A High-Level Separation Logic for Heap Space under Garbage Collection

Alexandre Moine  Arthur Charguéraud  François Pottier

Cambium Seminar, 28th November 2022
Verifying functional correctness is not enough!
Verifying functional correctness is not enough!

We have to take resources into account.
In a few Words

Verifying functional correctness is not enough!

We have to take resources into account.

We want to prevent the Dreaded Memory Leak!
In a few Words

Verifying functional correctness is not enough!

We have to take resources into account.
We want to prevent the Dreaded Memory Leak!

This work:

- A program logic to verify heap space bounds...
Verifying functional correctness is not enough!

We have to take resources into account.

We want to prevent the Dreaded Memory Leak!

This work:

- A program logic to verify heap space bounds...
- ...for a high-level language...
Verifying functional correctness is not enough!

We have to take resources into account.

We want to prevent the Dreaded Memory Leak!

This work:

- A program logic to verify heap space bounds...
- ...for a high-level language...
- ...equipped with a garbage collector.
Without a GC:

- **alloc** consumes space
- **free** produces space

With a GC:

- There is no syntax for deallocation.
- The GC can run at any time to deallocate blocks.
- The GC can deallocate only unreachable blocks.

To formally prove that some space is reclaimable by the GC:

- one has to prove that a block is unreachable,
- from the roots,
- following heap paths.
Formal Verification of Heap Space Bounds

Without a GC:

- alloc \textit{consumes} space
- free \textit{produces} space

With a GC:

- There is \textit{no syntax} for deallocation.
- The GC can run at any time to deallocates blocks.
- The GC can deallocate only \textit{unreachable blocks}.
Formal Verification of Heap Space Bounds

Without a GC:

- alloc *consumes* space
- free *produces* space

With a GC:

- There is no syntax for deallocation.
- The GC can run at any time to deallocates blocks.
- The GC can deallocate only *unreachable* blocks.

To formally prove that some space is *reclaimable* by the GC:

- one has to prove that a block is *unreachable*,
- from the roots,
- following heap paths.
A Motivating Example: mapsucc

let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)
A Motivating Example: mapsucc

```
let rec mapsucc (xs : int list) : int list =
    match xs with
    | [] -> []
    | y::ys -> (y+1)::(mapsucc ys)
```

In the presence of a GC, how much heap space does mapsucc need?
A Motivating Example: mapsucc

let rec mapsucc (xs : int list) : int list =
    match xs with
    | [] -> []
    | y::ys -> (y+1)::(mapsucc ys)

In the presence of a GC, how much heap space does mapsucc need?

It depends on the evaluation context!
A Motivating Example: \texttt{mapsucc}

\begin{verbatim}
let rec mapsucc (xs : int list) : int list =
    match xs with
    | []       -> []
    | y::ys    -> (y+1)::(mapsucc ys)
\end{verbatim}

In the presence of a GC, how much heap space does \texttt{mapsucc} need?

It depends on the evaluation context!

- If \texttt{xs} is \texttt{unreachable} from the evaluation context: $O(1)$
  
  The GC can claim the front cell of \texttt{xs} at each step.
let rec mapsucc (xs : int list) : int list =
  match xs with
  | [] -> []
  | y::ys -> (y+1)::(mapsucc ys)

In the presence of a GC, how much heap space does mapsucc need?

It depends on the evaluation context!

- If \(\text{xs}\) is unreachable from the evaluation context: \(O(1)\)
  The GC can claim the front cell of \(\text{xs}\) at each step.
- If \(\text{xs}\) is reachable from the evaluation context: \(O(\text{length } \text{xs})\)
SpaceLang, a low-level language by Madiot and Pottier [2022]. They

- Use **space credits** to account for free (reclaimable) space
  
  $\Diamond 1$
SpaceLang, a low-level language by Madiot and Pottier [2022]. They

- Use space credits to account for free (reclaimable) space
  
  \[ \diamond 1 \]

- Adapt pointed-by assertions to track predecessors
  
  \[ \ell \leftarrow_1 A \]
SpaceLang, a low-level language by Madiot and Pottier [2022]. They

- Use **space credits** to account for free (reclaimable) space
  \[ \diamond 1 \]
- Adapt **pointed-by** assertions to track predecessors
  \[ \ell \leftarrow_1 A \]
- Introduce a **logical deallocation rule**
  \[ \ell \leftarrow_1 [v_1, \ldots, v_n] \leftrightarrow_{1} \emptyset \quad \Rightarrow \quad \diamond n \leftrightarrow \top \ell \]
SpaceLang, a low-level language by Madiot and Pottier [2022]. They

- Use **space credits** to account for free (reclaimable) space
  \[ \diamond 1 \]
- Adapt **pointed-by** assertions to track predecessors
  \[ \ell \leftarrow_1 A \]
- Introduce a **logical deallocation rule**
  \[ \ell \leftrightarrow_1 [v_1, \ldots, v_n] \ast \ell \leftarrow_1 \emptyset \quad \Rightarrow \quad \diamond n \ast \dagger \ell \]
- Target an assembly-like language, with **explicit roots**
  - Trivializes the identification of roots
  - Non-standard syntax and semantics
Good ol’ POPL’22

SpaceLang, a low-level language by Madiot and Pottier [2022]. They

- Use **space credits** to account for free (reclaimable) space

\[
\begin{align*}
\ell & \mapsto 1 [v_1, \ldots, v_n] \quad \ast \quad \ell & \mapsto 1 \emptyset \quad \Rightarrow \quad \Diamond n \ast \dag \ell \\
\end{align*}
\]

- Adapt pointed-by assertions to track predecessors

\[
\begin{align*}
\ell & \mapsto_p \overline{\ell} \\
s & \mapsto \langle v \rangle \\
v & \leftarrow_q L \uplus \{s\} \\
v' & \leftarrow_{q'} L' \setminus \{s\}
\end{align*}
\]

- Introduce a logical deallocation rule

\[
\begin{align*}
\ast s & = \ast (r + o) \\
\ell & \mapsto \langle \ell \rangle \\
s & \mapsto \langle v \rangle \\
v & \leftarrow_q L \uplus \{s\} \\
v' & \leftarrow_{q'} L' \setminus \{s\}
\end{align*}
\]

- Target an assembly-like language, with **explicit roots**
  - Trivializes the identification of roots
  - Non-standard syntax and semantics
  - Polluted reasoning rules
Contributions

Building on the work of Madiot and Pottier, we present a logic for

- a high-level, ML-style, language,
- with closures.

Contributions

Building on the work of Madiot and Pottier, we present a logic for

- a high-level, ML-style, language,
- with closures.

Key solved challenges:

- Reasoning about roots in a garbage collected $\lambda$-calculus
- Reasoning about closures and the heap paths they introduce
Contributions

Building on the work of Madiot and Pottier, we present a logic for

- a high-level, ML-style, language,
- with closures.

Key solved challenges:

- Reasoning about roots in a garbage collected λ-calculus
- Reasoning about closures and the heap paths they introduce

Solved technical challenges:

- Modularity of specifications
- Theory and examples are fully mechanized in Coq on top of Iris
The Roots of the Problem

What are the roots considered by real-life GCs?
What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>let a = (ref 4) in let b = (ref 2) in !a + !b</code></td>
<td><code>∅</code></td>
</tr>
</tbody>
</table>
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let ( a = (\text{ref } 4) ) in let ( b = (\text{ref } 2) ) in (!a + !b)</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \rightarrow ) let ( a = \ell_a ) in let ( b = (\text{ref } 2) ) in (!a + !b)</td>
<td>{( \ell_a := 4 )}</td>
</tr>
</tbody>
</table>
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $a = (\text{ref } 4)$ in let $b = (\text{ref } 2)$ in $!a + !b$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\rightarrow$ let $a = \ell_a$ in let $b = (\text{ref } 2)$ in $!a + !b$</td>
<td>${\ell_a := 4}$</td>
</tr>
<tr>
<td>$\rightarrow$ let $b = (\text{ref } 2)$ in $!\ell_a + !b$</td>
<td>${\ell_a := 4}$</td>
</tr>
</tbody>
</table>
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{let } a = (\text{ref } 4) \text{ in let } b = (\text{ref } 2) \text{ in } !a + !b )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \rightarrow ) ( \text{let } a = \ell_a ) in ( \text{let } b = (\text{ref } 2) \text{ in } !a + !b )</td>
<td>( { \ell_a := 4 } )</td>
</tr>
<tr>
<td>( \rightarrow ) ( \text{let } b = (\text{ref } 2) \text{ in } !\ell_a + !b )</td>
<td>( { \ell_a := 4 } )</td>
</tr>
<tr>
<td>( \rightarrow ) ( \text{let } b = \ell_b ) in ( !\ell_a + !b )</td>
<td>( { \ell_a := 4, \ell_b := 2 } )</td>
</tr>
</tbody>
</table>
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let ( a = (\text{ref 4}) ) in let ( b = (\text{ref 2}) ) in ( !a + !b )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \rightarrow ) let ( a = \ell_a ) in let ( b = (\text{ref 2}) ) in ( !a + !b )</td>
<td>( { \ell_a := 4 } )</td>
</tr>
<tr>
<td>( \rightarrow ) let ( b = (\text{ref 2}) ) in ( !\ell_a + !b )</td>
<td>( { \ell_a := 4 } )</td>
</tr>
<tr>
<td>( \rightarrow ) let ( b = \ell_b ) in ( !\ell_a + !b )</td>
<td>( { \ell_a := 4, \ell_b := 2 } )</td>
</tr>
<tr>
<td>( \rightarrow ) ( !\ell_a + !\ell_b )</td>
<td>( { \ell_a := 4, \ell_b := 2 } )</td>
</tr>
</tbody>
</table>
**The Roots of the Problem**

What are the roots considered by real-life GCs?

**The Free Variable Rule (FVR)**

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $a = \text{(ref } 4\text{)}$ in let $b = \text{(ref } 2\text{)}$ in $!a + !b$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\rightarrow$ let $a = \ell_a$ in let $b = \text{(ref } 2\text{)}$ in $!a + !b$</td>
<td>${\ell_a := 4}$</td>
</tr>
<tr>
<td>$\rightarrow$ let $b = \text{(ref } 2\text{)}$ in $!\ell_a + !b$</td>
<td>${\ell_a := 4}$</td>
</tr>
<tr>
<td>$\rightarrow$ let $b = \ell_b$ in $!\ell_a + !b$</td>
<td>${\ell_a := 4, \ell_b := 2}$</td>
</tr>
<tr>
<td>$\rightarrow$ $!\ell_a + !\ell_b$</td>
<td>${\ell_a := 4, \ell_b := 2}$</td>
</tr>
<tr>
<td>$\rightarrow$ $4 + !\ell_b$</td>
<td>${\ell_a := 4, \ell_b := 2}$</td>
</tr>
</tbody>
</table>
The Roots of the Problem

What are the roots considered by real-life GCs?

The Free Variable Rule (FVR)

In a substitution-based semantics, the roots are the locations occurring in the term that remains to be evaluated.

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{let } a = (\text{ref } 4) \text{ in let } b = (\text{ref } 2) \text{ in } !a + !b</td>
<td>\emptyset</td>
</tr>
<tr>
<td>$\rightarrow$ let $a = \ell_a$ \text{ in let } b = (\text{ref } 2) \text{ in } !a + !b</td>
<td>{\ell_a := 4}</td>
</tr>
<tr>
<td>$\rightarrow$ let $b = (\text{ref } 2) \text{ in } !\ell_a + !b</td>
<td>{\ell_a := 4}</td>
</tr>
<tr>
<td>$\rightarrow$ let $b = \ell_b \text{ in } !\ell_a + !b</td>
<td>{\ell_a := 4, \ell_b := 2}</td>
</tr>
<tr>
<td>$\rightarrow$ $!\ell_a + !\ell_b</td>
<td>{\ell_a := 4, \ell_b := 2}</td>
</tr>
<tr>
<td>$\rightarrow$ $4 + !\ell_b</td>
<td>{\ell_a := 4, \ell_b := 2}</td>
</tr>
<tr>
<td>$\rightarrow$ $^*</td>
<td>6</td>
</tr>
<tr>
<td>Term</td>
<td>Heap</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>let $b = (\text{ref } 2) \text{ in } \ell_a + !b$</td>
<td>${ \ell_a := 4 }$</td>
</tr>
</tbody>
</table>

While reasoning about $(\text{ref } 2)$:

- The location $\ell_a$ is a root of the evaluation context.
- The GC cannot reclaim the space of $\ell_a$.

From a formal verification point of view:

- Roots may occur in the evaluation context.
- We need to prevent the logical deallocation of such invisible roots.
- Other visible roots may be found by inspecting the term under focus.
I am Root

<table>
<thead>
<tr>
<th>Term</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $b = (\text{ref 2})$ in $!\ell_a + !b$</td>
<td>${\ell_a := 4}$</td>
</tr>
</tbody>
</table>

While reasoning about $(\text{ref 2})$

- The location $\ell_a$ is a root of the evaluation context!
- The GC cannot reclaim the space of $\ell_a$. 
While reasoning about \((\text{ref} \ 2)\)

- The location \(\ell_a\) is a root of the evaluation context!
- The GC cannot reclaim the space of \(\ell_a\).

From a formal verification point of view:

- Roots may occur in the evaluation context.
- We need to prevent the logical deallocation of such invisible roots.
- Other visible roots may be found by inspecting the term under focus.
Introducing the *Stackable* $\ell p$ assertion to track invisible roots ($p \in (0, 1]$).

**Main property of the Stackable assertion**

*Stackable* $\ell 1$ asserts that $\ell$ is not an invisible root.
Introducing the *Stackable* $\ell \ p$ assertion to track invisible roots ($p \in (0, 1]$).

**Main property of the Stackable assertion**

*Stackable* $\ell \ 1$ asserts that $\ell$ is not an invisible root.

We refine the **FREE** rule of Madiot and Pottier.

\[
\ell \mapsto_1 [v_1, \ldots, v_n] \ast \ell \leftarrow_1 \emptyset \ast \lnot \ell \notin \text{locs}(t) \ast \text{Stackable } \ell \ 1 \quad \Rightarrow \quad \Diamond n \ast \dagger \ell
\]

- $\ell$ is not a visible root
- $\ell$ is not an invisible root
- $\ell$ is not pointed-by any reachable block
Handling Invisible Roots

The *Stackable* assertion is *splittable*

\[
\text{Stackable } \ell (p + q) \equiv \text{Stackable } \ell p \ast \text{Stackable } \ell q
\]

The construction let \( x = t_1 \) in \( t_2 \) may create invisible roots.

While reasoning about \( t_1 \), we *withhold* the *Stackable* assertions of \( \text{locs}(t_2) \).
The *Stackable* assertion is *splittable*

\[
Stackable \ell \ (p + q) \equiv Stackable \ell \ p \ast Stackable \ell \ q
\]

The construction let \( x = t_1 \) in \( t_2 \) may create invisible roots.

While reasoning about \( t_1 \), we **withhold** the *Stackable* assertions of \( \text{locs}(t_2) \).

The **LET** rule for a context with only one location:

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \\
\{ \Phi \} \ t_1 \ \{ \Psi' \} \quad &\forall v. \ \{ \Psi' \ v \} [v/x]t_2 \ \{ \Psi \} \\
\{ \} \quad &\Phi \ \text{let } x = t_1 \ \text{in } t_2 \ \{ \Psi \}
\end{align*}
\]
Handling Invisible Roots

The *Stackable* assertion is *splittable*

\[
\text{Stackable } \ell (p + q) \equiv \text{Stackable } \ell p \ast \text{Stackable } \ell q
\]

The construction let \( x = t_1 \) in \( t_2 \) may create invisible roots.

While reasoning about \( t_1 \), we *withhold* the *Stackable* assertions of \( \text{locs}(t_2) \).

The \textbf{Let} rule for a context with only one location:

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \\
\{ \Phi \} t_1 \{ \Psi \} &\quad \forall v. \{ \text{Stackable } \ell p \ast \Psi' \ v \} [v/x] t_2 \{ \Psi \} \\
\{ \text{Stackable } \ell p \ast \Phi \} &\text{ let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]
What is left?

In the rest of this talk

1. Other reasoning rules
2. Back to mapsucc
3. The Soundness Theorem
4. Closures
Pointed-by and *Stackable* assertions are created by $\text{ALLOC}$.

\[
\{ \Diamond n \} \text{ alloc } n \left\{ \lambda \ell. \begin{array}{c}
\ell \leftrightarrow_1 (\) \hspace{1cm} \\
\ell \leftrightarrow_1 \emptyset \ast \text{ Stackable } \ell \hspace{1cm}
\end{array} \right\}
\]
Pointed-by and *Stackable* assertions are created by \texttt{ALLOC}.

\[
\{ \Diamond n \} \text{ alloc } n \begin{array}{l}
\lambda \ell. \quad \ell \mapsto_1 ()^n
\end{array}
\]

\[
\ell \leftarrow_1 \emptyset \ast \textit{Stackable } \ell \ 1
\]

\texttt{LOAD} is the standard Separation Logic rule.

\[
0 \leq i < |\vec{w}|
\]

\[
\begin{array}{l}
\{ \ell \mapsto_p \vec{w} \} \quad \ell[i] \left\{ \lambda v. \quad v = \vec{w}(i) \downarrow \right\}
\end{array}
\]

\[
\ell \mapsto p \vec{w}
\]
Store is more complex: it modifies heap antecedents.

\[
\begin{align*}
0 \leq i < |\vec{w}| & \quad \vec{w}(i) = v \\
\ell \mapsto_1 \vec{w} & \quad v'[\leftarrow_p A] \quad \ell[i] \leftarrow v' \\
\lambda_. \quad v'[\leftarrow_p A \cup \{+\ell\}] & \quad v \leftarrow_0 \{-\ell\}
\end{align*}
\]

Proof Pearl
Two specifications for \texttt{map\_succ}

Pointed-by and \textit{Stackable} assertions often go together

$$\ell \leftarrow_p A \triangleq \ell \leftarrow_p A \ast \text{Stackable} \ell \ p$$

Split rule: $$\ell \leftarrow_{(p_1+p_2)} (A_1 \uplus A_2) \equiv \ell \leftarrow_{p_1} A_1 \ast \ell \leftarrow_{p_2} A_2$$
Two specifications for \texttt{map\texttt{suc}}c

Pointed-by and \textit{Stackable} assertions often go together

\[
\ell \leftarrow_p A \triangleq \ell \leftarrow_p A \ast \text{Stackable} \ell \ p
\]

Split rule:

\[
\ell \leftarrow_{(p_1+p_2)} (A_1 \uplus A_2) \equiv \ell \leftarrow_{p_1} A_1 \ast \ell \leftarrow_{p_2} A_2
\]

If \(\ell\) is \textcolor{red}{unreachable} from the evaluation context:

\[
\left\{ \text{List } L \ell \ast \ell \leftarrow_1 \emptyset \right\} \text{ \texttt{map\texttt{suc}}c \ell \left\{ \lambda \ell'. \text{List } (\text{map } (+1) L) \ell' \ast \ell' \leftarrow_1 \emptyset \right\}
\]
Two specifications for mapsucc

Pointed-by and *Stackable* assertions often go together

\[ \ell \leftarrow_p A \triangleq \ell \leftarrow_p A \ast Stackable \ell p \]

Split rule:

\[ \ell \leftarrow (p_1+p_2) (A_1 \uplus A_2) \equiv \ell \leftarrow_{p_1} A_1 \ast \ell \leftarrow_{p_2} A_2 \]

If \( \ell \) is unreachable from the evaluation context:

\[
\left\{ \text{List } L \ell \ast \ell \leftarrow_1 \emptyset \right\} \text{mapsucc } \ell \left\{ \lambda \ell'. \text{List } (\text{map } (+1) L) \ell' \ast \ell' \leftarrow_1 \emptyset \right\}
\]

If \( \ell \) may be reachable:

\[
\left\{ \text{List } L \ell \ast \ell \leftarrow_p A \right\} \text{mapsucc } \ell \left\{ \lambda \ell'. \text{List } (\text{map } (+1) L) \ell' \ast \ell' \leftarrow_1 \emptyset \right\}
\]

\[
\diamond (3 \times \text{length } L)
\]
The Soundness Theorem

Our semantics

- is parameterized by a **maximal** heap size $S$
- **interleaves** reduction steps and GC steps
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.
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 $\{\Diamond S\} t \{\Psi\}$ holds, then $t$ cannot reach a stuck configuration.

Reformulation: the live heap space of any execution of $t$ cannot exceed $S$. 
We encode closures as derived constructions using closure conversion

- closure creation and call are not in the syntax,
- but we provide macros implementing them,
- and provide reasoning rules about those macros!
Closures

We encode closures as derived constructions using closure conversion

- closure creation and call are not in the syntax,
- but we provide macros implementing them,
- and provide reasoning rules about those macros!

<table>
<thead>
<tr>
<th>A closure is</th>
<th>Allocating a closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>a heap allocated block</td>
<td>consumes space credits</td>
</tr>
<tr>
<td>pointing to its environment</td>
<td>updates pointed-by assertions</td>
</tr>
</tbody>
</table>
Closures

We encode closures as derived constructions using closure conversion

- closure creation and call are not in the syntax,
- but we provide macros implementing them,
- and provide reasoning rules about those macros!

<table>
<thead>
<tr>
<th>A closure is</th>
<th>Allocating a closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>a heap allocated block</td>
<td>consumes space credits</td>
</tr>
<tr>
<td>pointing to its environment</td>
<td>updates pointed-by assertions</td>
</tr>
</tbody>
</table>

I will show you very simple closures

- non-recursive
- no argument
- environment of size 1
let counter () =
  let r = ref 0 in
  ((fun () -> incr r), (fun () -> !r))
Playing with Closures: Counter Objects

```ocaml
let counter () =
  let r = ref 0 in
  ((fun () -> incr r), (fun () -> !r))
```

\[
\begin{align*}
\{\text{Counter } i \ k \ n\} & \quad (i())_{\text{clo}} & \{\lambda_. \ \text{Counter } i \ k \ (n+1)\} \\
\{\text{Counter } i \ k \ n\} & \quad (g())_{\text{clo}} & \{\lambda m. \ [m = n \downarrow] \ast \text{Counter } i \ k \ n\}
\end{align*}
\]
let counter () =
    let r = ref 0 in
    ((fun () -> incr r), (fun () -> !r))
Specifying Closures

We introduce the $Spec$ assertion

$$Spec \; E \; P \; f$$
We introduce the \textit{Spec} assertion

\[ \text{Spec } E \ P \ f \]

Definition of the \textit{Counter} predicate

\[
\text{Counter } i \ g \ n \triangleq \exists \ell. \left \{ \begin{array}{l}
\ell \mapsto [n] \\
\text{Spec } [(\ell, \frac{1}{2})] (P_{\text{incr } \ell}) \ i \\
\text{Spec } [(\ell, \frac{1}{2})] (P_{\text{get } \ell}) \ g
\end{array} \right \} \]

The specification predicate \( P \) abstracts away the closure code.
Specifying Closures

We introduce the \( \text{Spec} \) assertion

\[
\text{Spec} \ E \ P \ f
\]

Definition of the \textit{Counter} predicate

\[
\text{Counter} \ i \ g \ n \triangleq \exists \ell. \begin{cases} 
\ell \mapsto [n] & \text{spec} \\
\text{Spec} \ [(\ell, \frac{1}{2})] \ (P_{\text{incr}} \ell) \ i & \text{spec} \\
\text{Spec} \ [(\ell, \frac{1}{2})] \ (P_{\text{get}} \ell) \ g & \text{spec}
\end{cases}
\]

The specification predicate \( P \) \textbf{abstracts away} the closure code.

\[
P_{\text{incr}} \ell \triangleq \lambda u. \ \forall \ n. \ \{ \ell \mapsto [n] \} \ u \ \{ \lambda \_. \ \ell \mapsto [n + 1] \}
\]

\[
P_{\text{get}} \ell \triangleq \lambda u. \ \forall \ n. \ \{ \ell \mapsto [n] \} \ u \ \{ \lambda m. \ \exists m = n \ \* \ \ell \mapsto [n] \}
\]
Closure Creation

Closure creation is subtle to reason about

- the semantics is substitution-based,
- hence, the environment is substituted
- hence, we need to specify a substitution of the environment!
Closure creation is subtle to reason about

- the semantics is substitution-based,
- hence, the environment is substituted
- hence, we need to specify a substitution of the environment!

\[
\begin{align*}
fv(t) &= \{r\} \\
E &= [(\ell, p)] \\
P \left(\left[\ell/\ell\right] t\right) \\
\left\{\Diamond 2 \ast \ell \leftarrow_p \emptyset\right\} \left[\ell/\ell\right] (\lambda_{clo}() \cdot t) \\
&\quad \left\{\lambda f. \text{Spec} E P f \ast f \leftarrow_1 \emptyset\right\}
\end{align*}
\]
The Call of a Closure

Reasoning about a call:

\[(\forall u. P u \rightarrow \{\Phi\} u \{\Psi\})\]

\[\{\text{Spec } E P f \ast \Phi\} (f())_\text{clo} \\{\lambda v. \text{Spec } E P f \ast \Psi v\}\]

term describing the call

The general case is challenging, as a closure may:

- Call itself.
- Become unreachable just after a call, and self-destruct.
The Call of a Closure

Reasoning about a call:

\[
(\forall u . P u \rightarrow \{\Phi\} u \{\Psi\})
\]

\[
\{\text{Spec } E P f \ast \Phi\} (f ())_{\text{clo}} \{\lambda v. \text{Spec } E P f \ast \Psi v\}
\]

The general case is **challenging**, as a closure may:

- **Call itself.**
- **Become unreachable just after a call.**
The Call of a Closure

Reasoning about a call:

\[
(\forall u \cdot P u \rightarrow \{\Phi\} u \{\Psi\})
\]

\[
\{\text{Spec } E \cdot P \cdot (f \cdot \Phi)\} (f ())_{\text{clo}} \{\lambda v . \text{Spec } E \cdot P \cdot f \cdot \Psi \cdot v\}
\]

The general case is challenging, as a closure may:

- Call itself.
- Become unreachable just after a call, and self-destruct.
Read the Paper

- Recursive and self-destructive closures
- Simplified handling of *Stackable* assertions
- Simplified mode without logical free
- CPS-style example with `append`
- Amortized analysis with rational space credits (list of arrays)
- Illustration of modularity with stacks
- Fun technical contributions: fraction zero and signed multisets
Read the Paper

• Recursive and self-destructive closures
• Simplified handling of Stackable assertions
• Simplified mode without logical free
• CPS-style example with appends
• Amortized analysis with rational space credits (list of arrays)
• Illustration of modularity with stacks
• Fun technical contributions: fraction zero and signed multisets

A High-Level Separation Logic for Heap Space under Garbage Collection

Alexandre Moine, Arthur Charguéraud, and François Pottier.

We present a Separation Logic with space credits for reasoning about heap space in a sequential call-by-value calculus equipped with garbage collection and mutable state. A key challenge is to design sound and modular, lightweight mechanisms for establishing the unreachability of a block. Prior work in this setting uses assembly-like assertions to keep track of the unreachability of the garbage collector considers unreachable objects. For this purpose, we propose novel "stackable" assertions, which keep track of the existence of reachable objects while not explicitly recording their origin. Furthermore, we explain how to reason about stack-to-heap pointers without explicitly recording their origin.

CPS Concepts: • Theory of computation • Separation logic: Program verification

Read the Paper

Recursive and self-destructive closures
Simplified handling of Stackable assertions
Simplified mode without logical free
CPS-style example with appends
Amortized analysis with rational space credits (list of arrays)
Illustration of modularity with stacks
Fun technical contributions: fraction zero and signed multisets
Conclusion

We present a logic targeting

- a high-level language,
- with closures,
- equipped with a garbage collector,
- that obeys the free variable rule,
- and is fully mechanized in Coq on top of Iris.

Future work:

- Concurrency, lock-free data structures (ongoing)
- Weak pointers and ephemerons
- Links with the formal cost semantics of CakeML
We present a logic targeting

- a high-level language,
- with closures,
- equipped with a garbage collector,
- that obeys the free variable rule,
- and is fully mechanized in Coq on top of Iris.

Future work:

- Concurrency, lock-free data structures (ongoing)
- Weak pointers and ephemerons
- Links with the formal cost semantics of CakeML
Thank you for your attention!

alexandre.moine [at] inria.fr
arthur.chargueraud [at] inria.fr
francois.pottier [at] inria.fr
List $L \ell \ast \neg \ell \notin \text{locs}(t) \updownarrow \ast \ell \leftarrow_1 \emptyset \implies \diamond(3 \times \text{length } L)$
Triples with Souvenir

*Stackable* assertions are easy to manage in practice.

Introducing triples with souvenir $\langle R \rangle \{ \Phi \} t \{ \Psi \}$

“Give a *Stackable* assertion once and that’s it”
Triples with Souvenir

_STACKABLE assertions are easy to manage in practice.

Introducing _triples with souvenir_ \( \langle R \rangle \{ \Phi \} t \{ \Psi \} \)

"Give a _Stackable assertion once and that’s it""

\[
\text{LET ADD SOUVENIR}
\]

\[
\begin{align*}
\text{locs}(t_2) &= \{\ell\} \\
\langle R \cup \{\ell\} \rangle \{\Phi\} t_1 \{\Psi'\} &\quad \forall v. \langle R \rangle \{\text{Stackable } \ell \ p \ast \Psi' \ v\} [v/x] t_2 \{\Psi\} \\
\langle R \rangle \{\text{Stackable } \ell \ p \ast \Phi\} &\quad \text{let } x = t_1 \text{ in } t_2 \{\Psi\}
\end{align*}
\]
Triples with Souvenir

Stackable assertions are easy to manage in practice.

Introducing triples with souvenir $\langle R \rangle \{ \Phi \} t \{ \Psi \}$

“Give a Stackable assertion once and that’s it”

\[\text{LetAddSouvenir} \]

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \\
\langle R \cup \{ \ell \} \rangle \{ \Phi \} t_1 \{ \Psi' \} &\quad \forall v. \langle R \rangle \{ \text{Stackable } \ell \ p \ast \Psi' \ v \} [v/x] t_2 \{ \Psi \} \\
\langle R \rangle \{ \text{Stackable } \ell \ p \ast \Phi \} &\quad \text{let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]

\[\text{LetInSouvenir} \]

\[
\begin{align*}
\text{locs}(t_2) &= \{ \ell \} \quad \ell \in R \\
\langle R \rangle \{ \Phi \} t_1 \{ \Psi' \} &\quad \forall v. \langle R \rangle \{ \Psi' \ v \} [v/x] t_2 \{ \Psi \} \\
\langle R \rangle \{ \Phi \} &\quad \text{let } x = t_1 \text{ in } t_2 \{ \Psi \}
\end{align*}
\]
The NoFree Mode

- **Stackable** assertions are needed to prevent logical deallocation.
- If the user pledges to not deallocate, no tracking is needed.

\[
\text{LetNoFree} \quad \langle \bot \rangle \{ \Phi \} t_1 \{ \Psi' \} \quad \forall v. \quad \langle R \rangle \{ \Psi' \ v \} [v/x] t_2 \{ \Psi \}
\]

\[
\frac{\langle R \rangle \{ \Phi \} \text{let } x = t_1 \text{ in } t_2 \{ \Psi \}}{}
\]
A new consequence rule

New ghost update parameterized by the visible roots.

\[
\Phi \Rightarrow_{\text{locs}(t)} \Phi' \quad \{ \Phi' \} \ t \ \{ \Psi \} \\
\{ \Phi \} \ t \ \{ \Psi \}
\]

Our logical FREE rule.

\[
\ell \mapsto_1 [v_1, \ldots, v_n] \ast \ell \leftarrow_1 \emptyset \ast \ell \notin V^{-} \ast \text{Stackable } \ell \ 1 \quad \Rightarrow_{\vee} \quad \diamond n \ast \dagger \ell
\]