Verifying "Simple" Persistent Catenable Deques

Journées Francophones des Langages Applicatifs 2026

Juliette Ponsonnet[1] François Pottier[2]

[1]ENS de Lyon

[2]Inria Paris

9 Pluviôse an CCXXXIV

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

Kaplan, Okasaki and Tarjan's Deques

- catenable deque
- self adjusting data structure
- persistent (old copies cannot be invalidated)
- use references internally
- not so simple
- **all** operations in amortised $O(1)$

Figure: Кот, не очередь

An Unusual Algorithm

Kaplan, Okasaki, Tarjan (2000)

our method requires an extension of the concept of memoization: we allow any expression to be replaced by an equivalent expression

A Surprising Journey

What we ended up doing:

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"
- Prove concurrent correctness

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"
- Prove concurrent correctness
- Formalise sequential "stable references"

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"
- Prove concurrent correctness
- Formalise sequential "stable references"
- Point out a flaw in the original proof

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"
- Prove concurrent correctness
- Formalise sequential "stable references"
- Point out a flaw in the original proof
- Changing one line of code, fix the invariants

A Surprising Journey

What we set out to do:

- Understand and formalise the extension of memoisation
- Prove the correctness of the data structure
- Prove $O(1)$ amortised time complexity

What we ended up doing:

- Changing one line of code, make the structure concurrent
- Formalise concurrent "stable references"
- Prove concurrent correctness
- Formalise sequential "stable references"
- Point out a flaw in the original proof
- Changing one line of code, fix the invariants
- Prove $O(1)$ *sequential* amortised complexity

You are here

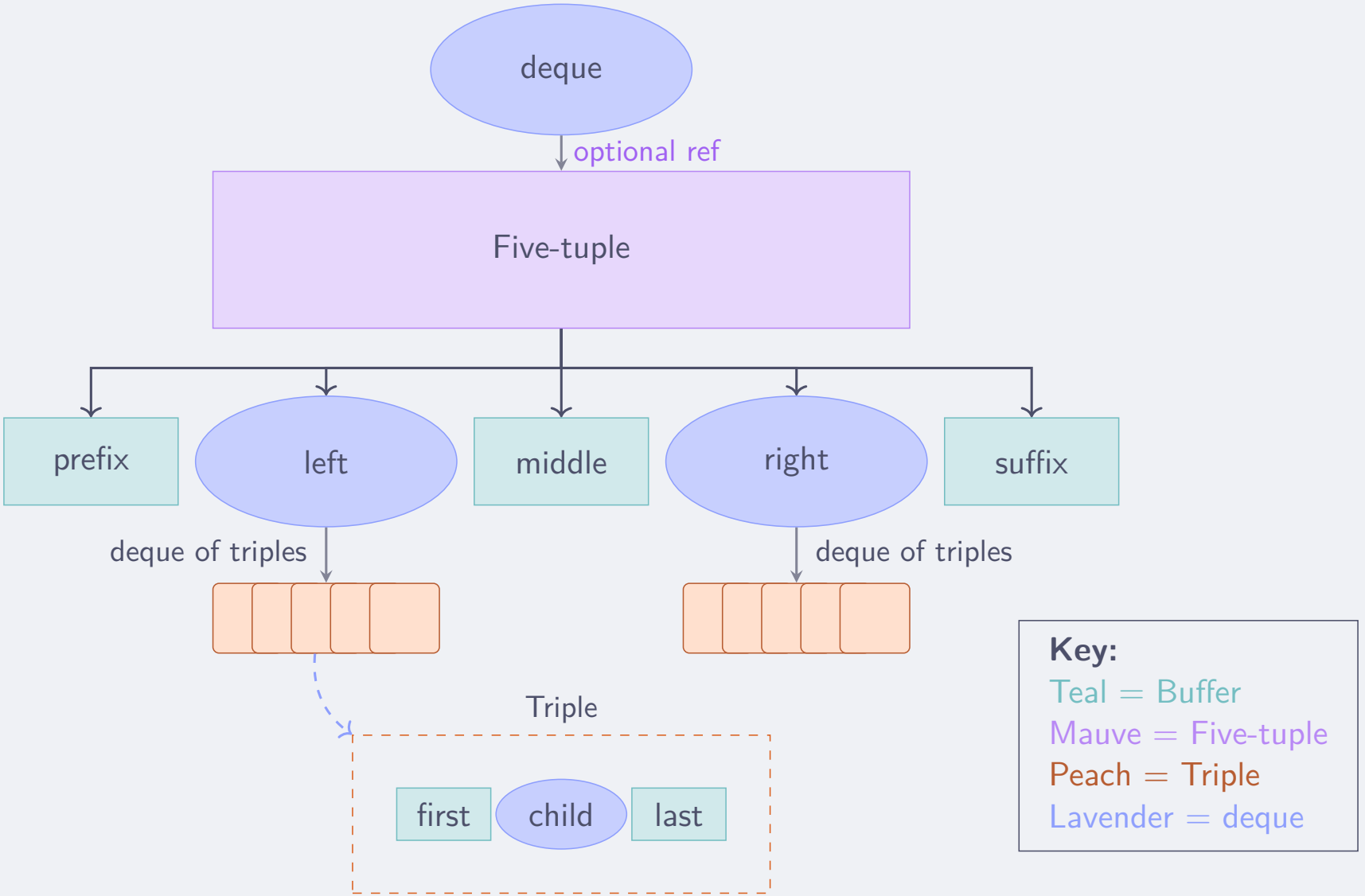
1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

OCaml Interface[1]

```
type 'a deque
val empty : 'a deque
val push  : 'a -> 'a deque -> 'a deque
val inject: 'a deque -> 'a -> 'a deque
val pop   : 'a deque -> 'a * 'a deque
val eject : 'a deque -> 'a deque * 'a
val concat: 'a deque -> 'a deque -> 'a deque
```

[1] opam install kot

Structure of a Deque



OCaml Type Definition

```
type 'a deque =
  'a nonempty_deque option
and 'a nonempty_deque =
  'a five_tuple ref
and 'a five_tuple = {
  prefix : 'a buffer;
  left   : 'a triple deque;
  middle : 'a buffer;
  right  : 'a triple deque;
  suffix : 'a buffer;
}
and 'a triple = {
  first  : 'a buffer;
  child  : 'a triple deque;
  last   : 'a buffer;
}
```

Pop is not so “Simple”

Pop is not so “Simple”

```
let naive_pop_safe (type a) (f : a five_tuple) : bool =
  let { prefix; middle; _ } = f in
  B.is_empty middle || B.length prefix > 3
```

Pop is not so “Simple”

```
let naive_pop_safe (type a) (f : a five_tuple) : bool =
  let { prefix; middle; _ } = f in
  B.is_empty middle || B.length prefix > 3

let naive_pop (type a) (f : a five_tuple) : a * a deque =
  (* omitted; just 7 lines *)
```

Pop is not so “Simple”

```
let naive_pop_safe (type a) (f : a five_tuple) : bool =
  let { prefix; middle; _ } = f in
  B.is_empty middle || B.length prefix > 3

let naive_pop (type a) (f : a five_tuple) : a * a deque =
  (* omitted; just 7 lines *)

let rec pop_nonempty : type a. a nonempty_deque -> a * a deque = fun r ->
  let f = !r in
  if naive_pop_safe f then
    naive_pop f
  else
    let f = prepare_pop f in
    r := f;
    assert (naive_pop_safe f);
    naive_pop f
```

Pop is not so “Simple”

```
let naive_pop_safe (type a) (f : a five_tuple) : bool =
  let { prefix; middle; _ } = f in
  B.is_empty middle || B.length prefix > 3

let naive_pop (type a) (f : a five_tuple) : a * a deque =
  (* omitted; just 7 lines *)

let rec pop_nonempty : type a. a nonempty_deque -> a * a deque = fun r ->
  let f = !r in
  if naive_pop_safe f then
    naive_pop f
  else
    let f = prepare_pop f in
    r := f;
    assert (naive_pop_safe f);
    naive_pop f

and prepare_pop : type a. a five_tuple -> a five_tuple = fun f ->
  (* omitted; about 100 lines; uses [pop_triple] *)
```

Foreshadowing

```
and pop_triple
 : type a. a triple nonempty_deque
         -> a triple * a triple deque
 = fun r ->
  let f = !r in
  let t = inspect_first f in
  let { first; last; _ } = t in
  (* The (repaired) mysterious condition: *)
  if not (B.is_empty last) || B.has_length_3 first then
    (* [naive_pop_safe f] is not necessarily true here! *)
    naive_pop f
  else
    pop_nonempty r
```

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

Specification

Deque d xs : *d* is a deque *and* represents the list *xs*

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

Specification

Deque d xs : *d* is a deque *and* represents the list *xs*

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

DEQUE-PERSIST
persistent(*Deque d xs*)

DEQUE-EMPTY
Deque empty []

Specification

Deque d xs : *d* is a deque *and* represents the list *xs*

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

DEQUE-PERSIST
$\mathit{persistent}(\mathit{Deque}\ d\ xs)$

DEQUE-EMPTY
$\mathit{Deque}\ \mathit{empty}\ []$

DEQUE-PUSH
$\{\mathit{Deque}\ d\ xs\}\ \mathit{push}\ x\ d\ (\exists d')\ d'\ \{\mathit{Deque}\ d'\ ([x] +\!\!+ xs)\}$

Specification

Deque d xs : *d* is a deque *and* represents the list *xs*

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

DEQUE-PERSIST
$\mathit{persistent}(\mathit{Deque}\ d\ xs)$

DEQUE-EMPTY
$\mathit{Deque}\ \mathit{empty}\ []$

DEQUE-PUSH
$\{\mathit{Deque}\ d\ xs\}\ \mathit{push}\ x\ d\ (\exists d')\ d'\ \{\mathit{Deque}\ d'\ ([x] +\!\!+ xs)\}$

DEQUE-POP
$\{\mathit{Deque}\ d\ ([x] +\!\!+ xs)\}\ \mathit{pop}\ d\ (\exists d')\ (x, d')\ \{\mathit{Deque}\ d'\ xs\}$

DEQUE-CONCAT
$$\{\mathit{Deque}\ d\ xs\ *\ \mathit{Deque}\ d'\ xs'\}\ \mathit{concat}\ d\ d'\ (\exists d'')\ d''\ \{\mathit{Deque}\ d''\ (xs +\!\!+ xs')\}$$

Specification

Deque d xs : *d* is a deque *and* represents the list *xs*

$$\mathit{Deque} : \mathit{Val} \to \mathit{list}\ \mathit{Val} \to \mathit{iProp}$$

DEQUE-PERSIST
$\mathit{persistent}(\mathit{Deque}\ d\ xs)$

DEQUE-EMPTY
$\mathit{Deque}\ \mathit{empty}\ []$

DEQUE-PUSH
$\{\mathit{Deque}\ d\ xs\}\ \mathit{push}\ x\ d\ (\exists d')\ d'\ \{\mathit{Deque}\ d'\ ([x] +\!\!+ xs)\}$

DEQUE-POP
$\{\mathit{Deque}\ d\ ([x] +\!\!+ xs)\}\ \mathit{pop}\ d\ (\exists d')\ (x, d')\ \{\mathit{Deque}\ d'\ xs\}$

DEQUE-CONCAT
$$\{\mathit{Deque}\ d\ xs\ *\ \mathit{Deque}\ d'\ xs'\}\ \mathit{concat}\ d\ d'\ (\exists d'')\ d''\ \{\mathit{Deque}\ d''\ (xs +\!\!+ xs')\}$$

This specification makes deques impossible to invalidate, concurrent and shareable.

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

Designing the Deque Predicate

How to encode this reference to make the deque predicate persistent?

```
and 'a nonempty_deque =
  'a five_tuple ref
```

Designing the Deque Predicate

How to encode this reference to make the deque predicate persistent? Recall: *We allow ft to be overwritten with a value ft′ provided ft and ft′ are equivalent in some sense.*

```
and 'a nonempty_deque =
  'a five_tuple ref
```

Stable References – Intuition

Standard Iris

$$\ell \mapsto v$$

too precise, exclusive ownership

- Using points-to, restricting writes with a property
- Reference to a property, value unknown

"Stable Reference" library

$$\ell \Mapsto \phi$$

invariant and shareable (everyone agrees on ϕ)

(Concurrent) Stable References (CSRef)

CSRef-Persist
$\mathit{persistent}(\ell \Rrightarrow \phi)$

CSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \Rrightarrow \phi\}$

CSRef-Read
$\mathit{persistent}(\phi) \twoheadrightarrow$
$\{\ell \Rrightarrow \phi\}\ !\ell\ (\exists v)\ v\ \{\phi\ v\}$

CSRef-Write
$\{\ell \Rrightarrow \phi\ *\ \phi\ v\}\ \ell := v\ ()\ \{\}$

(Concurrent) Stable References (CSRef)

CSRef-Persist
$\mathit{persistent}(\ell \Mapsto \phi)$

CSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \Mapsto \phi\}$

CSRef-Read
$\mathit{persistent}(\phi) \twoheadrightarrow$
$\{\ell \Mapsto \phi\}\ !\ell\ (\exists v)\ v\ \{\phi\ v\}$

CSRef-Write
$\{\ell \Mapsto \phi\ *\ \phi\ v\}\ \ell := v\ ()\ \{\}$

We also propose a sequential interface (SSRef), discussed later.

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

Defining the Deque Predicate

$$\begin{aligned} \mathit{Deque}\ d\ \mathit{xs} \triangleq\ & \ulcorner d = \mathit{None} \wedge \mathit{xs} = [] \urcorner \vee \\ & \exists \ell.\ \ulcorner d = \mathit{Some}(\ell) \urcorner * \\ & \ell \Mapsto (\lambda \mathit{ft}.\ \mathit{fiveTuple}\ n\ \mathit{ft}\ \mathit{xs}) \end{aligned}$$

$$\begin{aligned} \mathit{fiveTuple}\ n\ \mathit{ft}\ \mathit{xs} \triangleq\ & \exists\ p, l, m, r, s, \mathit{xs}_p, \mathit{xss}_l, \mathit{xs}_m, \mathit{xss}_r, \mathit{xs}_s. \\ & \ulcorner d = (p, l, m, r, s) \wedge \\ & \mathit{config}_5(|\mathit{xs}_p|, |\mathit{xss}_l|, |\mathit{xs}_m|, |\mathit{xss}_r|, |\mathit{xss}_s|) \wedge \\ & \mathit{xs} = \mathit{xs}_p +\!\!+ \mathit{join}(\mathit{xss}_l) +\!\!+ \mathit{xs}_m +\!\!+ \mathit{join}(\mathit{xss}_r) +\!\!+ \mathit{xs}_s \urcorner * \\ & \mathit{buffer}\ p\ \mathit{xss}_p * \\ & \mathit{dequeOfTriples}\ l\ \mathit{xs}_l * \\ & \mathit{buffer}\ m\ \mathit{xs}_m * \\ & \mathit{dequeOfTriples}\ r\ \mathit{xss}_r * \\ & \mathit{buffer}\ s\ \mathit{xs}_s \end{aligned}$$

Proof Excerpt – Creating Stable References

Initial Goal

```
"Hb" : buffer (1) [x] b
------------------------------------□
...
------------------------------------*
WP ref (bempty, InjLV #(), bempty, InjLV #(), b)
   {{ v, WP InjR v {{ v, ψ v }} }}
```

Proof Excerpt – Creating Stable References

Initial Goal

```
"Hb" : buffer (1) [x] b
------------------------------------□
...
------------------------------------*
WP ref (bempty, InjLV #(), bempty, InjLV #(), b)
   {{ v, WP InjR v {{ v, ψ v }} }}
```

```
wp_apply (csref_alloc (fiveTuple [x])) as "%ℓ #Hℓ".
```

Proof Excerpt – Creating Stable References

Initial Goal

```
"Hb" : buffer (1) [x] b
----------------------------------------□
...
----------------------------------------*
WP ref (bempty, InjLV #(), bempty, InjLV #(), b)
   {{ v, WP InjR v {{ v, ψ v }} }}
```

```
wp_apply (csref_alloc (fiveTuple [x])) as "%ℓ #Hℓ".
```

Subgoals

1. **Prove the initial value is correct:**

   ```
   "Hb" : buffer 1 [x] b
   ----------------------------------------□
   fiveTuple [x] (bempty, InjLV #(), bempty, InjLV #(), b)
   ```

2. **Continue the proof with the new reference ℓ:**

   ```
   "Hℓ" : ℓ ⤇ fiveTuple [x]
   ----------------------------------------□
   ...
   ----------------------------------------*
   WP InjR #ℓ {{ v, ψ v }}
   ```

Proving the Specification

- KOT did not provide a correctness proof
- Correctness of the operations is relatively self-evident
- Showing the invariants are upheld requires a more careful look

Pop is not “Simple” at all

Recall:

```
and pop_triple
 : type a. a triple nonempty_deque
         -> a triple * a triple deque
 = fun r ->
  let f = !r in
  let t = inspect_first f in
  let { first; last; _ } = t in
  (* The (repaired) mysterious condition: *)
  if not (B.is_empty last) || B.has_length_3 first then
    (* [naive_pop_safe f] is not necessarily true here! *)
    naive_pop f
  else
    pop_nonempty r
```

Specifying `pop_triple` – Naïve-Pop

NaivePop-Safe

$$\{\mathit{safeFiveTuple}\ n\ ft\ ([x] +\!\!+ xs)\}$$
$$\mathit{naive\_pop}\ ft$$
$$(\exists d)\ (x,\ d)\ \{\mathit{deque}\ n\ d\ xs\}$$

Specifying `pop_triple` – Naïve-Pop

$$\textsc{NaivePop-Safe}\quad \{\mathit{safeFiveTuple}\ n\ ft\ ([x] +\!\!+ xs)\}\ \mathit{naive\_pop}\ ft\ (\exists d)\ (x,\ d)\ \{\mathit{deque}\ n\ d\ xs\}$$

$$\textsc{NaivePop-Unsafe}\quad \{\mathit{fiveTuple}\ n\ ft\ ([x] +\!\!+ xs)\}\ \mathit{naive\_pop}\ ft\ (\exists d)\ (x,\ d)\ \left\{\begin{array}{c}\mathsf{wp}\\ \forall y,\ \mathit{push}\ y\ d\\ (\exists d')\ d'\ \{\mathit{deque}\ n\ d'\ ([y] +\!\!+ xs)\}\end{array}\right\}$$

Specifying `pop_triple` – The Mysterious Condition

The "mysterious condition" seen before is used to decide which path is executed in `pop_triple`. Depending on whether the triple popped is "special", the post-condition is different.

Specifying `pop_triple` – The Mysterious Condition

The "mysterious condition" seen before is used to decide which path is executed in `pop_triple`. Depending on whether the triple popped is "special", the post-condition is different.

PopTriple

$$\{\mathit{dequeOfTriples}\ n\ d\ ([t] +\!\!+ ts)\}$$

$$\mathit{pop\_triple}\ d$$

$$(\exists d')\ (t,\ d')\ \left\{ \begin{array}{c} \mathit{nonSpecialTriple}\ n\ xs\ t\ *\ \mathit{deque}\ n\ d'\ ts \\ \bigvee \\ \mathit{specialTriple}\ n\ xs\ t\ * \\ \forall t',\ \mathsf{wp}\ \mathit{push}\ t'\ d'\ (\exists d'')\ d''\ \{\mathit{deque}\ n\ d''\ ([t'] +\!\!+ ts)\} \end{array} \right\}$$

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

New Specification

$$\mathit{persistent}(\mathit{Deque}_\$\ \pi\ d\ xs) \qquad\qquad \mathit{Deque}_\$\ \pi\ \mathit{empty}\ []$$

$$\left\{ \mathit{Deque}_\$\ \pi\ d\ xs\ *\ \lightning^{\pi}\ *\ \$7 \right\}$$

$$\mathit{push}\ x\ d$$

$$(\exists d')\ d'\ \left\{ \mathit{Deque}_\$\ \pi\ d'\ ([x] +\!\!+ xs)\ *\ \lightning^{\pi} \right\}$$

$$\left\{ \mathit{Deque}_\$\ \pi\ d\ ([x] +\!\!+ xs)\ *\ \lightning^{\pi}\ *\ \$171 \right\}$$

$$\mathit{pop}\ d$$

$$(\exists d')\ (x, d')\ \left\{ \mathit{Deque}_\$\ \pi\ d'\ xs\ *\ \lightning^{\pi} \right\}$$

$$\left\{ \mathit{Deque}_\$\ \pi\ d\ xs\ *\ \mathit{Deque}_\$\ \pi\ d'\ xs'\ *\ \lightning^{\pi}\ *\ \$57 \right\}$$

$$\mathit{concat}\ d\ d'$$

$$(\exists d'')\ d''\ \left\{ \mathit{Deque}_\$\ \pi\ d''\ (xs +\!\!+ xs')\ *\ \lightning^{\pi} \right\}$$

New Deque Predicate

$$\mathit{Deque}_{\$}\ \pi\ d\ xs \triangleq \mathit{deque}_{\$}\ \pi\ 0\ d\ xs$$

$$\begin{aligned}\mathit{deque}_{\$}\ \pi\ n\ d\ xs \triangleq\ & \ulcorner d = \mathit{None} \wedge xs = [] \urcorner \vee \\ & \exists\, \ell.\ \ulcorner d = \mathit{Some}(\ell) \urcorner * \\ & \ell \overset{\pi.n}{\Longmapsto} (\lambda ft.\ \mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs)\end{aligned}$$

$$\begin{aligned}\mathit{fiveTuple}_{\$}\ \pi\ n\ ft\ xs \triangleq\ & \exists\, p, l, m, r, s, xs_p, xss_l, xs_m, xss_r, xs_s. \\ & \$\mathit{potential}(|xs_p|, |xs_s|) * \\ & \ldots\end{aligned}$$

Potential in the structure: the natural amortised analysis technique in **Iris**$^{\$}$.

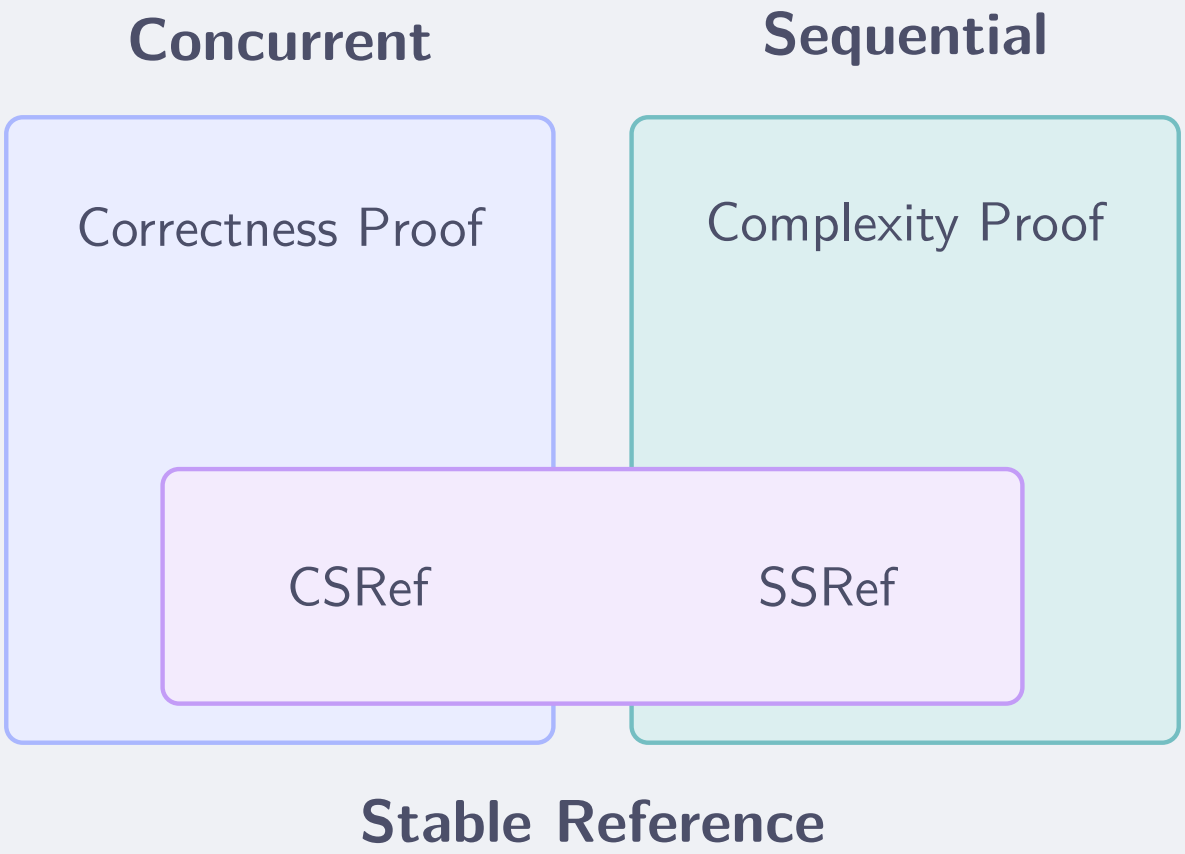
Potential is not Shareable

The concurrent stable references do not work with this definition.

$$\textsc{CSRef-Read}$$
$$\mathit{persistent}(\phi) \mathrel{-\!\!*}$$
$$\{\ell \Mapsto \phi\}\ !\ell\ (\exists v)\ v\ \{\phi\ v\}$$

fiveTuple is not persistent, as it contains credits. We need be able to borrow them and put them back later.

The Need For Two Interfaces



Sequential Stable References (SSRef)

SSRef-New-Pool
$\Rrightarrow \exists \pi.\, \lightning^{\pi.0}$

SSRef-Persist
$\mathit{persistent}(\ell \overset{\pi.n}{\Longmapsto} \phi)$

SSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \overset{\pi.n}{\Longmapsto} \phi\}$

Sequential Stable References (SSRef)

SSRef-New-Pool
$\Rrightarrow \exists \pi.\, \lightning^{\pi.0}$

SSRef-Persist
$\mathit{persistent}(\ell \overset{\pi.n}{\Longmapsto} \phi)$

SSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \overset{\pi.n}{\Longmapsto} \phi\}$

SSRef-Read
$(\forall v.\ \phi\ v \mathrel{-\!\!*} \phi\ v\ *\ \psi\ v) \mathrel{-\!\!*}$
$\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\}\ !\ell\ (\exists v)\ v\ \{\psi\ v\ *\ \lightning^{\pi.n}\}$

Sequential Stable References (SSRef)

SSRef-New-Pool
$\Rrightarrow \exists \pi.\, \lightning^{\pi.0}$

SSRef-Persist
$\mathit{persistent}(\ell \overset{\pi.n}{\Longmapsto} \phi)$

SSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \overset{\pi.n}{\Longmapsto} \phi\}$

SSRef-Read
$(\forall v.\ \phi\ v \mathrel{-\!\!*} \phi\ v\ *\ \psi\ v) \mathrel{-\!\!*}$
$\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\}\ !\ell\ (\exists v)\ v\ \{\psi\ v\ *\ \lightning^{\pi.n}\}$

SSRef-Read-Write
$$\left\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\right\}$$
$$!\ell$$
$$(\exists v)\ v\ \{\phi\ v\ *\ \lightning^{\pi.n+1}\ *\ \forall v'.\, \phi\ v' \mathrel{-\!\!*} \lightning^{\pi.n+1} \mathrel{-\!\!*} \mathsf{wp}\ \ell := v'\ ()\ \{\lightning^{\pi.n}\}\}$$

Sequential Stable References (SSRef)

SSRef-New-Pool
$\Rrightarrow \exists \pi.\, \lightning^{\pi.0}$

SSRef-Persist
$\mathit{persistent}(\ell \overset{\pi.n}{\Longmapsto} \phi)$

SSRef-Alloc
$\{\phi\ v\}\ \mathit{ref}\ v\ (\exists \ell)\ \ell\ \{\ell \overset{\pi.n}{\Longmapsto} \phi\}$

SSRef-Read
$(\forall v.\ \phi\ v \mathrel{-\!\!*} \phi\ v\ *\ \psi\ v) \mathrel{-\!\!*}$
$\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\}\ !\ell\ (\exists v)\ v\ \{\psi\ v\ *\ \lightning^{\pi.n}\}$

SSRef-Read-Write
$$\left\{\ell \overset{\pi.n}{\Longmapsto} \phi\ *\ \lightning^{\pi.n}\right\}$$
$$!\ell$$
$$(\exists v)\ v\ \{\phi\ v\ *\ \lightning^{\pi.n+1}\ *\ \forall v'.\, \phi\ v' \mathrel{-\!\!*} \lightning^{\pi.n+1} \mathrel{-\!\!*} \mathsf{wp}\ \ell := v'\ ()\ \{\lightning^{\pi.n}\}\}$$

This interface is not more general than CSRef: it forbids concurrent access.

Proof Excerpt – Temporarily Breaking Invariants

Initial Goal

```
"Hℓ" : ℓ ⤇{π, depth} fiveTuple π n xs
------------------------------------□
"Token" : Token π depth
------------------------------------*
WP let: "f" := ! #ℓ in
   [...]
   #ℓ <- "r";; naive_pop "r"
{{ v, ψ v }}
```

Proof Excerpt – Temporarily Breaking Invariants

Initial Goal

```
"Hℓ" : ℓ ⤇{π, depth} fiveTuple π n xs
--------------------------------------□
"Token" : Token π depth
--------------------------------------*
WP let: "f" := ! #ℓ in
   [...]
   #ℓ <- "r";; naive_pop "r"
{{ v, ψ v }}
```

```
wp_apply (ssref_read_write with "[Hℓ Token]") as "%ft (Token & Hft & DONE)".
```

Proof Excerpt – Temporarily Breaking Invariants

Initial Goal

```
"Hℓ" : ℓ ⤇{π, depth} fiveTuple π n xs
------------------------------------□
"Token" : Token π depth
------------------------------------*
WP let: "f" := ! #ℓ in
   [...]
   #ℓ <- "r";; naive_pop "r"
{{ v, ψ v }}
```

```
wp_apply (ssref_read_write with "[Hℓ Token]") as "%ft (Token & Hft & DONE)".
```

Subgoals

1. **Provide the token and the reference (`iFrame`).**
2. **Continue the proof:**

```
ft : val
"Hft" : fiveTuple π n xs ft
"Token" : Token π (depth + 1)
"DONE" : ∀t', fiveTuple π n xs t' -∗ Token π (S depth)
                -∗ WP (#ℓ ← t') {{ _, Token π depth }}
------------------------------------*
WP [...]
   #ℓ <- "r";; naive_pop "r"
{{ v, ψ v }}
```

Where the Original Proof Breaks

```
and pop_triple
 : type a. a triple nonempty_deque
         -> a triple * a triple deque
 = fun r ->
  let f = !r in
  let t = inspect_first f in
  let { first; child; last } = t in
  (* The (old)       mysterious condition: *)
  if not   (is_empty child) || B.has_length_3 first then
  (* The (repaired) mysterious condition: *)
  if not (B.is_empty  last) || B.has_length_3 first then
    naive_pop f
  else
    pop_nonempty r
```

The old one breaks potential invariants down the line in `pop_nonempty` in one single possible configuration of `t`. We have checked that this case is feasible.

You are here

1. Introduction
2. The KOT Data Structure
3. Specification
4. Stable References
5. Correctness Proof
6. Time Complexity: Statement and Proof
7. Conclusion

Conclusion

Summary of Achievements

- Applied the structure to a concurrent setting.
- Concurrent correctness and $O(1)$ amortised sequential complexity.
- Fixed a flaw in KOT's proof: thanks Rocq!

Key Contribution: Stable References

- We introduced *Stable References*, a novel abstraction for reasoning about shared memory cells supporting restricted updates.
- We developed two distinct interfaces with different capabilities:
  - **CSRef** for fully concurrent read/writes, requires persistent properties.
  - **SSRef** for any property (*e.g* time credits), forbids concurrent access.

Кошки

Figure: Philosophers' Problem

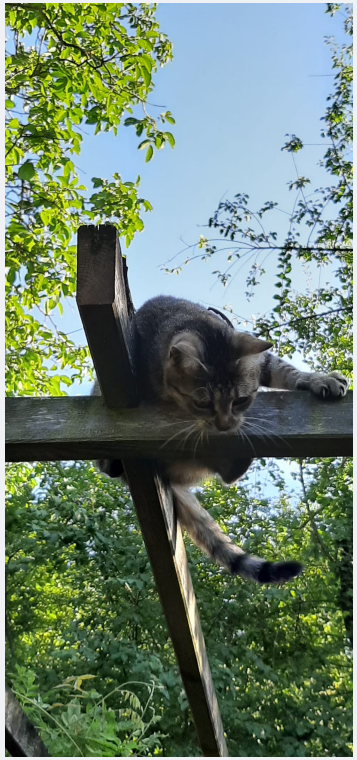


Figure: High Potential Cat

Tokens and Non-Atomic Invariants

Here is the (simplified) API for non-atomic invariants:

NAInv-New-Pool
$\Rrightarrow_0 \exists \pi.\ \lightning^{\pi.0}$

NAInv-New-Inv
$\triangleright P \Rrightarrow_n \square \mathit{NaInv}^{\pi.n}(P)$

NAInv-Acc
$\mathit{NaInv}^{\pi.n}(P) \vdash \lightning^{\pi.n} \propto_n (\triangleright P \ * \ \lightning^{\pi.n+1})$

Where $P \propto_n Q$ means we can get Q from P then recover P from Q[2].

[2] also does view-shifting into the next namespace

Time Credits

$$\textsc{tick\_cost} \in \{0, 1\}$$

Time-Zero
$\$0 \dashv\vdash \mathit{emp}$

Time-Combine
$\$(n + m) \dashv\vdash \$n \ * \ \$m$

Tick-Spec
$\{\$\,\textsc{tick\_cost}\}\ \mathit{tick}()\ ()\ \{\mathit{emp}\}$

OCaml → Tests & Proofs

OCaml Testing

- `Monolith` to identify issues.
- Confirmed KOT error feasibility.

HeapLang for Proof

- Manual translation from OCaml.
- Some recent tools propose automation.

Sequential Specification For `pop_triple`

PopTriple

$$\{\mathit{dequeOfTriples}_{\$}\ \pi\ n\ d\ ([t] +\!\!+ ts)\ *\ \lightning^{\pi \cdot n}\ *\ \$171\}$$
$$\mathit{pop\_triple}\ d$$
$$(\exists d')\ (t,\ d')\ \left\{ \lightning^{\pi \cdot n}\ *\ \left(\begin{array}{c} \mathit{nonSpecialTriple}\ \pi\ n\ xst\ *\ \mathit{deque}_{\$}\ \pi\ n\ d'\ ts \\ \bigvee \\ \mathit{specialTriple}\ \pi\ n\ xst\ *\ \$(171-4)\ * \\ \forall t',\ \mathsf{wp}\ \mathit{push}\ t'\ d'\ (\exists d'')\ d''\ \{\mathit{deque}_{\$}\ \pi\ n\ d''\ ([t'] +\!\!+ ts)\} \end{array} \right) \right\}$$