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

Alexandre Moine   Arthur Charguéraud   François Pottier

Cambium Seminar, 28th November 2022

In a few Words

Verifying functional correctness is not enough!

In a few Words

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...

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...
- ...for a high-level language...

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...
- ...for a high-level language...
- ...equipped with a garbage collector.

Formal Verification of Heap Space Bounds

Without a GC:

- `alloc` consumes space
- `free` produces space

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.

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: `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!

- If `xs` is unreachable from the evaluation context: $O(1)$
  The GC can claim the front cell of `xs` at each step.

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!

- If `xs` is unreachable from the evaluation context: $O(1)$
  The GC can claim the front cell of `xs` at each step.
- If `xs` is reachable from the evaluation context: $O(\text{length } xs)$

Good ol' POPL'22

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

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

$$\diamond 1$$

Good ol' POPL'22

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

$$\ell \hookleftarrow_1 A$$

Good ol' POPL'22

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

$$\ell \leftarrow\!\!\dashv_1 A$$

- Introduce a logical deallocation rule

$$\ell \mapsto_1 [v_1, \ldots, v_n] \ast \ell \leftarrow\!\!\dashv_1 \emptyset \quad \Rrightarrow \quad \diamond n \ast \dagger \ell$$

Good ol' POPL'22

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

$$\ell \hookleftarrow_1 A$$

- Introduce a logical deallocation rule

$$\ell \mapsto_1 [v_1, \ldots, v_n] \; * \; \ell \hookleftarrow_1 \emptyset \quad \Rrightarrow \quad \diamond n \; * \; \dagger \ell$$

- 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 (reclai

LOAD

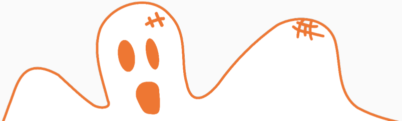
$$\frac{\vec{v}(o) = v}{\left\{\begin{array}{c} r \mapsto \langle \ell \rangle \\ \ell \mapsto_p \vec{v} \\ s \mapsto \langle v' \rangle \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{array}\right\} *s = [*r + o] \left\{\begin{array}{c} r \mapsto \langle \ell \rangle \\ \ell \mapsto_p \vec{v} \\ s \mapsto \langle v \rangle \\ v \hookleftarrow_q L \uplus \{s\} \\ v' \hookleftarrow_{q'} L' \setminus \{s\} \end{array}\right\}}$$

STORE

$$\frac{\vec{v}(o) = v}{\left\{\begin{array}{c} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 \vec{v} \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{array}\right\} [*s + o] = *r \left\{\begin{array}{c} s \mapsto \langle \ell \rangle \\ r \mapsto \langle v' \rangle \\ \ell \mapsto_1 [o := v']\vec{v} \\ v \hookleftarrow_q L \setminus \{\ell\} \\ v' \hookleftarrow_{q'} L' \uplus \{\ell\} \end{array}\right\}}$$

$$\ell \mapsto_1 [v_1, \dots, v_n] \; * \; \ell \hookleftarrow_1 \emptyset \quad \Rrightarrow \quad \diamond n \; * \; \dagger \ell$$

- 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 λ-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?

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.

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.

Term	Heap
let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$

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.

	Term	Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a := 4\}$

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.

	Term	Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$

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.

	Term	Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a := 4, \ell_b := 2\}$

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.

	Term	Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a := 4, \ell_b := 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a := 4, \ell_b := 2\}$

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.

	Term	Heap
	let $a = (\mathit{ref}\ 4)$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\emptyset$
$\longrightarrow$	let $a = \ell_a$ in let $b = (\mathit{ref}\ 2)$ in $!a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	let $b = \ell_b$ in $!\ell_a + !b$	$\{\ell_a := 4, \ell_b := 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a := 4, \ell_b := 2\}$
$\longrightarrow$	$4 + !\ell_b$	$\{\cancel{\ell_a := 4}, \ell_b := 2\}$

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.

	Term	Heap
	$\text{let } a = (\mathit{ref}\ 4) \text{ in let } b = (\mathit{ref}\ 2) \text{ in } !a + !b$	$\emptyset$
$\longrightarrow$	$\text{let } a = \ell_a \text{ in let } b = (\mathit{ref}\ 2) \text{ in } !a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	$\text{let } b = (\mathit{ref}\ 2) \text{ in } !\ell_a + !b$	$\{\ell_a := 4\}$
$\longrightarrow$	$\text{let } b = \ell_b \text{ in } !\ell_a + !b$	$\{\ell_a := 4, \ell_b := 2\}$
$\longrightarrow$	$!\ell_a + !\ell_b$	$\{\ell_a := 4, \ell_b := 2\}$
$\longrightarrow$	$4 + !\ell_b$	$\{\cancel{\ell_a := 4}, \ell_b := 2\}$
$\longrightarrow^*$	6	$\{\cancel{\ell_a := 4}, \cancel{\ell_b := 2}\}$

I am Root

Term	Heap
let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$

I am Root

Term	Heap
let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$

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

- The location ℓ_a is a root of the evaluation context!
- The GC cannot reclaim the space of ℓ_a.

I am Root

Term	Heap
let $b = (\mathit{ref}\ 2)$ in $!\ell_a + !b$	$\{\ell_a := 4\}$

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

- The location ℓ_a is a root of the evaluation context!
- The GC cannot reclaim the space of ℓ_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.

Free as a Ghost Update

Introducing the *Stackable* ℓ p assertion to track invisible roots ($p \in (0, 1]$).

Main property of the Stackable assertion

Stackable ℓ 1 asserts that ℓ is not an invisible root.

Free as a Ghost Update

Introducing the *Stackable* ℓ p assertion to track invisible roots ($p \in (0,1]$).

Main property of the Stackable assertion

Stackable ℓ 1 asserts that ℓ is not an invisible root.

We refine the FREE rule of Madiot and Pottier.

ℓ is not a visible root

ℓ is not an invisible root

$$\ell \mapsto_1 [v_1, \ldots, v_n] * \ell \hookleftarrow_1 \emptyset * \ulcorner \ell \notin locs(t) \urcorner * \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow \quad \diamond n * \dagger \ell$$

ℓ is not pointed-by any reachable block

Handling Invisible Roots

The *Stackable* assertion is splittable

$$\mathit{Stackable}\;\ell\;(p+q)\;\equiv\;\mathit{Stackable}\;\ell\;p\;*\;\mathit{Stackable}\;\ell\;q$$

The construction let $x = t_1$ in t_2 may create invisible roots.

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

Handling Invisible Roots

The *Stackable* assertion is splittable

$$Stackable\ \ell\ (p + q)\ \equiv\ Stackable\ \ell\ p\ *\ Stackable\ \ell\ q$$

The construction let $x = t_1$ in t_2 may create invisible roots.

While reasoning about t_1, we withhold the *Stackable* assertions of $locs(t_2)$.

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

$$\frac{\{\Phi\}\ t_1\ \{\Psi'\} \qquad \begin{array}{c} locs(t_2) = \{\ell\} \\ \forall v.\ \{\qquad\qquad\qquad\qquad \Psi'\ v\}\ [v/x]t_2\ \{\Psi\} \end{array}}{\{\qquad\qquad\qquad\qquad\qquad \Phi\}\ \text{let } x = t_1 \text{ in } t_2\ \{\Psi\}}$$

Handling Invisible Roots

The *Stackable* assertion is splittable

$$\mathit{Stackable}\;\ell\;(p+q)\;\equiv\;\mathit{Stackable}\;\ell\;p\;*\;\mathit{Stackable}\;\ell\;q$$

The construction let $x = t_1$ in t_2 may create invisible roots.

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

The LET rule for a context with only one location:

$$\frac{\begin{array}{c}\mathit{locs}(t_2) = \{\ell\} \\ \{\Phi\}\; t_1\; \{\Psi'\} \qquad \forall v.\, \{\mathit{Stackable}\;\ell\;p\;*\;\Psi'\;v\}\;[v/x]t_2\;\{\Psi\}\end{array}}{\{\mathit{Stackable}\;\ell\;p\;*\;\Phi\}\;\text{let}\;x = t_1\;\text{in}\;t_2\;\{\Psi\}}$$

What is left?

In the rest of this talk

1. Other reasoning rules
2. Back to `mapsucc`
3. The Soundness Theorem
4. Closures

Allocation and Load

Pointed-by and *Stackable* assertions are created by ALLOC.

$$\left\{\diamond n\right\}\ \mathsf{alloc}\ n\ \left\{\lambda\ell.\ \begin{array}{c}\ell \mapsto_1 ()^n \\ \ell \hookleftarrow_1 \emptyset\ *\ \mathit{Stackable}\ \ell\ 1\end{array}\right\}$$

Allocation and Load

Pointed-by and *Stackable* assertions are created by ALLOC.

$$\left\{\diamond n\right\}\ \mathsf{alloc}\ n\ \left\{\lambda\ell.\ \begin{array}{c}\ell \mapsto_1 ()^n \\ \ell \hookleftarrow_1 \emptyset \ *\ \mathit{Stackable}\ \ell\ 1\end{array}\right\}$$

LOAD is the standard Separation Logic rule.

$$\frac{0 \leq i < |\vec{w}|}{\left\{\ell \mapsto_p \vec{w}\right\}\ \ell[i]\ \left\{\lambda v.\ \begin{array}{c}\ulcorner v = \vec{w}(i) \urcorner \\ \ell \mapsto_p \vec{w}\end{array}\right\}}$$

Store

$\textsc{Store}$ is more complex: it modifies heap antecedents.

$$\dfrac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix} \ell \mapsto_1 \vec{w} \\ v' \hookleftarrow_p A \\ \end{matrix}\right\} \ \ell[i] \leftarrow v' \ \left\{\begin{matrix} \ell \mapsto_1 [i := v']\vec{w} \\ \lambda\_.\ v' \hookleftarrow_p A \uplus \{+\ell\} \\ \boxed{v \hookleftarrow_0 \{-\ell\}} \end{matrix}\right\}}$$

Proof Pearl

Two specifications for `mapsucc`

Pointed-by and *Stackable* assertions often go together

$$\ell \hookleftarrow_p A \quad \triangleq \quad \ell \leftarrowtail_p A \ * \ \mathit{Stackable}\ \ell\ p$$

Split rule: $\ell \hookleftarrow_{(p_1+p_2)} (A_1 \uplus A_2) \quad \equiv \quad \ell \hookleftarrow_{p_1} A_1 \ * \ \ell \hookleftarrow_{p_2} A_2$

Two specifications for `mapsucc`

Pointed-by and *Stackable* assertions often go together

$$\ell \hookleftarrow_p A \quad \triangleq \quad \ell \leftarrowtail_p A \ * \ \mathit{Stackable}\ \ell\ p$$

Split rule: $$\ell \hookleftarrow_{(p_1+p_2)} (A_1 \uplus A_2) \quad \equiv \quad \ell \hookleftarrow_{p_1} A_1 \ * \ \ell \hookleftarrow_{p_2} A_2$$

If ℓ is unreachable from the evaluation context:

$$\left\{ \mathit{List}\ L\ \ell \ * \ \ell \hookleftarrow_1 \emptyset \right\} \mathit{mapsucc}\, \ell \left\{ \lambda \ell'.\ \mathit{List}\ (\mathit{map}\,(+1)\, L)\ \ell' \ * \ \ell' \hookleftarrow_1 \emptyset \right\}$$

Two specifications for `mapsucc`

Pointed-by and *Stackable* assertions often go together

$$\ell \hookleftarrow_p A \quad \triangleq \quad \ell \mapsfrom_p A \;*\; \mathit{Stackable}\;\ell\;p$$

Split rule:
$$\ell \hookleftarrow_{(p_1+p_2)} (A_1 \uplus A_2) \quad \equiv \quad \ell \hookleftarrow_{p_1} A_1 \;*\; \ell \hookleftarrow_{p_2} A_2$$

If ℓ is unreachable from the evaluation context:

$$\left\{\mathit{List}\;L\;\ell \;*\; \ell \hookleftarrow_1 \emptyset\right\}\; \mathit{mapsucc}\;\ell\; \left\{\lambda\ell'.\; \mathit{List}\;(\mathit{map}\,(+1)\,L)\;\ell' \;*\; \ell' \hookleftarrow_1 \emptyset\right\}$$

If ℓ may be reachable:

$$\left\{\begin{matrix}\mathit{List}\;L\;\ell \;*\; \ell \hookleftarrow_p A \\ \diamond(3 \times \mathrm{length}\;L)\end{matrix}\right\}\; \mathit{mapsucc}\;\ell\; \left\{\lambda\ell'.\; \begin{matrix}\mathit{List}\;L\;\ell \;*\; \ell \hookleftarrow_p A \\ \mathit{List}\;(\mathit{map}\,(+1)\,L)\;\ell' \;*\; \ell' \hookleftarrow_1 \emptyset\end{matrix}\right\}$$

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.

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!

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!

A closure is	Allocating a closure
a heap allocated block	consumes space credits
pointing to its environment	updates pointed-by assertions

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!

A closure is	Allocating a closure
a heap allocated block	consumes space credits
pointing to its environment	updates pointed-by assertions

I will show you very simple closures

- non-recursive
- no argument
- environment of size 1

Playing with Closures: Counter Objects

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

Playing with Closures: Counter Objects

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

$$\left\{ Counter\ i\ g\ n \right\} \quad (i\ ())_{\mathsf{clo}} \quad \left\{ \lambda\_.\ Counter\ i\ g\ (n+1) \right\}$$
$$\left\{ Counter\ i\ g\ n \right\} \quad (g\ ())_{\mathsf{clo}} \quad \left\{ \lambda m.\ \ulcorner m = n \urcorner * Counter\ i\ g\ n \right\}$$

Playing with Closures: Counter Objects

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

$$\left\{\mathit{Counter}\ i\ g\ n\right\} \quad (i\ ())_{\mathsf{clo}} \quad \left\{\lambda\_.\ \mathit{Counter}\ i\ g\ (n+1)\right\}$$

$$\left\{\mathit{Counter}\ i\ g\ n\right\} \quad (g\ ())_{\mathsf{clo}} \quad \left\{\lambda m.\ \ulcorner m = n \urcorner * \mathit{Counter}\ i\ g\ n\right\}$$

$$\left\{\diamond 7\right\}\ (\mathit{counter}\ ())_{\mathsf{ptr}}\ \left\{\lambda \ell.\ \begin{array}{c} \exists i, g.\ \ell \mapsto [i; g] * \ell \hookleftarrow_1 \emptyset \\ i \hookleftarrow_1 \{\ell\} * g \hookleftarrow_1 \{\ell\} \\ \mathit{Counter}\ i\ g\ 0 \end{array}\right\}$$

Specifying Closures

We introduce the *Spec* assertion

$$Spec\ E\ P\ f$$

Specifying Closures

We introduce the *Spec* assertion

$$\mathit{Spec}\ E\ P\ f$$

Definition of the *Counter* predicate

$$\mathit{Counter}\ i\ g\ n \quad \triangleq \quad \exists \ell. \begin{cases} \ell \mapsto [n] & * \\ \mathit{Spec}\ [(\ell, \tfrac{1}{2})]\ (P_{\mathit{incr}}\ \ell)\ i & * \\ \mathit{Spec}\ [(\ell, \tfrac{1}{2})]\ (P_{\mathit{get}}\ \ell)\ g & \end{cases}$$

Specifying Closures

We introduce the *Spec* assertion

$$\mathit{Spec}\ E\ P\ f$$

Definition of the *Counter* predicate

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

The specification predicate P abstracts away the closure code.

$$\begin{aligned} P_{incr}\ \ell &\triangleq \lambda u.\, \forall\, n.\, \{\ell \mapsto [n]\}\ u\ \{\lambda\_.\, \ell \mapsto [n+1]\} \\ P_{get}\ \ell &\triangleq \lambda u.\, \forall\, n.\, \{\ell \mapsto [n]\}\ u\ \{\lambda m.\, \ulcorner m = n \urcorner * \ell \mapsto [n]\} \end{aligned}$$

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

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!

$$\frac{fv(t) = \{r\} \qquad E = [(\ell, p)] \qquad P\,([\ell/r]t)}{\left\{\diamond 2 \;*\; \ell \leftarrow\!\!\dashv_p \emptyset\right\}\;[\ell/r]\,(\lambda_{clo}\,().\;t)\;\left\{\lambda f.\;\mathit{Spec}\;E\;P\;f \;*\; f \hookleftarrow_1 \emptyset\right\}}$$

The Call of a Closure

Reasoning about a call:

term describing the call

$$\frac{(\forall\, \boxed{u}\,.\; P\; u \;-\!\!*\; \{\Phi\}\; u\; \{\Psi\})}{\{\mathit{Spec}\; E\; P\; f * \Phi\}\; (f\; ())_{\mathsf{clo}}\; \{\lambda v.\, \mathit{Spec}\; E\; P\; f * \Psi\; v\}}$$

The Call of a Closure

Reasoning about a call:

term describing the call

$$\frac{(\forall\, \boxed{u}\,.\ P\ u\ \twoheadrightarrow\ \{\Phi\}\ u\ \{\Psi\})}{\{Spec\ E\ P\ f * \Phi\}\ (f\ ())_{\mathsf{clo}}\ \{\lambda v.\ Spec\ E\ P\ f * \Psi\ v\}}$$

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:

term describing the call

$$\frac{(\forall\, u\,.\; P\; u \;\mathrel{-\!\!*}\; \{\Phi\}\; u\; \{\Psi\})}{\{\mathit{Spec}\; E\; P\; f * \Phi\}\; (f\; ())_{\mathsf{clo}}\; \{\lambda v.\; \mathit{Spec}\; E\; P\; f * \Psi\; v\}}$$

The general case is challenging, as a closure may:

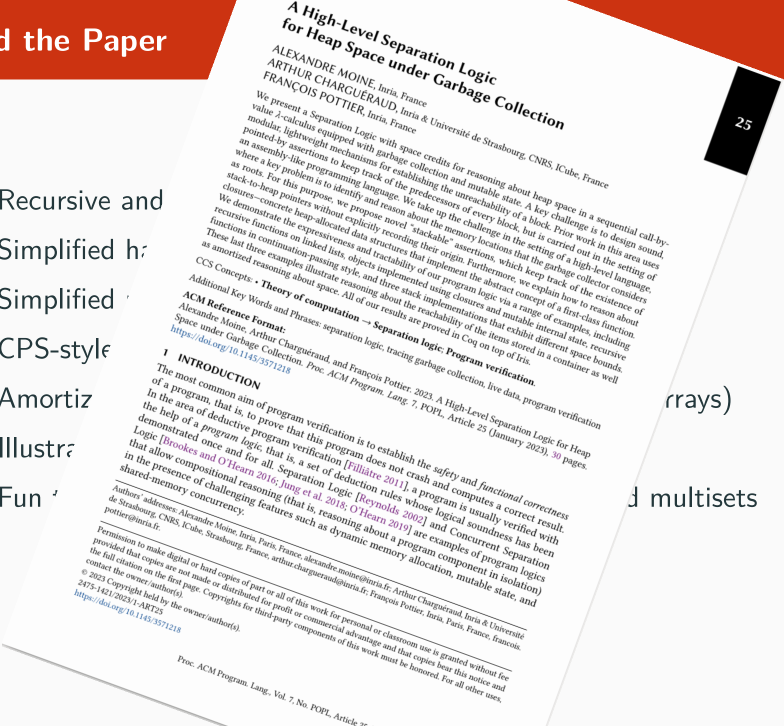
- Call itself.
- Become unreachable just after a call, and self-destruct.

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
- Simplified ha
- Simplified
- CPS-style
- Amortiz rrays)
- Illustra
- Fun d multisets



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

25

ALEXANDRE MOINE, Inria, France
ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
FRANÇOIS POTTIER, Inria, France

We present a Separation Logic with space credits for reasoning about heap space in a sequential call-by-value λ-calculus equipped with garbage collection and mutable state. A key challenge is to design sound, modular, lightweight mechanisms for establishing the unreachability of a block. Prior work in this area uses pointed-by assertions to keep track of the predecessors of every block, but is carried out in the setting of an assembly-like programming language. We take up the challenge in the setting of a high-level language, where a key problem is to identify and reason about the memory locations that the garbage collector considers as roots. For this purpose, we propose novel "stackable" assertions, which keep track of the existence of stack-to-heap pointers without explicitly recording their origin. Furthermore, we explain how to reason about closures—concrete heap-allocated data structures that implement the abstract concept of a first-class function. We demonstrate the expressiveness and tractability of our program logic via a range of examples, including recursive functions on linked lists, objects implemented using closures and mutable internal state, recursive functions in continuation-passing style, and three stack implementations that exhibit different space bounds. These last three examples illustrate reasoning about the reachability of the items stored in a container as well as amortized reasoning about space. All of our results are proved in Coq on top of Iris.

CCS Concepts: • **Theory of computation → Separation logic; Program verification.**

Additional Key Words and Phrases: separation logic, tracing garbage collection, live data, program verification

ACM Reference Format:
Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL, Article 25 (January 2023), 30 pages. https://doi.org/10.1145/3571218

1 INTRODUCTION

The most common aim of program verification is to establish the *safety* and *functional correctness* of a program, that is, to prove that this program does not crash and computes a correct result. In the area of deductive program verification [Filliâtre 2011], a program is usually verified with the help of a *program logic*, that is, a set of deduction rules whose logical soundness has been demonstrated once and for all. Separation Logic [Reynolds 2002; O'Hearn 2019] and Concurrent Separation Logic [Brookes and O'Hearn 2016; Jung et al. 2018] are examples of program logics that allow compositional reasoning (that is, reasoning about a program component in isolation) in the presence of challenging features such as dynamic memory allocation, mutable state, and shared-memory concurrency.

Authors' addresses: Alexandre Moine, Inria, Paris, France, alexandre.moine@inria.fr; Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, Strasbourg, France, arthur.chargueraud@inria.fr; François Pottier, Inria, Paris, France, francois.pottier@inria.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).

2475-1421/2023/1-ART25
https://doi.org/10.1145/3571218

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.

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

Thank you for your attention!

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

Logical Deallocation of Lists

$$\mathit{List}\ L\ \ell\ *\ \ulcorner \ell \notin \mathit{locs}(t) \urcorner\ *\ \ell \hookleftarrow_1 \emptyset \quad \Rrightarrow \quad \diamond(3 \times \mathsf{length}\ L)$$

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"

$$\textsc{LetAddSouvenir}\quad \frac{\langle R \cup \{\ell\} \rangle \, \{\Phi\} \; t_1 \; \{\Psi'\} \qquad \mathit{locs}(t_2) = \{\ell\} \qquad \forall v.\, \langle R \rangle \, \{\mathit{Stackable}\ \ell\ p * \Psi'\ v\} \; [v/x]t_2 \; \{\Psi\}}{\langle R \rangle \, \{\mathit{Stackable}\ \ell\ p * \Phi\} \; \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \; \{\Psi\}}$$

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"

$$\textsc{LetAddSouvenir}\quad \frac{locs(t_2) = \{\ell\} \qquad \langle R \cup \{\ell\} \rangle \, \{\Phi\} \, t_1 \, \{\Psi'\} \qquad \forall v.\, \langle R \rangle \, \{\mathit{Stackable}\ \ell\ p * \Psi'\ v\} \, [v/x]t_2 \, \{\Psi\}}{\langle R \rangle \, \{\mathit{Stackable}\ \ell\ p * \Phi\} \, \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \, \{\Psi\}}$$

$$\textsc{LetInSouvenir}\quad \frac{locs(t_2) = \{\ell\} \qquad \ell \in R \qquad \langle R \rangle \, \{\Phi\} \, t_1 \, \{\Psi'\} \qquad \forall v.\, \langle R \rangle \, \{\Psi'\ v\} \, [v/x]t_2 \, \{\Psi\}}{\langle R \rangle \, \{\Phi\} \, \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \, \{\Psi\}}$$

The NoFree Mode

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

$$\frac{\langle \bot \rangle \, \{\Phi\} \, t_1 \, \{\Psi'\} \qquad \forall v.\ \langle R \rangle \, \{\Psi' \, v\} \, [v/x]t_2 \, \{\Psi\}}{\langle R \rangle \, \{\Phi\} \, \text{let } x = t_1 \text{ in } t_2 \, \{\Psi\}} \; \textsc{LetNoFree}$$

A new consequence rule

New ghost update parameterized by the visible roots.

$$\frac{\Phi \Rrightarrow_{locs(t)} \Phi' \qquad \{\Phi'\}\ t\ \{\Psi\}}{\{\Phi\}\ t\ \{\Psi\}}$$

Our logical Free rule.

$$\ell \mapsto_1 [v_1, \ldots, v_n] \ * \ \ell \leftarrow\!\!\dashv_1 \emptyset \ * \ \ulcorner \ell \notin V \urcorner \ * \ \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow_V \quad \diamond n \ * \ \dagger\, \ell$$

References i

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.