Heap Space Bounds of Concurrent Programs under Garbage Collection with Separation Logic

Alexandre Moine    Arthur Charguéraud    François Pottier

November 16, 2023

Ínria
\[
\{ \Phi \} t \{ \Psi \}
\]

• Under the precondition \( \Phi \) the program \( t \) is safe to execute.

• If \( t \) terminates, then it does so with post-condition \( \Psi \).

• \( \Phi \) and \( \Psi \) are heap predicates.

\[\ell \mapsto v\]

Knowledge: \( \ell \) points-to \( v \).

\( \Phi_1 \) and \( \Phi_2 \) are separated.

Ownership: I uniquely own \( \ell_1 \mapsto v_1 \) implying \( \ell_1 \neq \ell_2 \) if this information.
Separation Logic: A Program Logic to Rule Them All

\[
\{ \Phi \} \; t \; \{ \Psi \} 
\]

- Under the precondition $\Phi$ the program $t$ is safe to execute.
- If $t$ terminates, then it does so with post-condition $\Psi$.
- $\Phi$ and $\Psi$ are heap predicates.
Separation Logic: A Program Logic to Rule Them All

\[
\begin{align*}
\{ \Phi \} \ t \ \{ \Psi \} \\
\end{align*}
\]

- Under the precondition $\Phi$ the program $t$ is safe to execute.
- If $t$ terminates, then it does so with post-condition $\Psi$.
- $\Phi$ and $\Psi$ are heap predicates.

\[\ell \mapsto v\]

\[\Phi_1 \ast \Phi_2\]
Separation Logic: A Program Logic to Rule Them All

\{ \Phi \} t \{ \Psi \}

- Under the precondition $\Phi$ the program $t$ is safe to execute.
- If $t$ terminates, then it does so with post-condition $\Psi$.
- $\Phi$ and $\Psi$ are heap predicates.

$\ell \mapsto v$

Knowledge: $\ell$ points-to $v$.
Ownership: I uniquely own this information.
Separation Logic: A Program Logic to Rule Them All

\{ \Phi \} \; t \{ \Psi \}

- Under the precondition $\Phi$ the program $t$ is safe to execute.
- If $t$ terminates, then it does so with post-condition $\Psi$.
- $\Phi$ and $\Psi$ are heap predicates.

$\ell \mapsto v$

Knowledge: $\ell$ points-to $v$.
Ownership: I uniquely own this information.

$\Phi_1 \ast \Phi_2$

$\Phi_1$ and $\Phi_2$ are separated.

$\ell_1 \mapsto v_1 \ast \ell_2 \mapsto v_2$ implies $\ell_1 \neq \ell_2$
Separated Logic with Space Credits

Key Idea

Space as a resource.

Following Hofmann [1999], let ♢ represent one space credit. 

\[
\begin{align*}
\{ \triangledown \text{size}(b) \}^\ell &:= \text{alloc}(b) \{ \ell \mapsto b \} \{ \ell \mapsto b \} \text{free}(\ell) \{ \triangledown \text{size}(b) \} \\
\end{align*}
\]

If \( \{ \triangledown S \} t \{ \Psi \} \) holds, then a heap of size \( S \) is sufficient to run \( t \).
Key Idea

Space as a resource.

Following Hofmann [1999], let \( \diamond 1 \) represent one space credit.
Separation Logic with Space Credits

**Key Idea**

Space as a resource.

Following Hofmann [1999], let $\Diamond 1$ represent one space credit.

$$\{ \Diamond \text{size}(b) \}\ell := \text{alloc}(b)\{\ell \mapsto b\} \quad \{ \ell \mapsto b \}\text{free}(\ell)\{\Diamond \text{size}(b)\}$$
Key Idea

Space as a resource.

Following Hofmann [1999], let $\Diamond 1$ represent one space credit.

\[
\{ \Diamond \text{size}(b) \} \ell := \text{alloc}(b) \{ \ell \mapsto b \} \quad \{ \ell \mapsto b \} \text{free}(\ell) \{ \Diamond \text{size}(b) \}
\]

If $\{ \Diamond S \} t \{ \Psi \}$ holds, then a heap of size $S$ is sufficient to run $t$. 
• OCaml (and many other languages) comes with a Garbage Collector (GC).
• There is no free operation.
• If space is needed, the GC will reclaim unreachable locations.
OCaml (and many other languages) comes with a Garbage Collector (GC).

There is no free operation.

If space is needed, the GC will reclaim unreachable locations.

The GC simplifies the life of programmers but complicates ours.

~~ availability of space depends on reachability arguments.
In the sequential setting:

- For a low-level language, Madiot and Pottier [2022]
In the sequential setting:

- For a low-level language, Madiot and Pottier [2022]
- For a high-level language, Moine, Charguéraud, and Pottier [2023]
In the sequential setting:

- For a low-level language, Madiot and Pottier [2022]
- For a high-level language, Moine, Charguéraud, and Pottier [2023]

**Key Ideas**

Track reachability inside the logic.
Free as a ghost update.

\[
\ell \mapsto b \quad \text{"} \ell \text{ is unreachable} \quad \Rightarrow \quad \Diamond \text{size}(b) \quad \uparrow \ell
\]
Two challenges:

1. **Reachability**: we need to track which location is reachable by which thread.
2. **Semantics**: we need to account for a fine interleaving of thread reduction and GC.

We address these two challenges.

- We introduce the pointed-by-thread assertion to track roots within the logic.
- We present a technique of GC-less sections to finely reason about concurrent GC.

Theory and examples are fully mechanized in Coq on top of Iris.
Two challenges:

1. **Reachability**: we need to track which location is reachable by which thread.
2. **Semantics**: we need to account for a fine interleaving of thread reduction and GC.

We address these two challenges.

1. We introduce the **pointed-by-thread** assertion to track roots within the logic.
2. We present a technique of **GC-less sections** to finely reason about concurrent GC.
Two challenges:

1. **Reachability**: we need to track which location is reachable by which thread.
2. **Semantics**: we need to account for a fine interleaving of thread reduction and GC.

We address these two challenges.

1. We introduce the *pointed-by-thread* assertion to track roots within the logic.
2. We present a technique of *GC-less sections* to finely reason about concurrent GC.

Theory and examples are **fully mechanized** in Coq on top of Iris.
The set of reachable locations is computed:

1. from the roots, the locations bounded in live variables,
2. following heap paths.
Reachability and Unreachability

The set of reachable locations is computed:

1. from the roots, the locations bounded in live variables,
2. following heap paths.

A location $\ell$ is unreachable if and only if:

1. $\ell$ is not a root
2. $\ell$ is not reachable by any reachable heap cell.
The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

\[ \ell \leftarrow A \]
The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

\[ \ell \leftarrow A \]

- Intuitively: a dual to the points-to assertion.
- \( \ell \leftarrow A \) asserts that \( A \) is an over-approximation of the reachable predecessors of \( \ell \).
- \( \ell \leftarrow \emptyset \) asserts that \( \ell \) has no reachable predecessors!
The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

\[ \ell \leftarrow A \]

- Intuitively: a dual to the points-to assertion.
- \( \ell \leftarrow A \) asserts that \( A \) is an over-approximation of the reachable predecessors of \( \ell \).
- \( \ell \leftarrow \emptyset \) asserts that \( \ell \) has no reachable predecessors!

This assertion is created upon allocation,
Reachability from the Heap: the Pointed-By-Heap Assertion

The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

\[ \ell \leftarrow A \]

- Intuitively: a dual to the points-to assertion.
- \( \ell \leftarrow A \) asserts that \( A \) is an over-approximation of the reachable predecessors of \( \ell \).
- \( \ell \leftarrow \emptyset \) asserts that \( \ell \) has no reachable predecessors!

This assertion is created upon allocation, updated by store,
The pointed-by-heap assertion [Kassios and Kritikos, 2013, Madiot and Pottier, 2022]

\[ \ell \leftarrow A \]

- Intuitively: a dual to the points-to assertion.
- \( \ell \leftarrow A \) asserts that \( A \) is an over-approximation of the reachable predecessors of \( \ell \).
- \( \ell \leftarrow \emptyset \) asserts that \( \ell \) has no reachable predecessors!

This assertion is created upon allocation, updated by store, and required for deallocation.
Reachability from Threads: the Pointed-by-Thread Assertion

\[ \ell \iff \Pi \]

- \[ \ell \leftarrow \emptyset \] asserts that \( \ell \) is not a root!

Load \{ \ell \mapsto \ell' \} \[ \rho : \ell \}

\{ \ell' \mapsto \ell' \} \leftarrow (\Pi \cup \{ \rho \})
Reachability from Threads: the Pointed-by-Thread Assertion

\[ \ell \Leftrightarrow \Pi \]

- \( \ell \Leftrightarrow \Pi \) asserts that \( \Pi \) is an over-approximation of the threads in which \( \ell \) is a root.
- \( \ell \Leftrightarrow \emptyset \) asserts that \( \ell \) is not a root!
Reachability from Threads: the Pointed-by-Thread Assertion

\[ \ell \iff \Pi \]

- \( \ell \iff \Pi \) asserts that \( \Pi \) is an over-approximation of the threads in which \( \ell \) is a root.
- \( \ell \iff \emptyset \) asserts that \( \ell \) is not a root!

**Load**

\[
\{ \ell \mapsto \ell' \ast \ell' \iff \Pi \} \pi : \forall \ell \{ \ell' : \ell \mapsto \ell' \ast \ell' \iff (\Pi \cup \{ \pi \}) \}
\]
The pointed-by-thread assertion can be cleaned.

\[
\text{CLEANUP}
\begin{align*}
\ell & \notin \text{locs}(t) & \{ \ell \leftrightarrow (\Pi \setminus \{\pi\}) \ast \Phi \} \pi : t \{ \Psi \} \\
\{ \ell \leftrightarrow \Pi \ast \Phi \} \pi : t \{ \Psi \}
\end{align*}
\]
Cleanup

The pointed-by-thread assertion can be cleaned.

\[
\begin{align*}
\text{CLEANUP} & \quad \ell \notin \text{locs}(t) \quad \{ \ell \Leftarrow (\Pi \setminus \{\pi\}) \ast \Phi \} \pi : t \{ \Psi \} \\
& \quad \{ \ell \Leftarrow \Pi \ast \Phi \} \pi : t \{ \Psi \}
\end{align*}
\]

Unveiling our logical deallocation rule.

\[
\ell \mapsto b \ast \ell \Leftarrow \emptyset \ast \ell \Leftarrow \emptyset \ \Rightarrow \ \diamond \text{size}(b) \ast \dagger \ell
\]
The semantics of concurrent languages is usually expressed with **interleaving**.

\[
\text{INTERLEAVE} \\
\frac{t / \sigma \xrightarrow{\text{step}} t' / \sigma'}{ts \cup \{t\} / \sigma \longrightarrow ts \cup \{t'\} / \sigma'}
\]
The semantics of concurrent languages is usually expressed with *interleaving*.

\[
\text{INTERLEAVE} \quad \frac{t / \sigma \xrightarrow{\text{step}} t' / \sigma'}{ts \cup \{t\} / \sigma \rightarrow ts \cup \{t'\} / \sigma'}
\]

- In OCaml (and in Java), the GC is *stop-the-world*.
- First idea: interleave GC with per-thread reduction.

\[
\text{INTERLEAVE} \quad \frac{t / \sigma \xrightarrow{\text{step}} t' / \sigma'}{ts \cup \{t\} / \sigma \rightarrow ts \cup \{t'\} / \sigma'}
\]

\[
\text{GC} \quad \frac{locs(ts) \vdash \sigma \xrightarrow{\text{gc}} \sigma'}{ts / \sigma \rightarrow ts / \sigma'}
\]
A linearizable lock-free stack, implemented as a reference on an immutable list.
A linearizable lock-free stack, implemented as a reference on an immutable list.

let rec pop s =
  let l = !s in
  match l with
  | nil -> assert false
  | v::l' -> if CAS s l l' then v else pop s
A Space-Leaking Interleaving for pop?

- The sleeping thread maintains reachable a morally dead structure.
- pop cannot produce space credits?!
A Space-Leaking Interleaving for pop?

- The sleeping thread maintains reachable a morally dead structure.
- pop $s$ cannot produce space credits?!
A Space-Leaking Interleaving for pop?

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

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

The sleeping thread maintains reachable a morally dead structure.

pop cannot produce space credits?!
A Space-Leaking Interleaving for pop?

let l = c in
match l with
...

\[ s \]
\[ c \]
\[ v1 \rightarrow v2 \rightarrow v3 \rightarrow v4 \]

- The sleeping thread maintains reachable a morally dead structure.
- pop cannot produce space credits?!
A Space-Leaking Interleaving for pop?

let l = c in
match l with
...

• The sleeping thread maintains reachable a morally dead structure.

• pop cannot produce space credits?!
A Space-Leaking Interleaving for pop?

let l = c in
match l with
...

...
A Space-Leaking Interleaving for pop?

let l = c in
match l with
...

• The sleeping thread maintains reachable a morally dead structure.

• \texttt{pop} cannot produce space credits?!
A Space-Leaking Interleaving for pop?

The sleeping thread maintains reachable a morally dead structure.
- pop cannot produce space credits?!
How are Stop-The-World GC Implemented?

• Threads only stop at safe points to look up if a GC was requested.
• Safe points are inserted by the compiler.
• Only guarantee: no infinite loop without a safe point.

⇝ the space-leaking interleaving of \texttt{pop} does not happen!
How are Stop-The-World GC Implemented?

- Threads only stop at **safe points** to look up if a GC was requested.
- Safe points are inserted by the compiler.  
  Only guarantee: no infinite loop without a safe point.
How are Stop-The-World GC Implemented?

- Threads only stop at safe points to look up if a GC was requested.
- Safe points are inserted by the compiler. Only guarantee: no infinite loop without a safe point.

The GC will wait until all threads have reached a safe point. ~⇒ the space-leaking interleaving of pop does not happen!
GC-less Sections

- We do not control where safe points are inserted.
- We propose a syntax for GC-less sections: sections without safe points.
- The GC only runs when all threads are outside GC-less sections.
We do not control where safe points are inserted.
- We propose a syntax for GC-less sections: sections without safe points.
- The GC only runs when all threads are outside GC-less sections.

```
let rec pop s =
  begin_nogc ();
  let l = !s in
  match l with
  | nil -> assert false
  | v::l' ->
    if CAS s l l' then (end_nogc (); y) else (end_nogc (); pop s)
```
We introduce two new assertions: outside $\pi$ and inside $\pi T$.

The $T$ parameter represents temporary roots.
Reasoning About GC-less Sections

- We introduce two new assertions: outside \( \pi \) and inside \( \pi \ T \).
- The \( T \) parameter represents *temporary* roots.

\[
\text{BEGIN} \\
\{ \text{inside} \, \pi \emptyset * \Phi \} \pi: t \{ \Psi \} \\
\{ \text{outside} \, \pi * \Phi \} \pi: (\text{begin\_nogc}(); \, t) \{ \Psi \}
\]
Reasoning About GC-less Sections

- We introduce two new assertions: outside $\pi$ and inside $\pi T$.
- The $T$ parameter represents temporary roots.

\[
\begin{align*}
\text{Begin} & \quad \{ \text{inside } \pi \emptyset \ast \Phi \} \pi : t \quad \{ \Psi \} \\
\quad \{ \text{outside } \pi \ast \Phi \} \pi : (\text{begin\_nogc}(); t) \quad \{ \Psi \}
\end{align*}
\]

\[
\text{LoadAlt} \quad \{ \ell \mapsto \ell' \ast \text{inside } \pi T \} \pi : !\ell \quad \{ \ell'. \ell \mapsto \ell' \ast \text{inside } \pi (T \cup \{\ell'\}) \}
\]
Reasoning About GC-less Sections

- We introduce two new assertions: outside $\pi$ and inside $\pi T$.
- The $T$ parameter represents temporary roots.

\[
\text{BEGIN} \\
\{ \text{inside } \pi \emptyset \ast \Phi \} \pi : t \{ \Psi \} \\
\{ \text{outside } \pi \ast \Phi \} \pi : (\text{begin\_nogc }(); t ) \{ \Psi \}
\]

\[
\text{LOADALT} \\
\{ l \mapsto l' \ast \text{inside } \pi T \} \pi : !l \{ l' . l \mapsto l' \ast \text{inside } \pi (T \cup \{l'\}) \}
\]

\[
\text{END} \\
T \cap \text{locs}(t) = \emptyset \{ \text{outside } \pi \ast \Phi \} \pi : t \{ \Psi \} \\
\{ \text{inside } \pi T \ast \Phi \} \pi : (\text{end\_nogc }(); t ) \{ \Psi \}
\]
We present the first program logic for verifying heap space bounds of concurrent programs under GC.
Conclusion

We present the first program logic for verifying heap space bounds of concurrent programs under GC.

There is more!

- Closures
- Cycles
- Mechanization

Takeaway: Separation Logic can be used to reason about reachability.

Can we apply our ideas to other areas?
Conclusion

We present the first program logic for verifying heap space bounds of concurrent programs under GC.

There is more!

- Closures
- Cycles
- Mechanization

Takeaway: Separation Logic can be used to reason about reachability.
We present the first program logic for verifying heap space bounds of concurrent programs under GC.

There is more!

- Closures
- Cycles
- Mechanization

Takeaway: Separation Logic can be used to reason about reachability.

⇝ Can we apply our ideas to other areas?
Disentanglement [Westrick et al., 2019]:

Parallel tasks remain oblivious to each other’s allocations.
Disentanglement [Westrick et al., 2019]:

Parallel tasks remain oblivious to each other’s allocations.

![Diagram of disentanglement](image)
Disentanglement [Westrick et al., 2019]:

*Parallel tasks remain oblivious to each other’s allocations.*

- A disentangled program can be equipped with a fast GC.
- The MPL compiler assumes disentanglement.
- There was no static analysis for disentanglement!
- **POPL24, with Sam Westrick and Stephanie Balzer**

DisLog: A Separation Logic for Disentanglement
Thank you for your attention!

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

- An allocation can be stuck, in need of a GC, waiting for the other threads to get out of their GC-less section.
- A thread can be stuck only for a finite number of steps.
Soundness Theorem

- An allocation can be stuck, in need of a GC, waiting for the other threads to get out of their GC-less section.
- A thread can be stuck only for a finite number of steps.
### Soundness Theorem

- An allocation can be stuck, in need of a GC, waiting for the other threads to get out of their GC-less section.
- A thread can be stuck only for a finite number of steps.

\[
\begin{align*}
\text{Now} & \quad P \; c \\
\text{within} \; n \; P \; c
\end{align*}
\]

\[
\begin{align*}
\text{Next} & \quad (\exists c'. \; c \rightarrow c') \\
& \quad (\forall c'. \; (c \rightarrow c') \implies \text{within} \; n \; P \; c') \\
& \quad \text{within} \; (n + 1) \; P \; c
\end{align*}
\]

The main theorem is:

\[
\forall ts, \sigma. \; ([t], \emptyset) \rightarrow^* (ts, \sigma) \implies \forall \pi. \; \exists n. \; \text{within} \; n \; (\text{not\_stuck} \; \pi) \; (ts, \sigma)
\]
We handle cycles following the approach of Madiot and Pottier [2022].

\[
\begin{align*}
\forall \text{True} & \rightarrow \emptyset \ast 0 P \\
D \ast^n P & \rightarrow (\{\ell\} \cup D) \ast^{n+\text{size}(\vec{v})} P \quad \text{if } A \subseteq P \\
D \ast^n D & \Rightarrow \bigtriangleup n \ast (\ast \uparrow \ell) \quad \text{if } D \cap \text{locs}(t) = \emptyset
\end{align*}
\]

References

