Diamonds Are Forever: Reasoning about Heap Space in a Concurrent and Garbage Collected Language

Alexandre Moine Arthur Charguéraud François Pottier

Iris'23

Formal Verification of Heap Space Bounds

Following Hofmann [1999], let $\Diamond 1$ represent one space credit.
$\rightsquigarrow$ the right to allocate one memory word.

Without GC:

- `alloc` consumes space

  $$\{\, \Diamond \mathit{size}(b) \,\}\; \ell := \mathtt{alloc}(b)\; \{\, \ell \mapsto b \,\}$$

- `free` produces space

  $$\{\, \ell \mapsto b \,\}\; \mathtt{free}(\ell)\; \{\, \Diamond \mathit{size}(b) \,\}$$

With GC: no syntactical point to recover space credits.

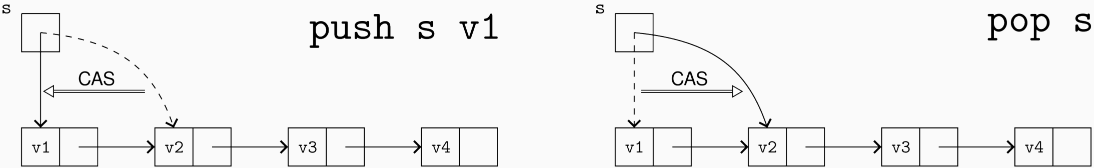
Formal Verification of Heap Space Bounds Under Garbage Collection

Space credits can be recovered as soon as a location becomes unreachable:
$\rightsquigarrow$ from the roots following heap paths.

Separation Logic for Heap Space with GC	High-Level	Concurrency
Madiot and Pottier [2022]	X	✓ (no examples)
Moine, Charguéraud and Pottier [2023]	✓	X
This work	✓	✓

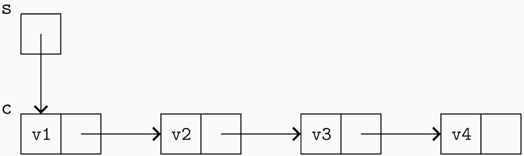
A Motivating Example: The Treiber's Stack

A lock-free linearizable stack implemented as a reference on a immutable list.

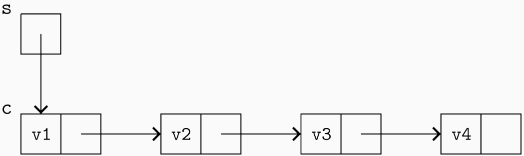


What is a space-aware specification for `push` and `pop`?

A Space-Leaking Interleaving for pop



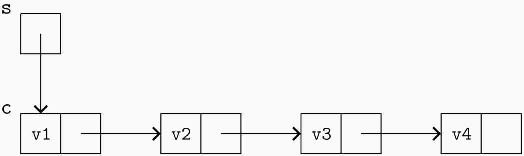
A Space-Leaking Interleaving for pop




```
pop s
```

```
pop s;
pop s;
pop s;
...
```

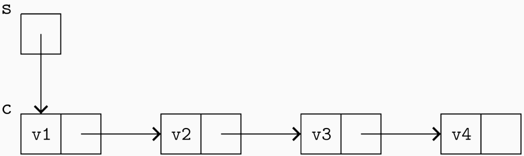
A Space-Leaking Interleaving for pop




```
let l = !s in
match l with
...
```

```
pop s;
pop s;
pop s;
...
```

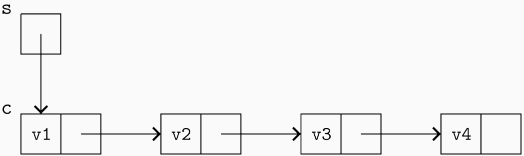
A Space-Leaking Interleaving for pop




```
let l = c in
match l with
...
```

```
pop s;
pop s;
pop s;
...
```

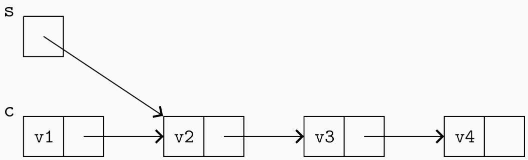
A Space-Leaking Interleaving for pop




```
let l = c in
match l with
...
```


```
pop s;
pop s;
pop s;
...
```

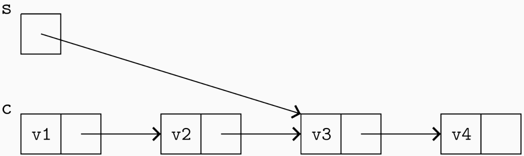
A Space-Leaking Interleaving for pop




```
let l = c in
match l with
...
```


```
pop s;
pop s;
pop s;
...
```

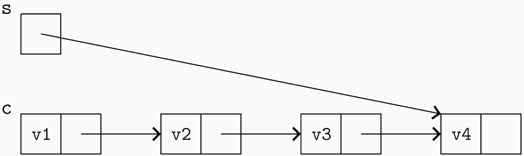
A Space-Leaking Interleaving for pop




```
let l = c in
match l with
...
```


```
pop s;
pop s;
pop s;
...
```

A Space-Leaking Interleaving for pop




```
let l = c in
match l with
...
```


```
pop s;
pop s;
pop s;
...
```

- The sleeping thread maintains reachable a morally dead structure.
- pop cannot produce space credits!

Contributions

We present the first program logic to reason about

heap space usage for a high-level and concurrent language with GC.

Key contributions:

- A new pointed-by-thread assertion, tracking in which thread a location is a root.
- Examples:
  - Lock-free data structures: Treiber's stack, Michael and Scott's queue
  - Closures: Concurrent counter
- Theory and examples are fully mechanized in Iris.

Prior Work: Pointed-by-Heap Assertions to Track Heap Predecessors

From Kassios and Kritikos [2013], Moine et al. [2023]:

$$\ell \mapsfrom_q L$$

(signed) multiset

$\in [0, 1]$

- $\ell \mapsfrom_1 L$ asserts that L is an over-approximation of the reachable predecessors of ℓ.
- $\ell \mapsfrom_1 \emptyset$ asserts that ℓ is unreachable from the heap.

$$\ell \mapsfrom_1 \{+\ell_1; +\ell_2\} \quad \mathrel{-\!\!*} \quad \ell \mapsfrom_{\frac{1}{2}} \{+\ell_1\} * \ell \mapsfrom_{\frac{1}{2}} \{+\ell_2\}$$

$$\ell \mapsfrom_{\frac{1}{2}} \{+\ell_1\} * \ell \mapsfrom_0 \{-\ell_1\} \quad \mathrel{-\!\!*} \quad \ell \mapsfrom_{\frac{1}{2}} \emptyset$$

Main invariant: if $\ell \mapsfrom_0 L$ then L must only contain negative elements.

What Are the Roots?

- We consider an interleaving semantics.
- The GC is interleaved with the per-thread reduction.

The Free Variable Rule (FVR), adapted from Felleisen and Hieb [1992]

In a substitution-based semantics, the roots of a threadpool consist of the union of the locations occurring in each thread.

Separation Logic Triples & Pointed-By-Thread Assertions

We use a ghost thread identifier π to identify a thread. $\{\,\Phi\,\}\,\pi\colon t\,\{\,\Psi\,\}$

The pointed-by-thread assertion:

$$\ell \Leftarrow\!\!|_p\ \Pi$$

set of thread identifiers

$\in (0, 1]$

- $\ell \Leftarrow\!\!|_1\ \Pi$ asserts that Π is an over-approximation of the threads in which ℓ is a root.
- $\ell \Leftarrow\!\!|_1\ \emptyset$ asserts that ℓ is not a root.

$$\ell \Leftarrow\!\!|_{(p_1+p_2)}\ (\Pi_1 \cup \Pi_2) \quad \equiv \quad \ell \Leftarrow\!\!|_{p_1}\ \Pi_1 \ * \ \ell \Leftarrow\!\!|_{p_2}\ \Pi_2$$

Our Logical Deallocation Rule

$$\ell \mapsto_p \vec{w} \;*\; \ell \hookleftarrow_1 \emptyset \;*\; \ell \Lleftarrow_1 \emptyset \quad\Rrightarrow\quad \ell \mapsto_p \vec{w} \;*\; \Diamond size(\vec{w}) \;*\; \dagger\, \ell$$

- Free does not consume the points-to. Useful for:
  - Persistent objects
  - Closing invariants

Our Logical Deallocation Rule

ℓ is a valid block

ℓ is not a root

$$\ell \mapsto_p \vec{w} \;*\; \ell \mapsfrom_1 \emptyset \;*\; \ell \Mapsfrom_1 \emptyset \quad \Rrightarrow \quad \ell \mapsto_p \vec{w} \;*\; \Diamond size(\vec{w}) \;*\; \dagger\,\ell$$

ℓ is unreachable from the heap

- Free does not consume the points-to. Useful for:
  - Persistent objects
  - Closing invariants

Our Logical Deallocation Rule

$$\ell \mapsto_p \vec{w} \ * \ \ell \longleftarrow\!\!\!\!\!\mid_1 \emptyset \ * \ \ell \Longmapsfrom_1 \emptyset \quad \overset{\text{only applicable in a precondition}}{\Rrightarrow} \quad \ell \mapsto_p \vec{w} \ * \ \Diamond \mathit{size}(\vec{w}) \ * \ \dagger \ell$$

- Free does not consume the points-to. Useful for:
    - Persistent objects
    - Closing invariants

Our Logical Deallocation Rule

$$\ell \mapsto_p \vec{w} \;\ast\; \ell \mapsfrom_1 \emptyset \;\ast\; \ell \Mapsfrom_1 \emptyset \quad \Rrightarrow \quad \underbrace{\ell \mapsto_p \vec{w}}_{\ell \text{ is still a valid block}} \;\ast\; \overbrace{\Diamond \mathit{size}(\vec{w})}^{\text{space credits}} \;\ast\; \underbrace{\dagger\,\ell}_{\ell \text{ is unreachable}}$$

- Free does not consume the points-to. Useful for:
  - Persistent objects
  - Closing invariants

Life of the Pointed-by-Thread Assertion, Part 1

The rule for allocation.

$$\{\, \Diamond n \,\}\; \pi \colon (\mathsf{alloc}\ n)\; \{\, \lambda\, \ell.\ \ \ell \mapsto [\,\overbrace{(), \ldots, ()}^{n\text{ times}}\,] \;*\; \ell \hookleftarrow \emptyset \;*\; \boxed{\ell \Lleftarrow \{\pi\}} \,\}$$

After a load, a thread points-to the loaded value.

$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\{\, \ell \mapsto_r \vec{w} \;*\; \boxed{v \Lleftarrow_p \emptyset} \,\}\; \pi \colon \ell[i]\; \{\, \lambda\, v'.\, \ulcorner v' = v \urcorner \;*\; \ell \mapsto_r \vec{w} \;*\; \boxed{v \Lleftarrow_p \{\pi\}} \,\}}$$

Life of the Pointed-by-Thread Assertion, Part 2

The $\ell \Lleftarrow_p \{\pi\}$ assertion can be cleaned when ℓ does not appear in the focused part of π.

$$\frac{\ell \notin \mathit{locs}(t) \qquad \{\, \ell \Lleftarrow_p \emptyset \, * \, \Phi \,\}\, \pi \colon t \,\{\, \Psi \,\}}{\{\, \ell \Lleftarrow_p \{\pi\} \, * \, \Phi \,\}\, \pi \colon t \,\{\, \Psi \,\}}$$

$\ell \Lleftarrow_p \{\pi\}$ is force-framed when ℓ becomes a root of the evaluation context in π.

$$\frac{\mathit{locs}(t_2) = \{\ell\} \qquad \{\, \Phi \,\}\, \pi \colon t_1 \,\{\, \Psi' \,\} \qquad \forall v.\, \{\, \boxed{\ell \Lleftarrow_p \{\pi\}} * \Psi' \; v \,\}\, \pi \colon [v/x]t_2 \,\{\, \Psi \,\}}{\{\, \boxed{\ell \Lleftarrow_p \{\pi\}} * \Phi \,\}\, \pi \colon (\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) \,\{\, \Psi \,\}}$$

Forking!

A fork updates pointed-by-thread assertions.

$$\frac{\begin{array}{c} \mathit{locs}(t) = \{\ell\} \\ (\forall \pi'.\quad \{\, \ell \Lleftarrow_p \{\pi'\} \ast \Phi \,\}\, \pi' : t \,\{\, \lambda\_.\, \ulcorner \mathsf{True} \urcorner \,\}) \end{array}}{\{\, \ell \Lleftarrow_p \{\pi\} \ast \Phi \,\}\, \pi : \mathsf{fork}\; t \,\{\, \lambda\_.\, \ulcorner \mathsf{True} \urcorner \,\}}$$

The Soundness Theorem

Our semantics

- is parameterized by a maximal heap size S
- interleaves reduction steps and GC steps

An allocation is stuck if, after a full GC, there is not enough free space.

Soundness Theorem

If $\{\ \Diamond S\ \}\ \pi\colon t\ \{\ \Psi\ \}$ holds, then t cannot reach a stuck configuration.

Reformulation: the live heap space of any execution of t cannot exceed S.

Treiber's Push – The Standard Specification Using Atomic Triples

Private precondition $\longrightarrow$ Public precondition $\longleftarrow$ Public postcondition $\longleftarrow$ Private postcondition $\longrightarrow$

$$\frac{\{\ \text{stack-inv}\ s\ \}}{\dfrac{\begin{array}{c}\langle\ \forall L.\ \text{content}\ s\ L\ \rangle \\ \pi : \text{push}\ s\ v \\ \langle\ \text{content}\ s\ (v :: L)\ \rangle\end{array}}{\{\ \lambda\_.\ \ulcorner\text{True}\urcorner\ \}}}$$

- The public precondition is atomically updated into the public postcondition.
- The user can open invariants around a public precondition and postcondition.

Space-Aware Treiber's Push

- Stack of unboxed values $\rightsquigarrow$ focus on the space consumption of the structure.
- A fractional access token for the stack ($r \in (0, 1]$):

$$\mathsf{stack}\ s\ r$$

If $r = 1$ then the stack is not being concurrently used.

$$\frac{\dfrac{\{\ \mathsf{stack}\ s\ r\ *\ \Diamond 2\ \}}{\langle\ \forall L.\ \mathsf{content}\ s\ L\ \rangle}}{\dfrac{\pi\colon \mathsf{push}\ s\ v}{\dfrac{\langle\ \mathsf{content}\ s\ (v :: L)\ \rangle}{\{\ \lambda\_.\ \mathsf{stack}\ s\ r\ \}}}}$$

Space-Aware Treiber's Pop

- pop cannot give back space.
- It returns virtual credits $\blacklozenge_s\, 2$.

$$\frac{\{\ \text{stack}\ s\ r\ \}}{\dfrac{\langle\ \forall\, v\, L.\ \text{content}\ s\ (v :: L)\ \rangle \quad \pi : \text{pop}\ s \quad \langle\ \text{content}\ s\ L\ \rangle}{\{\ \lambda\, v'.\ \ulcorner v' = v \urcorner\ *\ \text{stack}\ s\ r\ *\ \blacklozenge_s\, 2\ \}}}$$

If the stack is not being concurrently used,
virtual credits can be converted to physical credits.

$$\text{stack}\ s\ 1\ *\ \blacklozenge_s\, 2 \quad \Rrightarrow \quad \text{stack}\ s\ 1\ *\ \lozenge 2$$

Allocation and Deallocation of a Stack

Allocation of a stack:

$$\{\,\Diamond 1\,\}\,\pi : \mathsf{create}\,()\,\{\,\lambda s.\ \mathsf{stack}\ s\ 1\ *\ \mathsf{content}\ s\ []\ *\ s \hookleftarrow \emptyset\ *\ s \Lleftarrow \{\pi\}\,\}$$

Logical deallocation of a stack:

$$\mathsf{stack}\ s\ 1\ *\ \mathsf{content}\ s\ L\ *\ s \hookleftarrow \emptyset\ *\ s \Lleftarrow \emptyset \quad \Rrightarrow$$

$$\mathsf{stack}\ s\ 1\ *\ \Diamond(1 + 2 \times |L|)$$

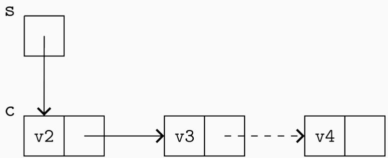
Stack With Boxed Values

$$\begin{array}{c} \{\ \mathsf{stack}\ s\ r \ \ast\ v \mapsfrom_q \emptyset \ \ast\ v \Mapsfrom_p \{\pi\} \ \ast\ \Diamond 2\ \} \\ \hline \langle\ \forall L.\ \mathsf{content}\ s\ L\ \rangle \\ \pi : \mathsf{push}\ s\ v \\ \langle\ \mathsf{content}\ s\ ((v, q, p) :: L)\ \rangle \\ \hline \{\ \lambda\_.\ \mathsf{stack}\ s\ r\ \} \end{array}$$

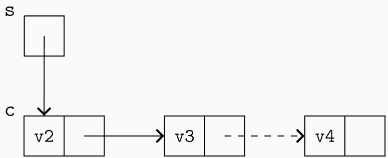
The specification of pop is also richer:

- The pointed-by-heap assertion cannot be returned without an additional write.
- Can the pointed-by-thread assertion be returned?
  $\rightsquigarrow$ yes, but we need to change the semantics.

Stack With Boxed Values – Another Dangerous Interleaving

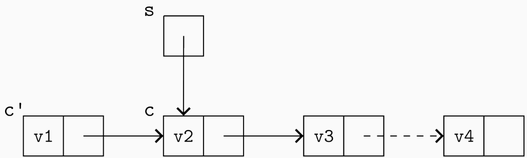


Stack With Boxed Values – Another Dangerous Interleaving



`push s v1`	`pop s;` `...`

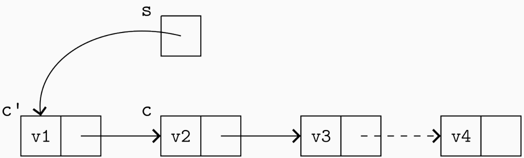
Stack With Boxed Values – Another Dangerous Interleaving




```
if CAS s c c'
then ()
else push s v1 -- retry
```

```
pop s;
...
```

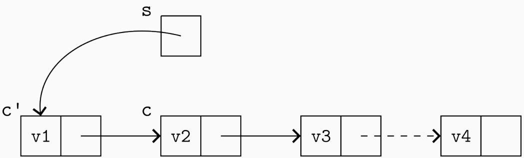
Stack With Boxed Values – Another Dangerous Interleaving




```
if true
then ()
else push s v1 -- retry
```

```
pop s;
...
```

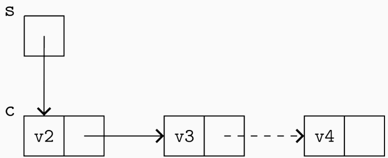
Stack With Boxed Values – Another Dangerous Interleaving




```
if true
then ()
else push s v1 -- retry
```

```
pop s;
...
```

Stack With Boxed Values – Another Dangerous Interleaving




```
if true
then ()
else push s v1 -- retry
```

```
pop s;
...
```

The sleeping thread apparently maintains reachable the pushed value.

Safe Points to the Rescue

The scenario of the previous slide does not occur in OCaml 5.

- The GC is stop-the-world.
- Threads run independently and, at safe points, lookup if a GC is pending.
- The GC runs only when every thread reached a safe point.
- In particular, no safe point before `if true then _ else _`.
- This expression always releases the roots of the `else` branch.
- Interleaving of small steps is too fine. We need a coarser interleaving.

For now: an atomic ifCAS primitive.

Not Shown Here

More meta-theory:

- Encoding of the logic for sequential programs
- Simplified mode where no deallocation is required
- Closures

More examples (with the indirect help of some of you!):

- Michael & Scott's queue, based on the proof of Vindum and Birkedal [2021]
- Async-Finish library (proof involving later credits [Spies et al., 2022])

$\rightsquigarrow$ Some automation thanks to Diaframe [Mulder et al., 2022]

Conclusion & Future Work

We present the first program logic to reason about

heap space usage for a high-level and concurrent language with GC.

Please talk to me if you know data structures with a cool space usage under GC!

Independently of heap space, our logic allows reasoning about unreachability.
$\rightsquigarrow$ Can we apply our ideas to other areas?

Thank you for your attention!

alexandre.moine [at] inria.fr
arthur.chargueraud [at] inria.fr
francois.pottier [at] inria.fr

Cycles

We handle cycles following the approach of Madiot and Pottier [2022].

$$\ulcorner \mathrm{True} \urcorner \twoheadrightarrow\!\!\!\ast\ \emptyset \,\text{☁}^{0}\, P$$

$$\begin{array}{r} D \,\text{☁}^{n}\, P \\ \ell \mapsto_p \vec{v} \ast \ell \leftarrow\!\!\dashv L \ast \ell \Leftarrow\!\!\dashv \emptyset \end{array} \;\; \twoheadrightarrow\!\!\!\ast\ (\{\ell\} \cup D) \,\text{☁}^{n+\mathit{size}(\vec{v})}\, P \quad \text{if } L \subseteq P$$

$$D \,\text{☁}^{n}\, D \Rrightarrow \Diamond n \ast \Big(\mathop{\Large\ast}_{\ell \in D} \dagger\ell \Big) \qquad \text{if } D \cap \mathit{locs}(t) = \emptyset$$

Proof Insight: Liveness-Based Cancellable Invariants

Recall standard cancellable invariants: $\mathsf{CInv}^{\gamma}\,\Phi \triangleq \boxed{\Phi \lor \fbox{1}^{\gamma}}$

We can define liveness-based invariants: $\mathsf{LInv}_{\ell}\,\Phi \triangleq \boxed{\Phi \lor \dagger\ell}$

- The user can access the invariant as long as the location is logically allocated.
- Deallocation cancels the invariant.
- Useful to avoid another fractional token.

The semantics ensures no dangling pointers.

$$\frac{\ell \in \mathit{locs}(t)}{\{\,\dagger\ell\,\}\;\pi\colon t\;\{\,\Psi\,\}}$$

Triples with Souvenir

Pointed-by-thread assertions are easy to manage in practice.

Introducing triples with souvenir $[\,R\,]\{\,\Phi\,\}\,\pi\colon t\,\{\,\Psi\,\}$

"Give a pointed-by-thread assertion once and that's it"

$$\frac{locs(t_2) = \{\ell\} \qquad [\,R \cup \{\ell\}\,]\{\,\Phi\,\}\,\pi\colon t_1\,\{\,\Psi'\,\} \qquad \forall v.\,[\,R\,]\{\,\ell \Lleftarrow_p \{\pi\} * \Psi'\ v\,\}\,\pi\colon [v/x]t_2\,\{\,\Psi\,\}}{[\,R\,]\{\,\ell \Lleftarrow_p \{\pi\} * \Phi\,\}\,\pi\colon \mathrm{let}\,x = t_1\,\mathrm{in}\,t_2\,\{\,\Psi\,\}}$$

$$\frac{locs(t_2) = \{\ell\} \qquad \ell \in R \qquad [\,R\,]\{\,\Phi\,\}\,\pi\colon t_1\,\{\,\Psi'\,\} \qquad \forall v.\,[\,R\,]\{\,\Psi'\ v\,\}\,\pi\colon [v/x]t_2\,\{\,\Psi\,\}}{[\,R\,]\{\,\Phi\,\}\,\pi\colon \mathrm{let}\,x = t_1\,\mathrm{in}\,t_2\,\{\,\Psi\,\}}$$

The NoFree Mode

If the user pledges not to clean, framing pointed-by-thread assertions is not needed.

$$\frac{\textsc{LetNoClean}\quad [\,\text{NoClean}\,]\{\,\Phi\,\}\,\pi\colon t_1\,\{\,\Psi'\,\} \qquad \forall v.\ [\,R\,]\{\,\Psi'\,v\,\}\,\pi\colon [v/x]t_2\,\{\,\Psi\,\}}{[\,R\,]\{\,\Phi\,\}\,\pi\colon \text{let}\,x = t_1\ \text{in}\ t_2\,\{\,\Psi\,\}}$$

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science, 103(2):235–271, 1992. URL https://www2.ccs.neu.edu/racket/pubs/tcs92-fh.pdf.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In Logic in Computer Science (LICS), pages 464–473, July 1999. URL https://doi.org/10.1109/LICS.1999.782641.

Ioannis T. Kassios and Eleftherios Kritikos. A discipline for program verification based on backpointers and its use in observational disjointness. In European Symposium on Programming (ESOP), volume 7792 of Lecture Notes in Computer Science, pages 149–168. Springer, March 2013. URL https://doi.org/10.1007/978-3-642-37036-6_10.

References] References

Jean-Marie Madiot and François Pottier. A separation logic for heap space under garbage collection. Proceedings of the ACM on Programming Languages, 6(POPL), January 2022. URL http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf.

Alexandre Moine, Arthur Charguéraud, and François Pottier. A high-level separation logic for heap space under garbage collection. Proc. ACM Program. Lang., 7(POPL), jan 2023. doi: 10.1145/3571218. URL https://doi.org/10.1145/3571218.

[] [

Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in iris. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, page 809–824, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432.

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Later credits: Resourceful reasoning for the later modality. Proc. ACM Program. Lang., 6(ICFP), aug 2022. doi: 10.1145/3547631. URL https://doi.org/10.1145/3547631.

[] [

Simon Friis Vindum and Lars Birkedal. Contextual refinement of the Michael-Scott queue. In Certified Programs and Proofs (CPP), pages 76–90, January 2021. URL https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf.