\!] \Longrightarrow False$
apply (*erule sim.xfer-reachable*)
apply (*simp add*: *xfer-holds*)
by (*erule* (5) *mutual-exclusion*)

Fairness

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

theorem *final-fair*:
assumes *RUN*: *C.is-run s*
assumes *ACQ*: $t{<}N$ **and** $tts\ (s\ i)\ t{=}INIT$ **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
using *RUN*
proof (*rule sim.xfer-run*)
fix *as*
assume *Ra*: *A.is-run as* **and** *SIM*[*rule-format*]: ∀*i*. *sim-rel* (*as i*) (*s i*)

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

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

Liveness

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

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

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

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

end